TECHNISCHE UNIVERSITÄT WIEN

Master Thesis

Shape Optimization based on Reinforcement Learning

carried out for the purpose of obtaining the degree of Master of Science (MSc. or M.Sc.), submitted at TU Wien, Faculty of Mechanical and Industrial Engineering, by

Michael BINDER

Mat.Nr.: 01325632

[redacted]

[redacted]

under the supervision of

Stefanie Elgeti, Univ.Prof. Dr.-Ing.

Projektass. Daniel Wolff, M.Sc.

Institut of Lightweight Design and Structural Biomechanics

Vienna, November 2021

....

Stefanie Elgeti,
Univ.Prof. Dr.-Ing.
ILSB, E317
Gumpendorfer Straße 7 / Objekt 8
A- 1060 Vienna, Austria

....

Johannes Edelmann,
Univ.Prof. Dipl.-Ing. Dr.techn.
MEC, E325
Getreidemarkt 9/E325
A- 1060 Vienna, Austria

I confirm, that going to press of this thesis needs the confirmation of the examination committee.

Affidavit
I declare in lieu of oath, that I wrote this thesis and performed the associated research myself, using only literature cited in this volume. If text passages from sources are used literally, they are marked as such.
I confirm that this work is original and has not been submitted elsewhere for any examination, nor is it currently under consideration for a thesis elsewhere.
I acknowledge that the submitted work will be checked electronically-technically using suitable and state-of-the-art means (plagiarism detection software). On the one hand, this ensures that the submitted work was prepared according to the high-quality standards within the applicable rules to ensure good scientific practice "Code of Conduct" at the TU Wien. On the other hand, a comparison with other student theses avoids violations of my personal copyright.

Vienna, November, 2021

Signature

Contents

Abstract

The main focus of this thesis is to explore the feasibility of learning-based algorithms such as Reinforcement Learning (RL) as a data-driven alternative to classical optimization algorithms. For this, a simple geometry T-shaped geometry, which can be seen as an abstraction of the flow channel inside a profile extruder, is optimized with two different RL algorithms.
First, a test function for optimization is introduced to establish if the RL algorithm works and if the training of the algorithm can be improved. Based on this test function, a reward function is shaped, and a hyperparameter study is performed. The results show, that a dynamic reward function is most suitable for this task and show that the standard hyperparameter are good enough and do not need to be changed.
For the shape optimization task, a specific mass flow ratio between the two outflows of the geometry has to be configured. The flow channel geometry is parameterized by two different methods — one changes the corner points of the geometry directly, while the other one applies Free-Form Deformation (FFD). FFD deforms a box surrounding the object to change its shape. The experiments are carried out in order of increasing Degrees Of Freedom (DOF), as this turns out to be a measurement of the difficulty of the tasks. The RL algorithms are trained for a specific number of episodes and are evaluated if they can achieve the pre-defined goal of a specific mass flow ratio and if the learning decreases the number of time steps needed per episode.
The RL algorithms tested, namely Advantage Actor Critic (A2C) and Proximal Policy Optimization (PPO), can both achieve the pre-defined goals most of the time. In the tasks with the direct change of coordinates, the algorithms can improve their policy while their performance stays fairly constant for the task with the FFD, probably because it has too many DOF. In the test cases where the agents can improve their policy, the A2C agents outperforms the PPO agent.
The methods for shape optimization introduced in this thesis look very promising and, if further improved, could become a new standard for shape optimization tasks.

Kurzfassung

Das Hauptaugenmerk dieser Arbeit liegt in einer Machbarkeitsanalyse von lernbasierten Algorithmen wie RL als datengesteuerte Alternative zu klassischen Optimierungsalgorithmen. Dazu wird eine einfache T-förmige Geometrie, die als Abstraktion des Fließkanals innerhalb eines Profilextruders angesehen werden kann, mit zwei verschiedenen RL-Algorithmen optimiert.
Zunächst wird eine Testfunktion für die Optimierung eingeführt, um festzustellen, ob der RL-Algorithmus funktioniert und ob das Training des Algorithmus verbessert werden kann. Basierend auf dieser Testfunktion wird eine Belohnungsfunktion erstellt und eine Hyperparameterstudie durchgeführt. Die Ergebnisse zeigen, dass eine dynamische Belohnungsfunktion am besten für diese Aufgabe geeignet ist und dass die Standard-Hyperparameter gut genug sind und nicht geändert werden müssen.
Für die Formoptimierungsaufgabe muss ein bestimmtes Massenstromverhältnis zwischen den beiden Abflüssen der Geometrie konfiguriert werden. Die Fließkanalgeometrie wird durch zwei verschiedene Methoden parametrisiert — eine ändert die Eckpunkte der Geometrie direkt, während die andere FFD anwendet. FFD verformt eine das Objekt umgebende Box, um seine Form zu verändern. Die Experimente werden in der Reihenfolge der zunehmenden Freiheitsgrade durchgeführt, da sich dies als Maß für die Schwierigkeit der Aufgaben herausgestellt hat. Die RL-Algorithmen werden für eine bestimmte Anzahl von Episoden trainiert und daraufhin bewertet, ob sie das vordefinierte Ziel eines bestimmten Massenstromverhältnisses erreichen können und ob das Lernen die Anzahl der pro Episode benötigten Zeitschritte verringert.
Die getesteten RL-Algorithmen, nämlich A2C und PPO, können beide die vordefinierten Ziele die meiste Zeit über erreichen. Bei den Aufgaben mit direkter Änderung der Koordinaten können die Algorithmen ihre Strategie verbessern, während ihre Leistung bei der Aufgabe mit FFD ziemlich konstant bleibt, wahrscheinlich weil sie zu viele Freiheitsgrade hat. In den Testfällen, in denen die Agenten ihre Strategie verbessern können, übertrifft der A2C-Agent den PPO-Agenten.
Die in dieser Arbeit vorgestellten Methoden zur Formoptimierung sehen sehr vielversprechend aus und könnten, wenn sie weiter verbessert werden, zu einem neuen Standard für Formoptimierungsaufgaben werden.

Glossary

a	An Action
a_t	Action at timestep t
$\underline{A}_{\pi}(s,a)$	Advantage function
$A_{\pi}(s,a)$	Estimate of $\underline{A}_{\pi}(s,a)$
$\mathbb{E}$	Expectation value over a batch of samples
γ	Discount factor
$\mathcal{H}_1$	Space of once weakly differentiable and square-integrable functions
$\mathcal{L}_2$	Space of square-integrable functions
π_θ	Stochastic policy parameterized by θ
r	A reward
r_t	Reward at timestep t
G_t	Return
s	A state
s_t	State at timestep t
θ	Parameters of the policy
$\underline{V}_{\pi}(s)$	Value function
$V_{\pi}(s)$	Estimate of $\underline{V}_{\pi}(s)$
$\underline{Q}_{\pi}(s,a)$	Action-value function
$Q_{\pi}(s,a)$	Estimate of $\underline{Q}_{\pi}(s,a)$
χ	Coordinates inside a splines parameter space
n_{cp}	Number of control points
Ω	Initial configuration
Ω_{param}	Parameter space of a spline

ξ . . . . Spatial coordinates inside the Fourier space

Acronyms

Notation	**Description**
A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
AI	Artificial Intelligence
ASCII	American Standard Code for Information Interchange
BFGS	Broydon-Fletcher-Goldfarb-Shanno
CFD	Computational Fluid Dynamics
CNN	Convolutional Neural Network
DL	Deep Learning
DOF	Degrees Of Freedom
DRL	Deep Reinforcement Learning
DV	Design Variable
FE	Finite Element
FEM	Finite Element Method
FFD	Free-Form Deformation
FR	Fletcher-Reeves
ILSVRC	ImageNet Large-Scale Visual Recognition Challenge
LBB	Ladyzhenskaya-Babuška-Brezzi condition
LSTD	Least-Squares Temporal Difference
MC	Monte Carlo evaluation
MDP	Markov Decision Process
ML	Machine Learning
NN	Neural Network
NURBS	Non-Uniform Rational B-Spline
PDE	Partial Differential Equation
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RSS	Residual Sum of Squares
RWTH	Rheinisch-Westfälische Technische Hochschule
SB3	`Stable Baselines3`
SGD	Stochastic Gradient Descend
TD	Temporal Difference learning
TRPO	Trust Region Policy Optimization

List of Figures

List of Tables

1. Motivation

The engineering world is becoming more and more complex and often surpasses the engineering intuition. Numerical methods have become essential in handling this complexity. This particular thesis is concerned with the topic of shape optimization in numerical design.
Shape optimization is used and needed in numerous fields, to name a few: it can reduce the consumption of cars by making them more aerodynamic or reduce the weight of planes while maintaining the same structural integrity. In the field profile extrusion, it is used to modify the die to produce a correct profile. Classical shape optimization methods of profile extrusion dies can require a lot of manual work. RL algorithms are mainly automatic and require not much user intake as soon as the task is set up. In this thesis, it is evaluated if learning-based optimization algorithms such as RL can be used as a data-based alternative to classical optimization algorithms. While not necessarily superior to classic optimization algorithms (such as gradient-based) for one single optimization problem, we expect RL techniques to thrive when similar optimization tasks are repeated. In the context of this work, no transfer learning tasks were investigated, but rather the extent to which the shape optimization problem can be transferred into a reinforcement learning framework and whether this type of problem can be solved with a reinforcement learning algorithm is investigated.
The shape optimization problem, which is investigated in this thesis, is the mass flow ratio of a simple T-geometry, which can be seen as an abstraction of the flow channel inside a profile extruder. Finding the right parameterization technique is a big part of classical shape optimization. We want to find such a technique, which is adequate for the task posed, and explore if the number of DOF is important.
RL is based on trial and error interaction of an agent with an environment. For each interaction, the agent is informed about a reward and the subsequent state of the environment, but there is no information about long-term interests as classical optimization algorithms would provide.
Chapter 2 gives an overview about classical shape optimization methods of profile extrusion dies. Some Machine Learning (ML) methods are introduced, as well as a closer look at RL and the algorithms used. Chapter 3 deals with the methods used in the context of this work. Direct geometry parameterization and FFD is used to parameterize the geometry to be optimized. The steady Stokes problem is introduced in Section 3.2. Section 3.3 deals with the interaction between the RL agent and the Finite Element Method (FEM) solver. In Chapter 4, the results of the tests are displayed, starting with an overview of the tests performed. In Section 4.2 a test function for optimization is analyzed in detail and a sample episode is discussed. Section 4.3 discusses the results of the actual shape optimization tests. In the end of this chapter, the runtime between the different tests is compared as well as the actual shapes, produced by the RL agent.

2. State of the Art

This section introduces the terminology and principles required to understand the concepts of the algorithms used in this thesis. First, an overview of classic shape-optimization methods is given. The next section focuses on machine learning principles. In the last section the two algorithms used in this thesis are explained in detail.

2.1. Classic shape optimization of profile extrusion dies

In this section, some shape optimization techniques for profile extrusion die design are introduced. Profile extrusion is a production technique for manufacturing continuous products. Plastic melts are forced through an extrusion die to create a profile of specific shape and thickness. A water bath or spray chamber then cools the extruded shape and often provides pressure or vacuum controls to properly size the product as it passes through. Belts or cleated pullers apply smooth tension on the product and keep it moving in pace with the extruder. A cutter or saw creates the final product length. Figure 1 depicts this process.

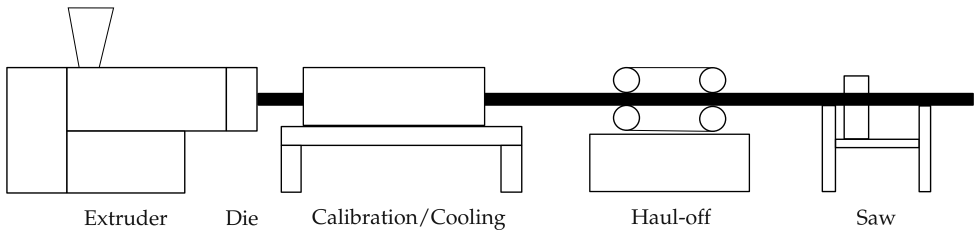


Figure 1: Extrusion line for the production of thermoplastic profiles.

The strongly nonlinear behavior of plastic melts, namely shear-thinning and the viscoelastic properties, make the extrusion die design a challenging task. Traditional methods rely heavily on the designers' experience and need time-consuming experiments for each new die. A few different approaches to shorten this development time will be introduced in the following.
First, we need to define what the goal of a correct profile is. According to [13], a profile is appropriate when "[...] (i) the skeleton line of the cross-section matches the designed one, and (ii) the thickness distribution is as required." Producing the correct profile is the task of the die, the calibrator freezes this profile and makes some minor corrections to the outer shape of the profile. Problems with these goals occur through unsatisfactory flow distribution at the die exit and through elastic stresses induced by flow in the melt at the die exit. This leads to some possible criteria for good design from which objective function definitions can be derived. In the paper which will be discussed now, the homogeneity of the velocity distribution at the die outlet is considered to be the

primal quality criterion. This is usually used in the least squares term as an objective function ([14], [15], [16]).
In [15] the three-dimensional die geometry is split up into a series of two-dimensional 'die-slices'. Each of these die slices is then divided into i partitions and a local objective function $F_{obj(i)}$ is introduced. The idea behind this function is that the mass flow fraction distribution of each die slice should be as close as possible to the area fraction distribution of the product geometry.

$$F_{obj(i)} = \left(\frac{Q_i}{A_i} - 1\right) \times 100\% \qquad (1)$$

Through Finite Element (FE) analysis, the area fraction A_i and the mass flow fraction Q_i are determined for each product partition i. The global objective $F_{obj(global)}$ function is a weighted, squared sum of the local objective functions of all partitions and parts. The goal is to optimize the global objective function in an optimization loop using finite element analysis. Four different strategies to achieve a correct profile are introduced. The strategies, are evaluated in the paper according to the achieved quality, computational cost and user interaction. The two overall best performing methods, namely a height approximation method and a global scheme which decouples the Design Variables (DVs) are discussed here.
A standard optimization cycle, depicted in Figure 2, is implemented for this task. In the preprocessing step, the DVs are selected. The mesh is then generated and a flow solver is used to determine the velocity field. The optimization algorithm determines the objective function and returns the new DVs which alter the geometry in a step called parameterization.

Optimization schemes

In the height approximation method, the cross-sections are approximated with rectangular shapes which can be solved analytically. This works especially well for partitions, where the shapes are similar to a rectangle. If they are different however a characteristic height, which should be a function of the DVs has to be found manually. This can require a lot of manual work. The parallel decoupled scheme uses an objective function for each partition of the die slices and optimizes all of them at the same iteration. The coupling between different DVs is estimated using a reference position. For more details see [15].

Comparison with a manual optimization

In a manual optimization, which is already in industrial use, no initial preprocessing has to be done. However, the parameterization is not automatic and all required geometry changes have to be done manually, which takes much more time. The quality of the design measured on the objective function is very similar.

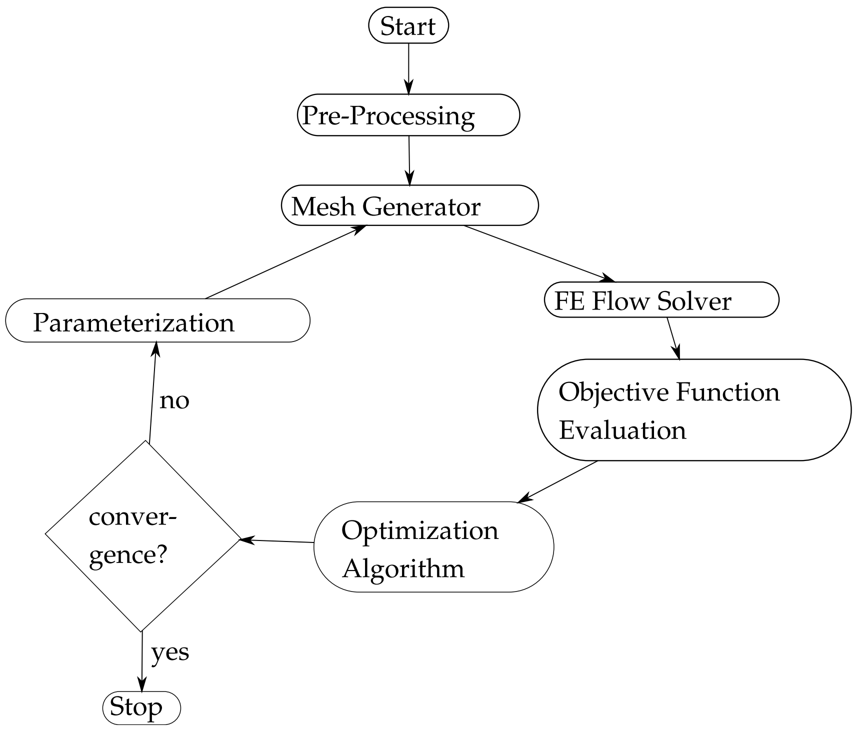


Figure 2: Schematic shape optimization cycle according to [1].

2.2. Machine Learning

In recent years, there has been an increasing interest in artificial intelligence, which is mostly based on deep learning. The decisive event was an image classification competition in 2012 called ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The aim of the competition was to classify 1.2 million images into the 1000 different classes. A deep neural Network called AlexNet won the competition by a great margin. The winning top-5 test error rate was 15.3%, compared to 26.2% achieved by the second-best competitor, which is considerably better than the previous state-of-the-art [17].

Differences between AI, ML, DL and RL:
It is hard to define AI exactly, but a general understanding is that the objective of AI is to endow machines with human intelligence [2]. ML is a method for realizing this goal using algorithms to learn from data, make decisions and predictions. Deep Learning (DL) is a machine learning technique inspired by the way the human brain filters information [18]. RL is a technique of machine learning as well. DL and RL can be combined to Deep Reinforcement Learning (DRL) [2]. Figure 3 shows the connection between these fields of artificial intelligence.

Before going into the details about machine learning we must define what learning is: "Learning is the process of acquiring new understanding, knowledge, behaviors, skills, values, attitudes, and preferences." [19]
This means a machine or computer program should learn from experience and change

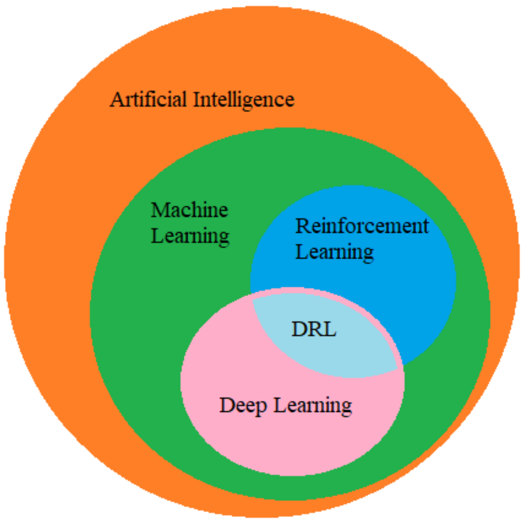


Figure 3: Connections between the different fields of AI [2].

its behavior according to that. Instead of programming machines statically, we want to use techniques which help computers to learn behavior from data. The data represents the experiences the machine makes. It is possible to let algorithms that use neural networks learn either continuously, use transfer learning, or train a behavior once with the help of data and then freeze it. To freeze a strategy simply means saving the weights from the neural network layers. Transfer learning is an area of ML which uses the knowledge from an old problem to solve a new problem. It is for example possible to train a Convolutional Neural Network (CNN) algorithm on image recognition of cats and dogs and reuse this already trained network to classify images of letters, using the weights of the old task as a starting point for the new task [3]. Continual learning is a bit more challenging. Reusing the example above, the Neural Networks (NNs) should not only fulfill the new task (classifying the letters) but should still be able to perform well on the previous task (classifying images). This proves to be challenging, but a few strategies have been developed already [20].
A big advantage of ML is that the programmer doesn't have to know the underlying relation between the inputs and outputs, but can let the machine infer a model of this relation from the data.
In ML, there are three different types of learning-paradigms: supervised learning, unsupervised learning and reinforcement learning. All of them have in common that they want to find a function mapping input variables to output variables. These learning paradigms differ in the definitions of the input and output sets, as well as in data required to learn this function [3].

2.2.1. Supervised Learning

In supervised learning, the inputs as well as the outputs of a component can be observed [21]. This means every supervised learning paradigm needs a set of labeled data. Using the example in the introduction ILSVRC: The inputs $X = (x_1, ...x_n)$ are the (centered) raw RGB values of pixels. The output variable is the class of the picture, for example cat or dog. For the input variables there are different names: predictors, independent variables, features, or just variables. The output variable is called response or dependent variable and typically denoted by Y [22].
In the ILSVRC, the winning AlexNet CNN algorithm used a fixed resolution of $256 \times 256 \times 3$ pixels, and then cropped out the central patch consisting of 224×224 pixels from the resulting image. This means, that for each picture, the first layer had $n = 224 \times 224 \times 3 = 150528$ independent variables but only one response, the classification Y[1] [17].
A lot of the classical statistical learning methods like logistic- and linear regression operate in the supervised learning category [22].
Linear regression is a simple supervised learning paradigm. It tries to find a linear relationship between X and Y. There are a number of use cases, like evaluating trends, making estimates and forecasts. A more specific example is: determining a correlation between house prices and the crime rate per capita. Linear regression has been around for a long time and can be used as a good starting point for newer approaches.
The goal of linear regression is predicting a quantitative response Y on the basis of a feature variable X. The most basic form called simple linear regression uses a single feature variable X. A linear relationship between X and Y is assumed. Mathematically, it can be written as Equation (2) [22]:

$$Y \approx \beta_0 + \beta_1 X \tag{2}$$

The character $\approx$ means: "is approximately modeled as". In an imaginative example that is based on the Boston housing data frame, the prices of the houses directly correlate to the crime rate per capita. In this example, we can write Equation (2) as Equation (3):

$$houseprice \approx \beta_0 + \beta_1 \times crimerate \tag{3}$$

The variable β_0 is the house price if the crime rate is zero, called intercept, β_1 is the slope term in the linear equation. To distinguish the exact values from approximated

[1]In this special example, a softmax was used to calculate the probability for any chosen image to be in any of the 1000 classes. The 5 most probably classes were then used in the output to calculate the top-5 test error rate. Top-5 error rate is the fraction of test images for which the correct label is not among the five labels considered most probably by the model [17].

model the hat symbol "^" is used in the equations for the model (Equation (4)). The variable $\hat{y}$ is a prediction of Y on the basis of $X = x$.

$$\hat{y} = f(x) = \hat{\beta}_0 + \hat{\beta}_1 x \tag{4}$$

The goal is to find coefficient estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ such that the resulting line is as close as possible to the $n = 506$ data points. Before getting into details about how to estimate the coefficients, let us recall that for supervised learning, a set of labeled data with the inputs as well as the outputs is needed. In this example, the data can be written as:

$$(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$$

This means there are m observation pairs, each consists of a measurement of X and a measurement of Y. In the Boston housing example, the values for X would be the crime rate per capita and Y would be the house price.
Closeness can be defined in a number of different ways, a common approach is using the squared Euclidean distance (method of the least squares), also used in Section 2.2.2 for the clustering algorithm. Figure 4 shows an example of a linear regression on artificial data. The features x_i correlate approximately linear with the response y_i. The dotted line shows the residual, which is the difference between the ith observed response value y_i and its ith predicted response value $\hat{y}_i$ (Equation (5)) [22].

$$r_i = y_i - \hat{y}_i = y_i - f(x_i) \tag{5}$$

Using the square of the residuals has the effect that errors over 1 get enhanced and

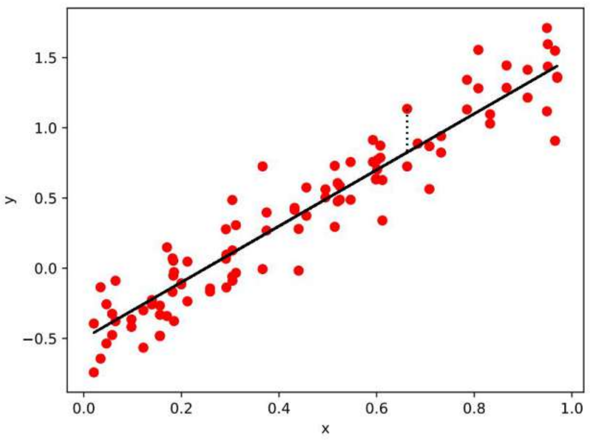


Figure 4: Example of a linear regression on artificial data [3].

errors under 1 get weakened. The least squared approach minimizes the Residual Sum of Squares (RSS) as Equation (6) [3]:

$$\sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{6}$$

Using some calculus, the estimated coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$ can be calculated according to Equation (7) [22]:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^2}{\sum_{i=1}^{n}(x_i - \bar{x})} \qquad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x} \tag{7}$$

The variables $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ and $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ are the sample means.
In practice, there is usually more than one feature for each dependent variable. For example, in the Boston housing data frame, there are 13 features for each dependent variable. The good thing is that the simple linear regression model can be easily extended to a multiple linear regression model. Each feature of the features vector gets a separate slope coefficient in a single model. Having p distinct predictors Equation (4) takes the form of Equation (8):

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + ... + \hat{\beta}_p x_p \tag{8}$$

This gives each feature of the predictor measurements x_j a different coefficient β_j. The values of $\hat{\beta}_0, \hat{\beta}_1, ..., \hat{\beta}_p$ can be calculated using multiple least squares regression coefficient estimates [22].

2.2.2. Unsupervised Learning

In the unsupervised learning category, there is no information given about the correct outputs available [21]. Every observation X_j has a vector of p features x_i for $i = 1, ...n$, measured on m observations $j = 1, ...m$, but no associated response Y_j. It is called unsupervised because we lack a response variable to supervise our analysis, i.e. no response variable can be predicted. One statistical learning tool that might be used in this setting is cluster analysis, or clustering. In this approach, on the basis of the features, we try to find out if the observation fall into distinct groups [22].
This can be used for a variety of tasks, like optimizing the marketing strategy by recommending similar products or products that are often bought together. Furthermore, it can be used in medicine, grouping similar symptoms together and finding the root cause.
The goals of cluster analysis are maximizing similarities within the groups and minimizing the similarities to different groups [3]. There are a few different types of clustering analysis like connectivity-based clustering, centroid-based clustering or distribution-based clustering. To keep it short, this section describes a rather simple centroid-based clustering analysis called *k-means clustering*.
The goal of k-means clustering is to separate the observations X_j into K cluster. To perform k-means clustering we must specify the desired number of clusters first, then the k-means algorithm will assign each observation to exactly one of the K clusters [22]. Each cluster C_k has a representative prototype μ_k, which doesn't have to be in the

original data but is a mean value of the data from the cluster [3]. The criteria for the quality of the split is that the sum of the derivations from the cluster representatives in the chosen metric is minimal. Mathematically speaking, a minimum of Equation (9) has to be calculated.

$$W = \sum_{k=1}^{K} \sum_{X_j \epsilon C_k} d(X_j, \mu_k) \tag{9}$$

The term $d(X_j, \mu_k)$ is the distance in the metric. Often, the method of least squares(squared Euclidean distance) is used for the metric Equation (10) [22]. This is also called clustering through variance minimization, as the sum of the variances of the clusters gets minimized.

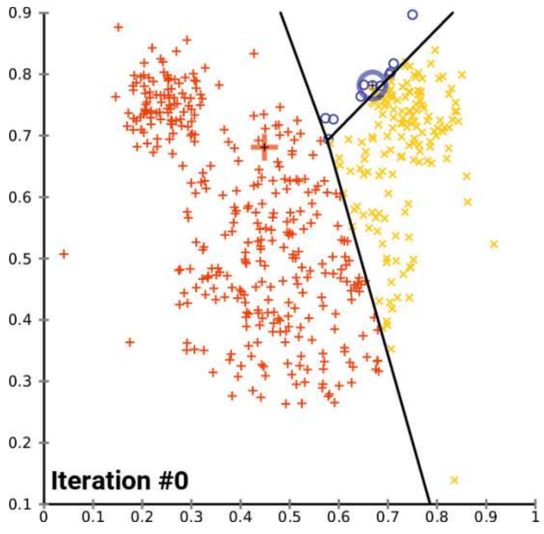
$$\begin{gathered} d(X_j, \mu_k) = \|X_j - \mu_k\|^2 : \\ W = \sum_{k=1}^{K} \sum_{X_j \epsilon C_k} \|X_j - \mu_k\|^2 \end{gathered} \tag{10}$$

The Lloyd algorithm is the standard algorithm for k-means [3, 22]. The procedure for the Lloyd algorithm is shown in Table 2

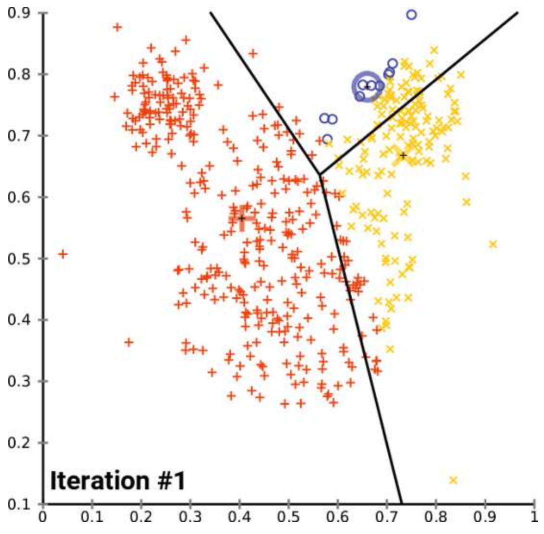
1. Initialize k representatives μ_k for the cluster.
2. Assign each element X_j to the cluster for which the distance to the cluster centroid μ_k is the smallest.
3. Calculate by averaging the new representatives μ_k for the clusters.

Table 2: Procedure of the Lloyd algorithm [3].

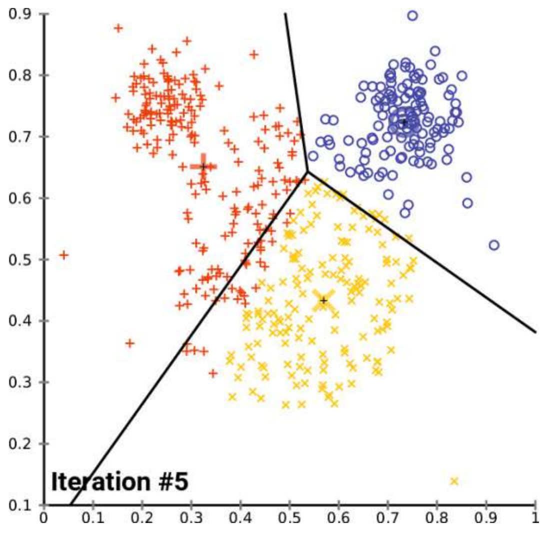
The steps 2 and 3 are repeated until the position of the cluster representatives does not change anymore.
Figure 5 shows an example of the iteration process involved in the Lloyd k-means clustering analysis. In the first step, Figure 5(a) the cluster centroids are initialized randomly, and the closest elements are assigned to the 3 different groups. The averaging new representatives are then calculated and used in Figure 5(b). The result after 5 iterations is shown in Figure 5(c). This process is repeated till the position of the cluster representatives is stable Figure 5(d).



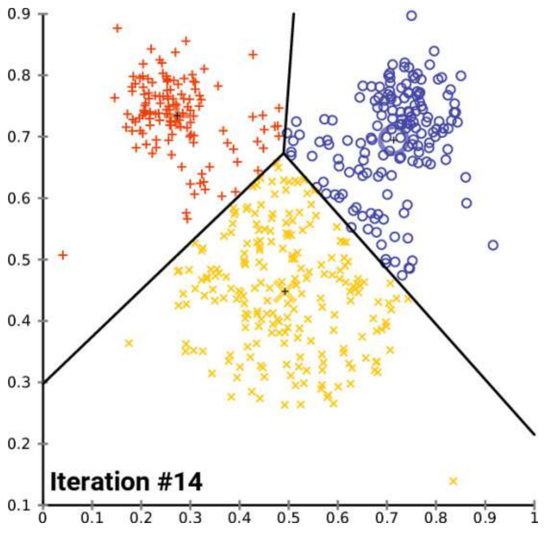
(a) Iteration 0 (initialisation).



(b) Iteration 1.



(c) Iteration 5.



(d) Iteration 14 (final).

Figure 5: Lloyd k-means algorithm, different iteration stages until it converges [4]. The clusters' mean values are shown as black crosses.

2.3. Reinforcement Learning

The difference between reinforcement learning and supervised learning is, that in RL the agent (autonomous entity that acts according to a rule) just receives an evaluation of the action but not what the correct action should have been, i.e. there are no input/output pairs given. Consider the following example: The agent drives a car with a specific speed at a time t (state) and has to learn to brake (action) in order to stop in 10 m (desired outcome) but with the action chosen it stops in 15 m (actual outcome) and rear-ends another car. After some time, the agent gets an evaluation of its underlying actions (huge bill for rear-ending a car) but is not told the correct actions (braking earlier) [21]. Through exploration, the agent has to find, the best actions to take for each state. The idea behind it is that rewarding or punishing an agent for its behavior increases the probability of the agent to repeat or stop said behavior [23].
RL methods have a wide variety of use cases. They can be used to teach computers how to control robots in a simulation or play sophisticated strategy games. In October 2015 AlphaGo, a RL-Algorithm for playing Go, beat the reigning European Champion with a score of 5-0 [5].
Another difference to supervised learning is that the online performance is important. Online learning means processing the data as it is made available, whereas offline learning collects all the data first, then learns from it. The evaluation of the system is often done concurrently with learning [24].
In the example above, some terminology has already been introduced, but we will define them more clearly here as they are substantial for the rest of the thesis. The main terms that are needed are: action(a_t), state(s_t), reward(r_t), agent and environment, where the subscript $_t$ refers to "at a time step t". The environment is the world that the agent lives in and interacts with. The agent uses a policy, which is a rule determining which actions to take in a certain state. The agent makes an action which influences the environment. As feedback, it gets back the current state of the environment as well as a reward. The reward tells the agent how good the current state of the environment is. The goal of the agent is to maximize the cumulative reward, called return [5]. In Figure 6 this interaction loop is depicted.
Another important term in RL is the trajectory τ. It is defined as "a sequence of states and actions in the world that happened from s_0, a_0 onwards" [5]:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{11}$$

Figure 7 visualizes this definition using an example of a robotic arm. The agent starts in a state s_0 and takes an action a_0. This leads the agent to the next state s_1 where it takes an action a_1. In Figure 7 the red dashed line shows the trajectory. The variables denoted with $_T$ are the terminal state variables, i.e. when the sequence of actions and states is finished. According to [8] everything the agent cannot control is considered

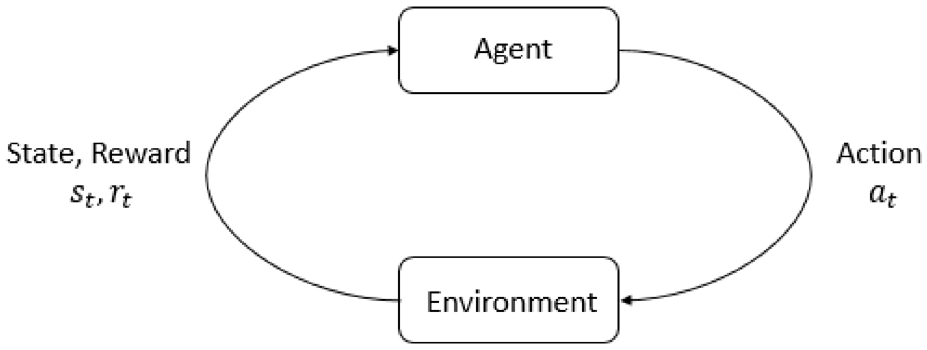


Figure 6: Agent environment interaction loop [5].

part of the environment. In the example of the robot arm, the motors of the robot can be considered part of the agent while the exact functioning of them is beyond the agent's control and part of the environment [7].
In Figure 8 possible interactions between the agent and the environment are depicted. The agent chooses an action in each state and receives a scalar reward for each action, as well as some information about the current state of the environment. In this case, a discrete model is used with distinct numbers for each state and action.
A key concept in RL called Markov Decision Process (MDP) can be explained with Figure 7 as well. Transitions, i.e. going from state s_t to state s_{t+1}, only depend on the most recent state and action and no prior history i.e. it doesn't depend on the trajectory. This means the system obeys the Markov property [25]. Many problems have been successfully modelled in terms of MDP, in fact MDPs have become the de facto standard for learning sequential decison making [7].

The goal of the agent is to maximize the cumulative reward over a trajectory. The reward-function R depends on the current state of the world s_t, the action a_t just taken and the next state of the world s_{t+1} (Equation (12)):

$$r_t = R(s_t, a_t, s_{t+1}) \tag{12}$$

For the cumulative reward, the sum over all rewards attained over the course of the trajectory is used. Most of the time the infinite-horizon discounted return is needed, which is the sum of discounted future rewards. Returns at successive time steps are related to each other according to Equation (13) [8].

$$G_t = r_{t+1} + \gamma r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = r_{t+1} + \gamma G_{t+1} \tag{13}$$

A discount factor is needed because a reward obtained at the current time t is better than a reward in the future. Moreover, it is also mathematically convenient as it converges to a finite number [8].

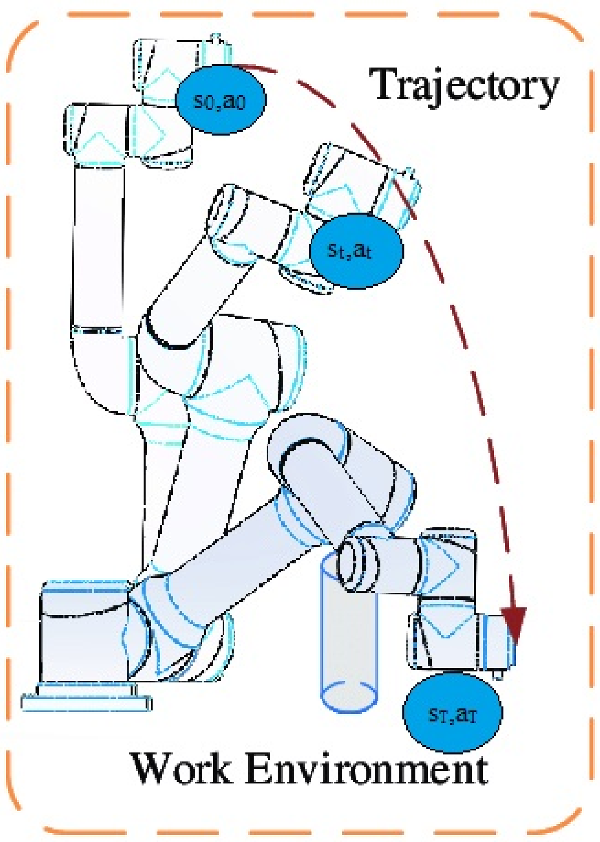


Figure 7: Exemplary trajectory of a robotic arm [6] (edited).

The agent needs rules that determine which actions to take in order to maximize the return. These rules are called policy. It can either be deterministic (usually denoted by μ) or stochastic (denoted by π). In DRL parameterized policies are used e.g. the parameters of the policy are the weights and biases of a neural network. These parameters can be adjusted to change the behavior via some policy optimization. The parameters are denoted by θ [5]:

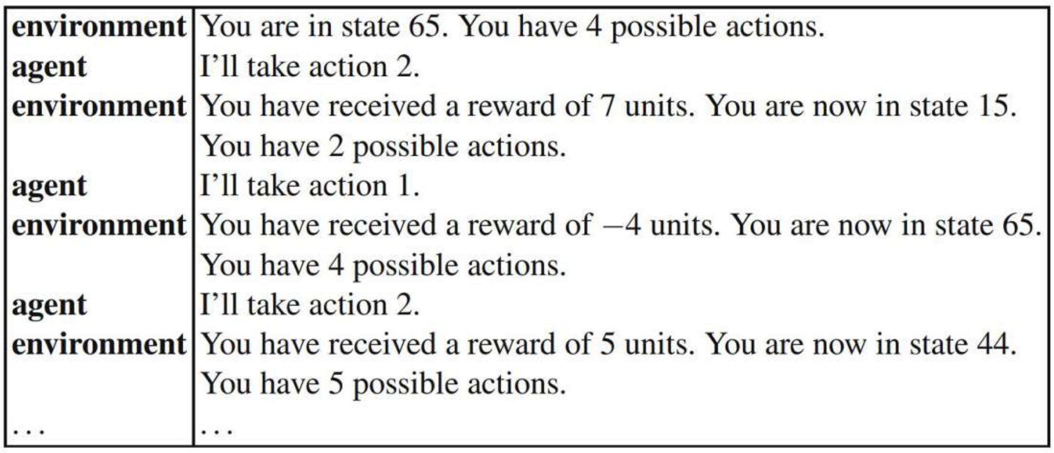
$$a_t \sim \pi_\theta(\cdot|s_t) \tag{14}$$

The agent needs to estimate how good it is to be in any given state. For this, the value-function is needed. Precisely the (state)-value function determines the expected return for any given state. The expected return depends on the actions it is going to take, which are defined by the policy [8].
The state-value is the expected return if we are in a state s at a time t, $s_t = s$:

$$\underline{V_\pi}(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{15}$$

Sometimes we need to know how much better it is to take a specific action a at a time step t over selecting a random action. For this, we define the action-value function:

$$\underline{Q_\pi}(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{16}$$



environment	You are in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 7 units. You are now in state 15. You have 2 possible actions.
agent	I'll take action 1.
environment	You have received a reward of −4 units. You are now in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 5 units. You are now in state 44. You have 5 possible actions.
...	...

Figure 8: Example of an interaction between an agent and its environment [7].

In order to determine how much better said action is, the difference between the action-value function and the value function has to be calculated. This is called the advantage function $A_\pi(s,a)$ [26]:

$$\underline{A_\pi}(s,a) = \underline{Q_\pi}(s,a) - \underline{V_\pi}(s) \tag{17}$$

In Equation (13), a recursive relationship between successive time steps has been established. Value functions satisfy similar recursive relationships. Equation (18) shows the Bellman equation for $\underline{V_\pi}$, which is the basis for solving sequential decision problems [8]:

$$\underline{V_\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')], \text{ for all } s \in \mathcal{S} \tag{18}$$

For any policy π and any state s this consistency condition holds between the value of s and the value of its possible successor states s'. The function $p(s',r|s,a)$ is the probability of transitioning to state s' with reward r, from state s under taking action a. The final expression can be read as the expected value: for each triple (a, s' and r) the probability $\pi(a|s)p(s',r|s,a)$ is calculated, then the quantity in brackets is weighted in by that probability and finally the sum over all possibilities is made to get the expected value [8]. For the action-value function Q_π a similar equation can be derived. The Bellman equations help to establish a connection between the state-value and state-action-value functions, as well as expressing a relationship between the value of a state and the values of its successor states. It decomposes the value function into the immediate reward plus the discounted future values [8].
Figure 9 shows a graphical representation of this equation. Update rules are also known as backup operations as they transfer information back from future states to the current one [27]. The top node s is the starting state. Based on the policy π the agent chooses an action a. Depending on the transition probability p in the environment, the next state s' along with the reward r is reached. The value of the start state

equals the (discounted) value of the expected next state, plus the reward expected on the way.

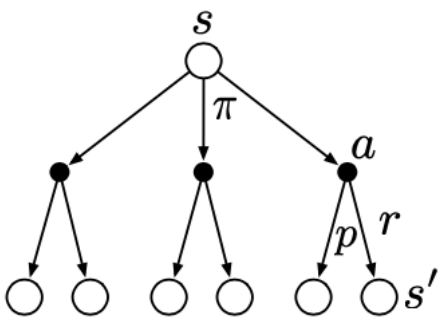


Figure 9: Backup diagram for V_π [8] (edited).

Exploration and Exploitation Trade-off:
In order to get the best possible reward, an agent has to exploit actions it has used in the past and that turned out to be effective in producing a high reward, but also explore new actions that might yield an even bigger reward. This can result in a worse performance, as the actions might be less good than the current policy. Without trying them, however, it might never find possible improvements. Furthermore, in a non-stationary environment, the agent has to keep exploring to keep the policy up to date. In order to learn, it has to explore, but in order to perform well it has to exploit what it already knows. This dilemma is called the exploration-exploitation problem [7].
This trade off is fundamental to many RL algorithms, so we will discuss a few strategies using an example: the k-armed bandit problem, named by the analogy to a slot machine called "one-armed bandit", except that it has k levers instead of one. Each action refers to playing on one of the slot-machine's levers and the rewards are the payoffs for hitting the jackpot. After enough action selections, the agent should learn to concentrate on the lever with the highest win rate [8].
The simplest policy to solve this problem is the greedy policy, where the agent always chooses the action with the highest estimated action value.

$$a_t = \underset{a}{\operatorname{argmax}}\, Q_\pi(s_t, a) \tag{19}$$

Another basic exploration strategy is the epsilon-greedy policy. It tells the agent in which percentage of the time it should act according to the current policy, and how often it should perform a random action instead. The parameter used is ε. For example, an agent with $\varepsilon = 0.1$ would perform a random action 10% of the time [8]. A third policy called decaying-epsilon-greedy method is introduced where epsilon slowly decays overtime [9].
In Figure 10 the actions performed by agents with different epsilon greedy parameters on a 10-armed bandit problem are shown. The greedy policy explored very little and settled on choosing action 5 very quickly. Both the epsilon-greedy and the

decaying-epsilon-greedy algorithms found the optimal action(action 7) but continued to explore. The amount of exploring in the decaying-epsilon-greedy algorithm is reduced over time. Towards the end of the simulation, it is choosing the optimal action almost all the time. In Figure 11 the average rewards for the three policies over time are depicted. The greedy algorithm converges to a sub-optimal point, while the epsilon-greedy and the decaying-epsilon-greedy paradigms converge to a higher average reward. The epsilon-greedy algorithm will have a lower average reward as it always explores with a certain probability, while the decaying-epsilon-greedy paradigm reduces these random actions over time, thus concentrating on taking the optimal action.

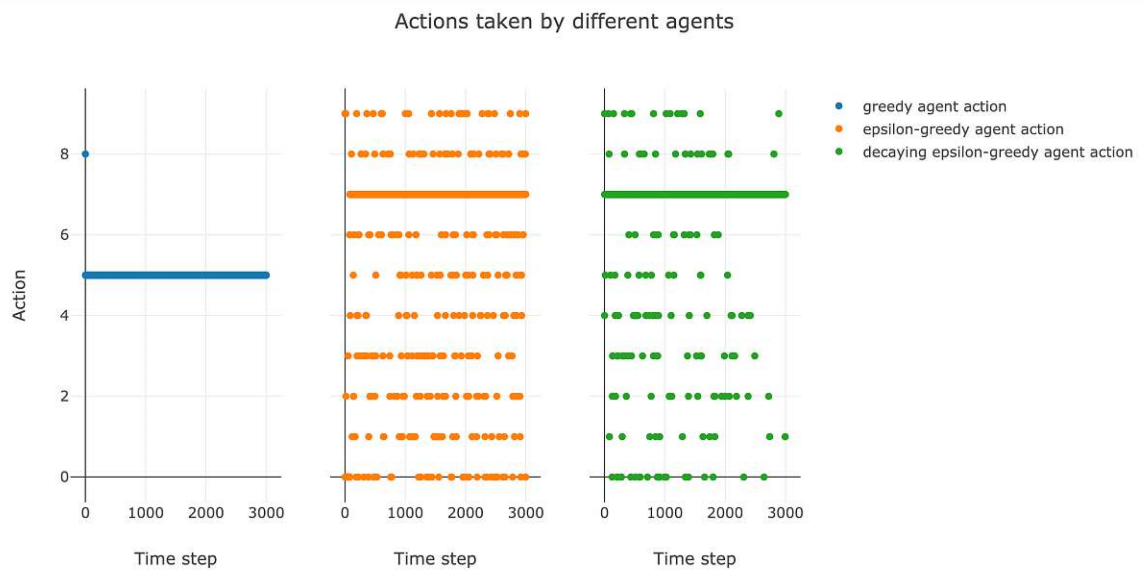


Figure 10: Actions performed by different epsilon greedy agents [9].

Classification of RL Algorithms

Figure 12 shows a non-exhaustive overview of some popular reinforcement learning algorithms. One way to distinguish RL algorithms is by dividing them into two groups: model based and model-free algorithms. A model based algorithm knows a function which predicts state transition and rewards. This allows the agent to plan ahead and results in a substantial improvement of the sample efficiency. Usually such a model is not available[5].

The program in this thesis uses model-free algorithms, namely A2C and PPO, which are not as efficient as model-based paradigms but easier to implement. The next sections will explain the ideas behind these approaches.
Another distinction between the algorithms can be made in the way the agent is learning. It can be either on-policy or off-policy. To understand the difference more easily, two terms are introduced: target policy and behavior policy. The target policy is the policy that the agent is trying to learn, i.e., the agent is learning a value function for this policy. The behavior policy is the policy the agent is using to select actions, it is

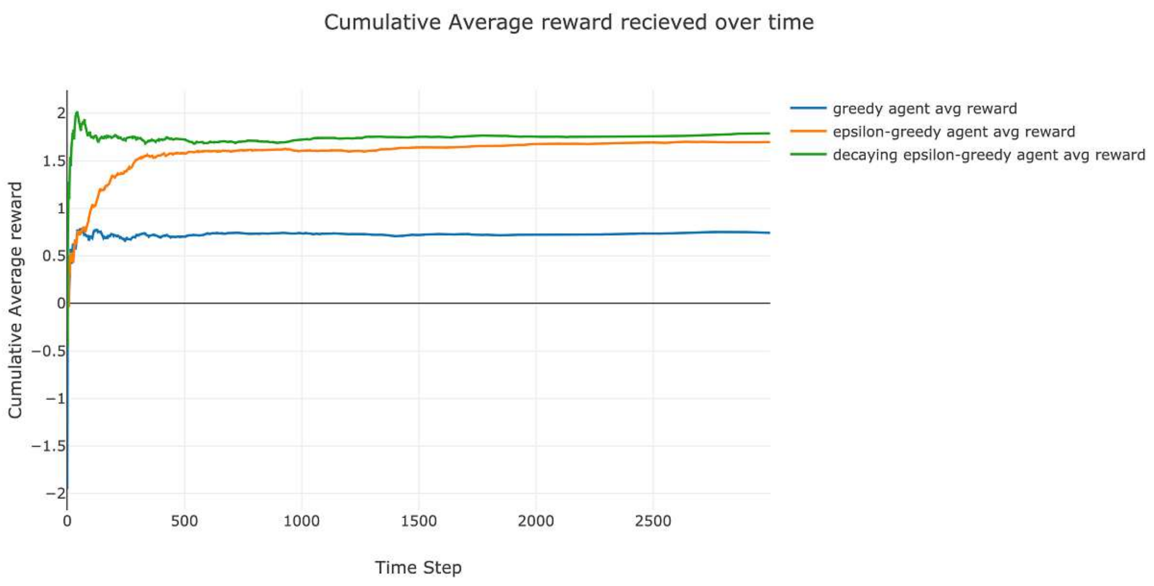


Figure 11: Average rewards for three policies [9].

using this policy to interact with the environment.
On- policy learning algorithms use the same policy to select actions that is being improved, i.e.: the target policy is the same as the behavior policy. On-policy methods are generally simpler and considered first. The algorithms used in this thesis are on policy. Off-policy algorithms evaluate and improve a policy that is different from the one being used to select actions. Examples are Q-learning or deep Q-learning.
Advantages of off-policy algorithms are that they can learn from stored data, while on-policy algorithms have no replay buffer and learn directly from whatever the agent encounters in the environment. This makes on-policy algorithms less sample efficient, as each batch of experience is just used once and is discarded once the policy update has been completed [28].

2.3.1. Proximal Policy Optimization - PPO

This section summarizes the paper [29] which describes the paradigm of PPO used in this thesis. PPO is a DRL algorithm proposed by OpenAi in 2017. It is a new family of policy gradient methods for reinforcement learning which alternates between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Instead of performing one gradient update per data sample like standard policy gradient methods, we alternate between sampling data from the policy and performing multiple epochs of minibatch updates in order to enhance sample efficiency. PPO has the same motivation as Trust Region Policy Optimization (TRPO), taking the biggest possible improvement step on a policy, on data we currently have, without stepping so far that we accidentally cause performance collapse. To elaborate this a bit: if the policy gets an update that pushes it into a

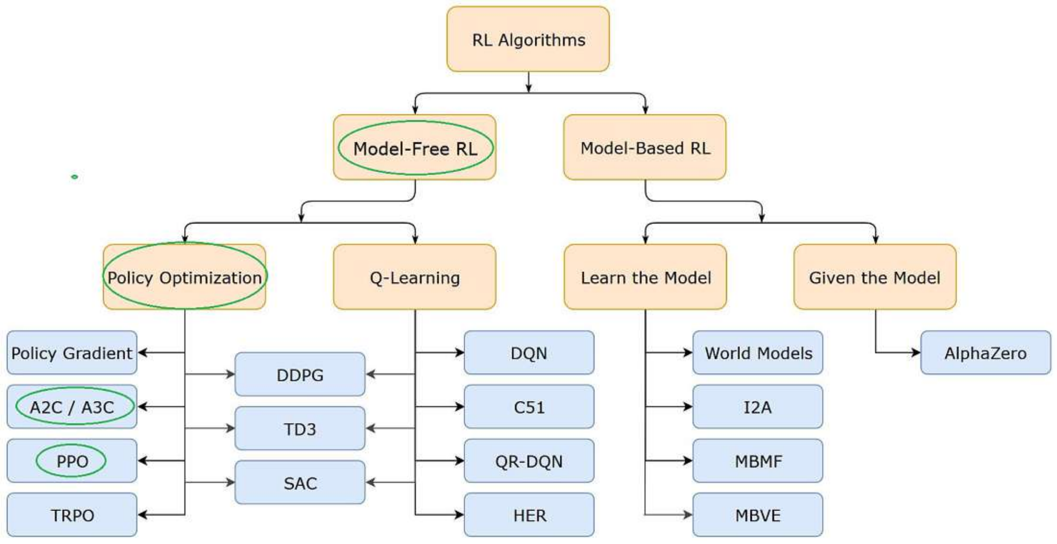


Figure 12: Taxonomy and overview of popular RL algorithms [5].

region of the parameter space where it is going to collect the next batch of data under a very poor policy, it could happen that it can't recover from this. The new methods are simpler to implement, more general and have better sample complexity while keeping some benefits of TRPO. Table 3 shows an overview of the main goals and the attributes for the implementation used in `stablebaselines3`, used in this thesis.

Main goals	Attributes
Easy implementation	On-policy algorithm
Sample efficient	Discrete and continuous action-spaces
Ease to tune	Multi processing available

Table 3: Overview about the main objectives in PPO.

The way policy gradient algorithms work is by using a stochastic gradient ascend algorithm on an estimator of the policy gradient. A commonly used gradient estimator is given by Equation (20):

$$g_\pi = \mathbb{E}[\nabla_\theta \log \pi_\theta(a_t|s_t) A_\pi(s_t, a_t)] \tag{20}$$

The variable $A_\pi(s_t, a_t)$ is an estimator of the advantage function at time step t, the expectation $\mathbb{E}$ is the empirical average over a batch of samples. The objective function is a function whose gradient is the policy gradient estimator. According to [29] the estimator g_π is obtained differentiating the objective Equation (21):

$$L^{PG}(\theta) = \mathbb{E}[\log \pi_\theta(a_t|s_t) A_\pi(s_t, a_t)] \tag{21}$$

While it is possible to perform multiple steps of optimization on this objective L^{PG} using the same trajectory, it often leads to large policy updates that can cause performance collapse [29]. Even small changes in the parameters θ of the policy can lead to big changes in the behavior of the policy, therefore it is not sufficient to just limit the changes of the parameters.
The way PPO solves this issue is by using a different (surrogate) objective function. In the paper two primary variants: PPO-Clipping and PPO-Penalty are introduced. The PPO-Clipping version will be explained further as it is used in the `stablebaselines3` implementation, which is used in this thesis.
First, a probability ratio $r_t(\theta)$ is introduced:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{22}$$

If the probability of taking a specific action a_t in a state s_t with the new policy π_θ is higher than it would have been with the old policy $\pi_{\theta_{old}}$ then $r_t(\theta) > 1$ holds.
In TRPO a "surrogate" objective gets maximized:

$$L^{CPI}(\theta) = \mathbb{E}\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_\pi(s_t, a_t)\Big] = \mathbb{E}[r_t(\theta)A_\pi(s_t, a_t)] \tag{23}$$

CPI stands for conservative policy iteration [30]. Without a constraint, the policy update could be too big. Therefore, L^{CLIP} is introduced:

$$L^{CLIP}(\theta) = \mathbb{E}[\min\big(r_t(\theta)A_\pi(s_t, a_t), clip\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)A_\pi(s_t, a_t)\big)] \tag{24}$$

With this approach, the maximum policy update is limited to $1-\epsilon, 1+\epsilon$, where ϵ is the clipping range that can be adjusted to the problem setting [29]. [5] found a considerably simpler version Equation (25) which is also implemented in the code used:

$$L(s, a, \theta_{old}, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\pi_{\theta_{old}}}(s, a), g(\epsilon, A_{\pi_{\theta_{old}}})\right) \tag{25}$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

If the advantage is positive, i.e. the selected action is better than selecting a random action, Equation (25) reduces to Equation (26)

$$L(s, a, \theta_{old}, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, (1+\epsilon)\right) A_{\pi_{\theta_{old}}}(s, a) \tag{26}$$

If the action becomes more likely, i.e. $\pi_\theta(a|s)$ increases, the objective increases. The min in this term puts a limit to how much the objective can increase. The maximum it can increase is limited by the clipping parameter ϵ in $(1+\epsilon)$. This means the policy does

not benefit from going far away from the old policy.
Assuming the advantage is negative, i.e. the selected action is worse than selecting a random action, Equation (25) reduces to Equation (27):

$$L(s, a, \theta_{old}, \theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, (1-\epsilon)\right) A_{\pi_{\theta_{old}}}(s, a) \tag{27}$$

The objective will increase if $\pi_\theta(a|s)$ decreases, i.e. the action becomes less likely. The max term puts a limit on how much the objective can increase. If $\pi_\theta(a|s)$ gets lower than $(1-\epsilon)\pi_{\theta_{old}}(a|s)$ the max kicks in and the term gets reduced to $(1-\epsilon)A_{\pi_{\theta_{old}}}(s, a)$. This limits again how far the new policy can step away from the old policy [5].
To sum it up: clipping limits how much the policy can change, and the hyperparameter ϵ corresponds to this upper and lower bounds. The final version of the clipped PPO function is given by Equation (28), which is approximately maximized in each iteration, a loss and an entropy function are added to Equation (24):

$$L_t^{CLIP+VF+S}(\theta) = \mathbb{E}[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{28}$$

$L_t^{VF}(\theta)$ is a squared-error loss Equation (29) and updates the baseline estimate determining how good it is to be in this state, i.e.: what is the average count of discounted rewards that is expected to be obtained from this point onward.

$$L_t^{VF}(\theta) = (V_\theta(s_t)) - V_t^{targ})^2 \tag{29}$$

The last term $S[\pi]_\theta(s_t)$ is called the entropy term, and it is added to encourage more exploration. The parameters c_1 and c_2 are further hyperparameters that can be adjusted to the problem [29]. On this surrogate objective, multiple epochs of minibatch updates can be applied. The optimization can be done either by Stochastic Gradient Descend (SGD) or usually for better performance, Adam [31].
Adam is an optimization algorithm that can be used instead of the classical SGD procedure to update network weights iterative based on training data. The method is computationally efficient and has little memory requirements [31]. Without going into the exact detail about how Adam works, just the hyperparameter learning rate (α_t in [31]) or Adam stepsize (as called in [29]) is introduced. Basically, the learning rate determines how much the weights of a neural network can change in each step. This controls how quickly or slowly a neural network learns a problem. The optimal learning rate is usually close to the maximum learning rate that doesn't cause divergence. A heuristic for choosing the learning rate would be starting with a larger learning rate and if the training criterion ($L^{CLIP+VF+S}$ in the PPO example) diverges α_t can be reduced [32].
This method computes individual adaptive learning rates for different parameters. The name Adam is derived from adaptive moment estimation [31].

2.3.2. Advantage Actor Critic - A2C/A3C

A2C/A3C is a conceptually simple and lightweight framework for DRL that uses (asynchronous) gradient descent for optimization of deep neural network controllers. The advantage actor critic has two main variants: the A3C and the A2C.
The core idea of an actor-critic algorithm is to use a separate memory structure for the policy and for the action-value function. The policy structure selects actions and is known as the actor $\pi_\tau(s,a)$, the estimated action-value function $Q_\pi(s,a)$ criticizes said actions and is known as the critic. In the advantage actor-critic paradigm, the action-value function is replaced by an estimator $A_\pi(s,a)$ of the advantage function $A^\varphi(s,a)$. This reduces the variance of the policy gradient. To elaborate: the variance comes from the fact that we learn stochastic policies in stochastic environments, i.e. two similar trajectories can have completely different returns depending on the stochasticity of the policy, the transition probabilities and the probabilistic rewards. Adding a baseline can reduce this variance [10].
$A_\pi(s,a)$ is called the advantage estimate and can be computed with different methods like Monte Carlo evaluation (MC) advantage estimate, Temporal Difference learning (TD) advantage estimate or as in the case of A2C and A3C with the n-step advantage estimate given in Equation (30).

$$A_\pi(s,a) = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_\pi(s_{t+n+1}) - V_\pi(s_t) \tag{30}$$

N-step advantage estimating relies on n-step updating, which is a trade-off between MC and TD.

- MC waits until the end of an episode to update the value of an action using the sum of obtained rewards $R(s,a)$.
- TD updates immediately using the immediate reward $r(s,a,s')$ and approximates the rest with the value of the next state $V_\pi(s)$.
- n-step uses the n next immediate rewards and updates the rest with the value of the state visited n steps later.

Inserting the n-step advantage estimate Equation (30) into the policy gradient estimator Equation (21) gives us Equation (31):

$$g_\pi = \mathbb{E}\Big[\nabla_\theta \log \pi_\theta(s_t,a_t)\Big(\sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_\pi(s_{t+n+1}) - V_\pi(s_t)\Big)\Big] \tag{31}$$

TD can be seen as a 1-step algorithm, as just the immediate reward is used. MC learns from complete episodes. In the n-step estimation there is no need for finite episodes and a trade-off between bias (wrong updates based on estimated values as in TD) and variance (variability of the returns as in MC) [10].
The architecture of A2C/A3C is shown in Figure 13. The actor outputs the policy π_θ

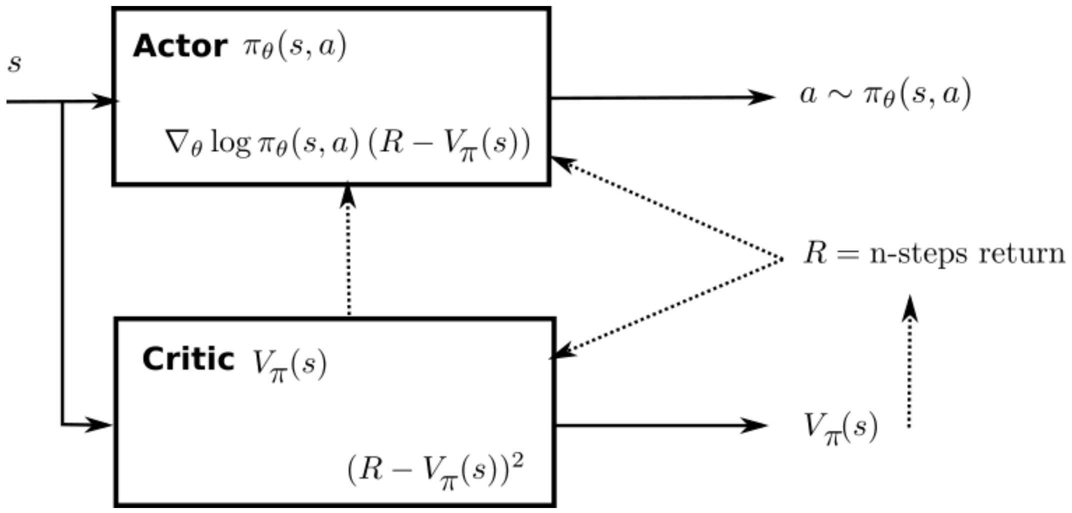


Figure 13: Advantage actor critic architecture (edited) [10].

for a state s, i.e. a vector of probabilities for each action. The critic outputs the value $V_\pi(s)$ of a state s [10].
A simplified pseudo-algorithm for a single A2C worker could look like Table 4:

Multiple workers perform these updates in parallel. In the asynchronous version A3C the workers aren't synchronized as shown in Figure 14. Each worker receives at the beginning of each episode a copy of the actor and critic weight from the global network. As soon as the worker is finished sampling an episode and computing the gradients, it send it back to the global network. In the synchronized version, a coordinator would wait for all workers to finish and merge the gradients, while in the asynchronous version this step is skipped. In Figure 15 this difference is depicted.

According to [33] the synchronous A2C version performs better than asynchronous implementations. It is more cost-effective when using single-GPU machines, and is faster than a CPU-only A3C implementation when using larger policies.

1. Acquire a batch of transitions (s, a, r, s') using the current policy π_θ.
2. For each state encountered, compute the discounted sum of the next n rewards $\sum_{k=0}^{n} \gamma^k r_{t+k+1}$ and use the critic to estimate the value of the state encountered n steps later $V_\pi(s_{t+n+1})$.

$$R_t = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_\pi(s_{t+n+1})$$

3. Update the actor using Equation (31).
4. Update the critic to minimize the TD error between the estimated value of a state and its true value.

$$\mathcal{L}(\pi) = \sum_t (R_t - V_\pi(s_t))^2$$

5. Repeat.

Table 4: Simplified pseudo-algorithm for a single A2C worker [10].

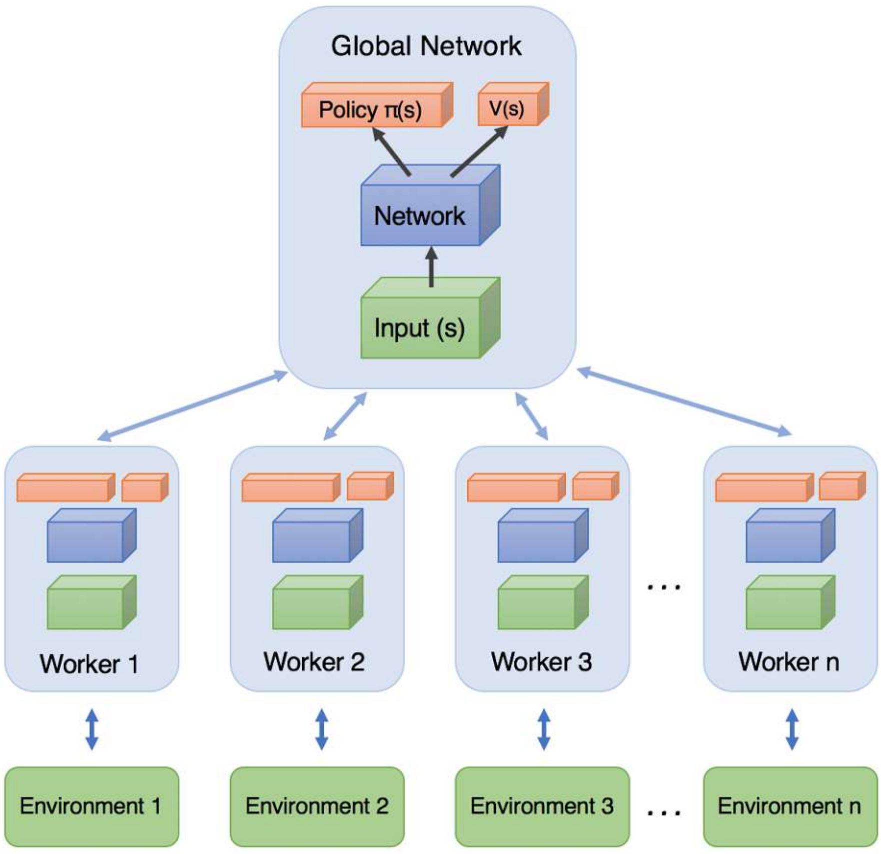


Figure 14: A3C architecture [11].

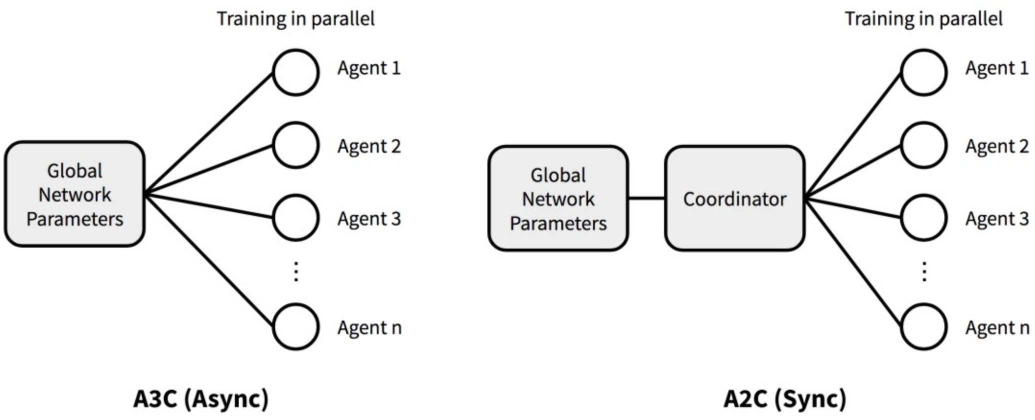


Figure 15: Difference in architecture between A3C and A2C [12].

2.3.3. Related works

To the best knowledge of the author, at the time of writing, there are few contributions proposed to exploit RL for problems related to shape optimization, which all focus on airfoil design. Two contributions are concerned with the optimization of morphing airfoils using Q-learning methods with two [34] and four [35] control parameters. The contribution [36] used a PPO algorithm aiming to reduce the drag of airfoils. [37] used RL techniques for a typical aerodynamic shape optimization of missile control surfaces with Computational Fluid Dynamics (CFD). Transferring the learned policy, remarkably reduced the requirements for CFD calls and time cost. [38] used a PPO algorithm to learn the policy of supercritical airfoil drag reduction. The latest publication [39] focuses on multi-object aerodynamic design optimization using the PPO algorithm as well. The PPO agent learns to maximize the lift-to-drag ratio in two states while restricting the thickness. According to this paper the agent requires less computational time and achieves up to 10% improvements of the aerodynamic performance compared with the results of baseline.

3. Methodology

This particular thesis is concerned with the topic of shape optimization in numerical design by means of RL. This is evaluated by modifying a simple geometry, which can be seen as an abstraction of the flow channel inside a profile extruder. The geometry is parametrized with two different methods. For the first method the coordinates of the corner points are directly altered, for the second parameterization the geometry is modified by the means of FFD. The mode of operation for the FFD is explained in detail in the following. Next, the foundations of fluid dynamics, which equations describe our problem and how they can be solved, are declared. In this case, the behavior of fluids is analyzed using the FEM. Then, the interaction between the RL agent and the CFD software used is explained. At the end of the section, some specifics for the RL agent, namely the action-space and the reward function, are shown. These methods are needed to understand the dynamics of the experiments, which will be explained in detail in Chapter 4.

3.1. Shape optimization by means of FFD

Shape optimization by means of FFD means that instead of manipulating the coordinates of the corner points of the geometry directly, the control point coordinates of a deformation spline function are modified. This is afterwards used to transform the whole two-dimensional mesh, yielding the modified geometry. This has the advantage

that the mesh does not need to be created new each time [40].
FFD was developed in 1986 by T. Sederberg and S. Parry. It is a technique in computer graphics for changing an object in 2D or 3D. The method deforms a box surrounding the object, making it a purely geometric method that can be applied to any finite object. It can deform surface primitives as well as implicit defined surfaces and can be applied either locally or globally. The continuity of derivatives can be preserved as well as volume [41]. In the original paper [41] a Bernstein polynomial is used to construct a mapping from $\mathbb{R}^3$ to $\mathbb{R}^3$ through a triviate tensor product. For this thesis, B-Splines are used instead of a Bernstein polynomials. The equation and the notation for the B-Spline are adopted from [40].
In the first step, the flow channel Ω_{phys} which will be represented by a computational mesh $\Omega_{\#1} \subset \Omega_{phys}$ is scaled to fit inside a 2D unit cube. The spatial coordinates of the mesh in the initial geometry are denoted by $\mathbf{x}_{\#1}$. This is done by a simple linear transformation, depicted in Figure 16. The equation for this transformation is:

$$\mathbf{T} : \Omega_{\#1} \to \Omega_{\#2}, \mathbf{T}(\mathbf{x}_{\#1}) = \mathbf{A} \cdot \mathbf{x}_{\#1} + \mathbf{b} \tag{32}$$

$$\mathbf{x}_{\#2} := \mathbf{T}(\mathbf{x}_{\#1}) \tag{33}$$

The transformation matrix is denoted by $\mathbf{A}$, $\mathbf{b}$ is a constant vector, the spatial coordinates of the mesh in the second domain $\Omega_{\#2} \subseteq [0,1]^2 \subseteq \Omega_2$ are denoted by $\mathbf{x}_{\#2}$. This step of fitting the mesh into a unit cube is taken, so the spline used in this thesis can be used for any geometry that is scaled to these dimensions.
For the second step, a B-Spline is defined:

$$\mathbf{x}_3 : \Omega_{para} \to \Omega_3, (\xi, \eta) \mapsto \mathbf{x}_3(\xi, \eta) = \sum_{i=1}^{n} \sum_{j=1}^{m} N_{i,p}(\xi) M_{j,q}(\eta) \mathbf{B}_{i,j} \tag{34}$$

$$\mathbf{x}_{\#3} := \mathbf{x}_3(\xi, \eta) \tag{35}$$

The domain of the undeformed mesh is denoted by $\Omega_{\#2} \subseteq \Omega_{para} = [0,1]^2$, the domain of the deformed mesh is denoted by $\Omega_{\#3} \subset \Omega_3$. Ω_{para} is the parameter space of the (deformed) spline, and Ω_3 is the image domain of the spline. The variable $\mathbf{x}_{\#3}$ denotes the spatial coordinates of the mesh inside $\Omega_{\#3}$. The variables ξ and η are the coordinates in the parameter space of the spline. The variables q and p are the polynomial orders of the B-Spline basis functions which are two in our example. The number of basis functions used to construct the B-Spline surface are denoted with n and m, they are three each. $N_{i,p}, i = 1, 2, ..n$ and $M_{j,q}, j = 1, 2, ...m$ are the basis functions for the B-Spline and $\mathbf{B}_{i,j}$ are the corresponding control points. The spline is used as a mapping which transforms the grid, representing the geometry, inside the parameter space of the spline into the image space of the spline Ω_3. Formally speaking, we have:

$$\mathbf{x}_3(\Omega_{\#2}) = \Omega_{\#3} \tag{36}$$

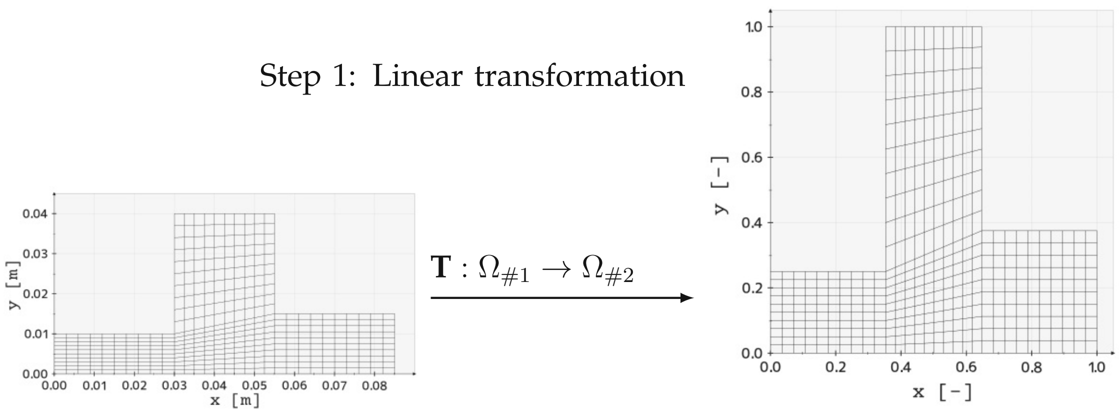
This means the deformed, scaled mesh of the grid $\Omega_{\#3}$ is the image of the mesh $\Omega_{\#2}$ under the function $\mathbf{x}_3(\xi, \eta)$. This transformation is depicted in Figure 17a and Figure 17b.

Figure 17c shows the undeformed and Figure 17d the deformed B-Spline.
In the third step, the mesh is transformed back with a linear transformation. The transformation matrix is the inverse of the matrix **A**:

$$\mathbf{T}^{-1} : \Omega_{\#3} \to \Omega_{\#4}, \mathbf{T}^{-1} = \mathbf{A}^{-1} \cdot (\mathbf{x}_{\#3} - \mathbf{b}) \tag{37}$$

$$\mathbf{x}_{\#4} := \mathbf{T}^{-1}(\mathbf{x}_{\#3}) \tag{38}$$

This linear back transformation is depicted in Figure 18. The coordinates of the final mesh are denoted with $\mathbf{x}_{\#4}$ and are in the domain $\Omega_{\#4} \subset \Omega_{phys}$.
In this approach, the algorithm changes the coordinates of the control points $A - I$ (Figure 19) in order to change the whole geometry. The control polygon is depicted by the black lines. The points can be moved vertically and horizontally, yielding 36 DOF. The red boxes indicate the limits of each control point, preventing too large deformation that might lead to errors. This approach can be used on a lot of different geometries and is not limited to the T-geometry of the extruder shape in the example.

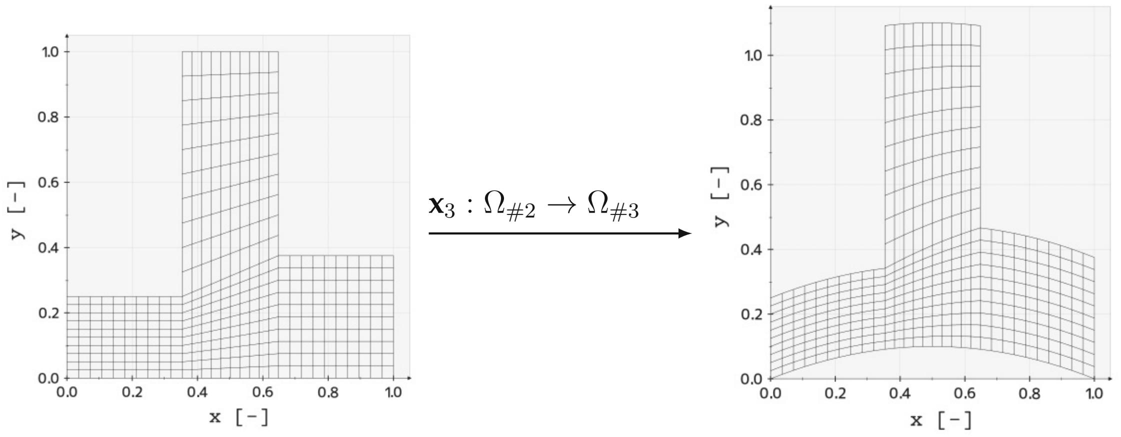


(a) Intitial geometry.

(b) Scaled geometry

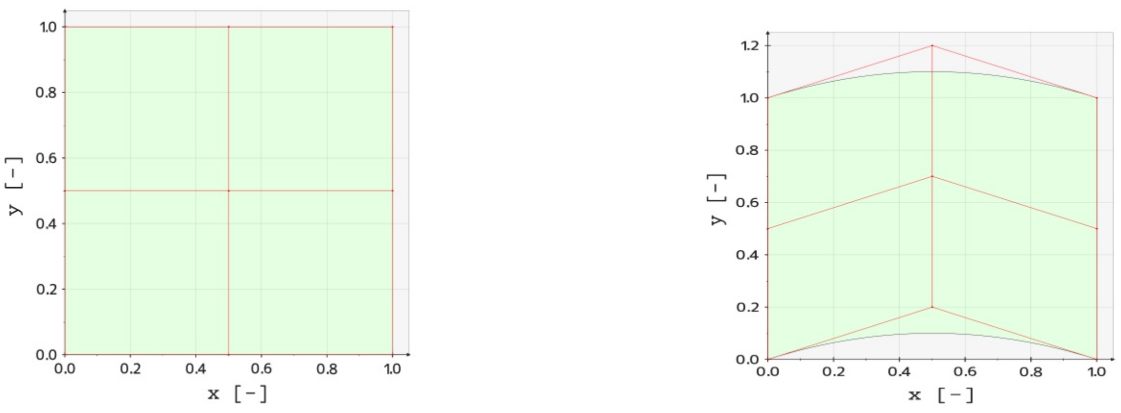
Figure 16: Linear transformation of the computational mesh to fit inside a 2D unit cube.

Step 2: B-Spline transformation



(a) Scaled geometry.

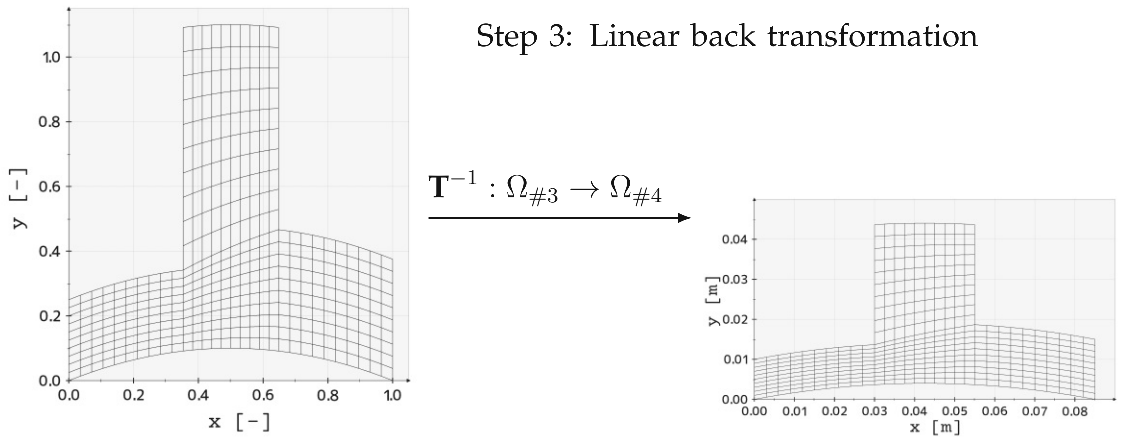
(b) Scaled and deformed geometry.



(c) Undeformed Spline.

(d) Deformed Spline.

Figure 17: B-Spline transformation of the computational mesh.



(a) Scaled and deformed geometry.

(b) Final geometry

Figure 18: Linear back transformation of the computational mesh.

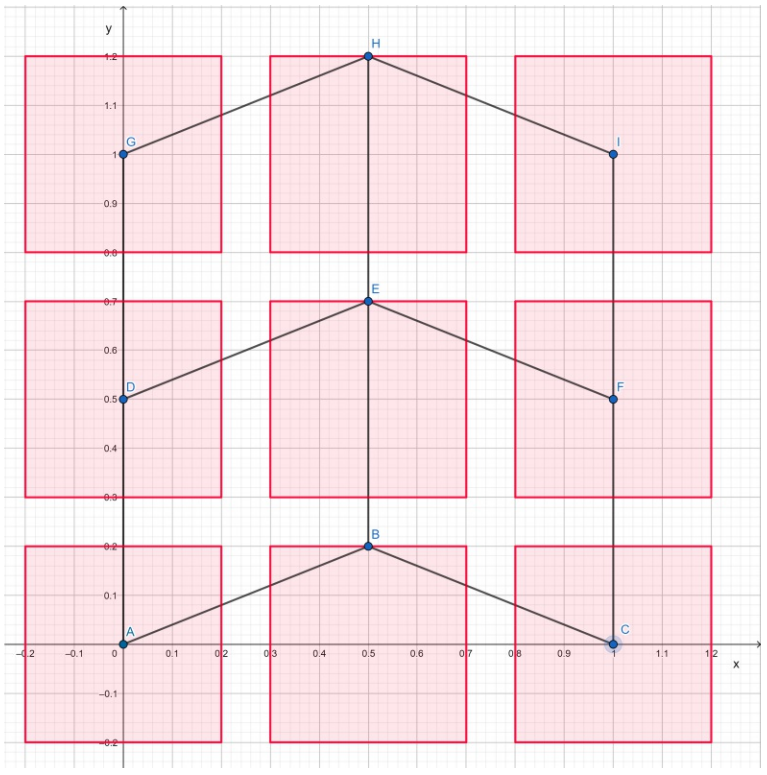


Figure 19: Control points and control polygon.

3.2. FEM for flow problems

In our environment, the use of FEM is indispensable, as it is not possible to solve the problem analytically. The fluid in the extruder is a non-Newtonian fluid, which means the viscosity of the material is non-linearly dependent on the strain rate [42]. FEM is a widely used method for numerically solving differential equations. The basic approach is splitting a larger system into smaller subsystems/elements that can be solved.
The fundamental basis of fluid dynamics are the Navier-Stokes equations. These equations provide a highly accurate physical theory that can predict nearly all phenomena in flows of liquid [43]. The following equations are adopted from [44] for a stationary, incompressible fluid, where the convective term can be neglected. The body force **b** and the traction force **t** are zero. This example solves the following Partial Differential Equation (PDE) problem:

$$\nabla \cdot \mathbf{v} = 0 \text{ on } \bar{\Omega} \tag{39}$$
$$-\nabla \cdot \boldsymbol{\sigma} = \mathbf{0} \text{ on } \bar{\Omega} \tag{40}$$

Equation (39) is the formula for mass conservation. The variable **v** the velocity vector. The flow region is denoted by $\bar{\Omega}$ which is the domain occupied by the fluid. Equation (40) is the formula for the conservation of momentum. The variable $\boldsymbol{\sigma}$ is the Cauchy stress tensor. As boundary conditions, we get:

$$v_y = -0.45 \text{ on } \Gamma_1 \tag{41}$$
$$v_x = 0 \text{ on } \Gamma_1 \tag{42}$$
$$-\boldsymbol{\sigma} \cdot \mathbf{n} = \mathbf{0} \text{ on } \Gamma_2 \cup \Gamma_3 \tag{43}$$
$$\mathbf{v} = \mathbf{0} \text{ on } \Gamma_4 \tag{44}$$

The boundary $\Gamma = \partial\bar{\Omega}$ of the fluid domain is assumed to be a closed and sufficiently regular surface. In Figure 20 the boundareis are visualized. Γ_1 is illustrated in red, Γ_2 in green, Γ_3 in yellow and Γ_4 in blue. Γ_4 is the wall of the geometry where the velocity is zero. The variable v_y is the speed of the mass input flow in y direction. The unit outward normal vector to Γ is denoted by **n**.

The constitutive laws for the Cauchy stress tensor $\boldsymbol{\sigma}$ are:

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\epsilon} - p\mathbf{I} \tag{45}$$
$$\boldsymbol{\epsilon} = \frac{1}{2}(\nabla\mathbf{v} + \nabla\mathbf{v}^T) \tag{46}$$

The pressure field is denoted by p, **I** is the unit matrix, $\boldsymbol{\epsilon}$ is the rate-of-strain tensor, the viscosity is denoted by μ. The material parameters have been chosen, such that the material resembles the properties of polymers commonly used in profile extrusion. The density has been chosen as $\rho = 850\frac{kg}{m^3}$.

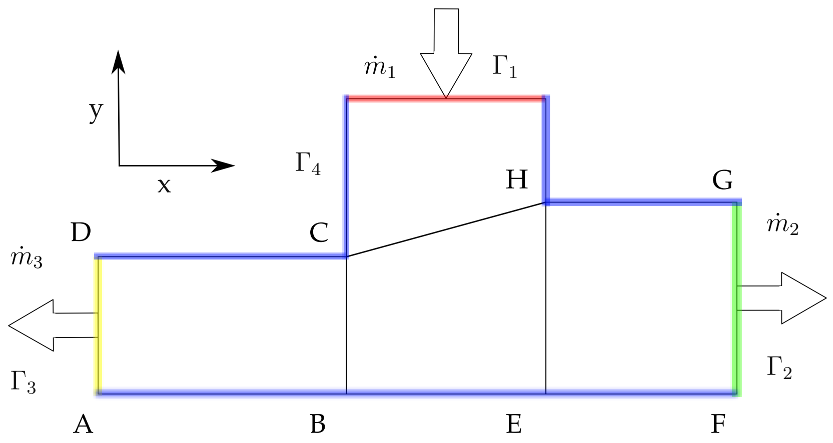


Figure 20: Visualisation of the boundaries Γ_1 to Γ_4.

Symbol	Description	Value	Unit
A	Zero-shear viscosity	18900	$\frac{kg}{ms}$
B	Reciprocal transition rate	0.355	$\frac{1}{s}$
C	Slope of viscosity curve in pseudoplastic region	0.700	-

Table 5: Parameter values of the Carreau-Yasuda model.

In order to model the behavior of molten polymer correctly, we consider a varying viscosity according to the Carreau-Yasuda shear-thinning model:

$$\mu = \frac{A}{(1+B\dot{\gamma})^C} \tag{47}$$

$$\dot{\gamma} = \sqrt{2\boldsymbol{\epsilon} : \boldsymbol{\epsilon}} \tag{48}$$

The parameter values of the Carreau-Yasuda model are chosen according to Table 5. To solve these equations using a FEM approach, we first need to derive a weak formulation. The required function spaces are taken from [44] as:

$$\boldsymbol{\mathcal{S}} := \{\mathbf{v} \in \mathcal{H}^1(\bar{\Omega}) \mid \mathbf{v} = \mathbf{v}_D \text{ on } \Gamma_D\} \tag{49}$$

$$\boldsymbol{\mathcal{V}} := \{\boldsymbol{\omega} \in \mathcal{H}^1(\bar{\Omega}) \mid \boldsymbol{\omega} = \mathbf{0} \text{ on } \Gamma_D\} \tag{50}$$

$$\mathcal{Q} := \mathcal{L}_2(\bar{\Omega}) \tag{51}$$

$$\Gamma_D = \Gamma_1 \cup \Gamma_4 \tag{52}$$

Using these function spaces, we can now multiply Equation (39) and (40) with the test functions $q \in \mathcal{Q}$ and $\boldsymbol{\omega} \in \boldsymbol{\mathcal{V}}$ and integrate them over the computational domain $\bar{\Omega}$. This leads to the following weak formulation:

Find $(p, \mathbf{v}) \in \mathcal{Q} \times \boldsymbol{\mathcal{S}}$:

$$\int_{\bar{\Omega}} q\ (\boldsymbol{\nabla} \cdot \mathbf{v})\ d\bar{\Omega} = 0 \tag{53}$$

$$\int_{\bar{\Omega}} -\boldsymbol{\omega} \cdot (\boldsymbol{\nabla} \cdot \boldsymbol{\sigma})\ d\bar{\Omega} = 0 \tag{54}$$

For all $(q, \boldsymbol{\omega}) \in \mathcal{Q} \times \boldsymbol{\mathcal{V}}$

The velocity field $\mathbf{v} \in \boldsymbol{\mathcal{S}}$ and the pressure field $p \in \mathcal{Q}$ are the unknowns in this equation. By integrating by parts the term involving $\boldsymbol{\sigma}$, using the divergence theorem and using the fact that $\boldsymbol{\omega} = 0$ on Γ_D we can rewrite the equations as:

Find $(p, \mathbf{v}) \in \mathcal{Q} \times \boldsymbol{\mathcal{S}}$:

$$\int_{\bar{\Omega}} q\ (\boldsymbol{\nabla} \cdot \mathbf{v})\ d\bar{\Omega} = 0 \tag{55}$$

$$\int_{\bar{\Omega}} \boldsymbol{\nabla}\boldsymbol{\omega} : \boldsymbol{\sigma}\ d\bar{\Omega} = 0 \tag{56}$$

For all $(q, \boldsymbol{\omega}) \in \mathcal{Q} \times \boldsymbol{\mathcal{V}}$

Now we can add up both equations, to obtain the weak form:

$$0 = \int_{\bar{\Omega}} q\ (\boldsymbol{\nabla} \cdot \mathbf{v})\ d\bar{\Omega} + \int_{\bar{\Omega}} \boldsymbol{\nabla}\boldsymbol{\omega} : \boldsymbol{\sigma}\ d\bar{\Omega} \tag{57}$$

The weighting functions $\boldsymbol{\omega}$ and q are arbitrary, thus the solution of the variational problem Equation (57) verifies the strong form (Equation 39, 40, 43, 44) of the steady Stokes problem.
For the discretization of the Stokes equation linear finite elements are used. Since these in itself would violate the Ladyzhenskaya-Babuška-Brezzi condition (LBB) compatibility condition [44], additional stabilization is required in order to obatin a well-posed problem that can actually be solved. The stabilized finite element formulation reads as follows:

$$0 = \int_{\bar{\Omega}} q\ (\boldsymbol{\nabla} \cdot \mathbf{v})\ d\bar{\Omega} + \int_{\bar{\Omega}} \boldsymbol{\nabla}\boldsymbol{\omega} : \boldsymbol{\sigma}\ d\bar{\Omega} + \sum_{e=1}^{n_{el}} \int_{\bar{\Omega}^e} \tau^e_{mom} \frac{1}{\rho} \boldsymbol{\nabla} q \cdot \boldsymbol{\nabla} p\ d\bar{\Omega}^e + \tag{58}$$

$$\sum_{e=1}^{n_{el}} \int_{\bar{\Omega}^e} \tau^e_{cont} (\boldsymbol{\nabla} \cdot \boldsymbol{\omega}) \rho (\boldsymbol{\nabla} \cdot \mathbf{v})\ d\bar{\Omega}^e \tag{59}$$

Here, τ^e_{mom} and τ^e_{cont} are stabilization parameters, which depend on a geometric metric of the current element under consideration. This formulation of the weak form is impemented in the flow solver XNS. For more details, we refer to [45, Chapter 4].

3.3. Interaction RL-FEM

In terms of RL, the model of our analysis is programmed in the finite element flow simulation program XNS and the RL agent has to interact with it. The FEM-program solves the specific Navier-Stokes equations numerically. The interaction takes place by reading the output of the XNS-solver, in this case the mass flows, and feeding these values to our agent. After the algorithm changes some values in the input files of the solver, XNS is used again with the altered values. The algorithms used in `Stable Baselines3` (SB3) are implemented in PyTorch [46].
Figure 21 depicts this interaction loop. First, the model is created with all boundary conditions etc. in a step called Pre-Processing. The cycle starts by a mesh generation. Next, our CFD program XNS performs a flow simulation, resulting in the mass flows as an output. From this, the RL agent calculates a quantity of interest, which is in our case a mass flow ratio, and evaluates if it is within a specified tolerance and if it got closer or further away from the desired mass flow ratio. This is then used to calculate the reward r_t. If the quantity of interest is not within a specific limit, then either the coordinates of the corner points of the geometry are directly changed, and the mesh is newly created, which starts the cycle again. Or, the coordinates of the control points are changed, with a FFD, the mesh is altered and calculated again. If the quantity of interest is within a threshold, and it was the last episode, the algorithm stops. If it was not the last episode, the cycle starts again with the mesh generator.
The general idea for this optimization cycle is very similar to the shape optimization cycle introduced in Figure 2. The difference lies in the optimization algorithm. [15] uses a gradient-based optimization algorithm as well as a function approximation, while in this thesis a RL algorithm is used.
To use SB3 for this task, the interface needs to follow the OpenAI gym interface (inherit from Open AI gym class) (Listing 2) [46]. OpenAI gym is a toolkit for reinforcement learning. It was built to give a benchmark for the various different algorithms, including custom ones from individuals. The environments are tasks with a common interface [47] i.e. in this thesis the task is to get a specific mass flow ratio, it could also be used to train an agent to play Pong. SB3 was chosen because of the ease of implementation and the various different algorithms provided.

3.4. Action and Observation Space

Algorithms can use continuous or discrete action and observation spaces. These action and observation spaces are represented by the `Box` and `Discrete` data structures provided by Gym. They describe the legitimate values for observations and actions for the environments. In a continuous action space, the agent gets upper and lower bound values for all `j` dimensions and can choose an action in between, i.e.: The action-space

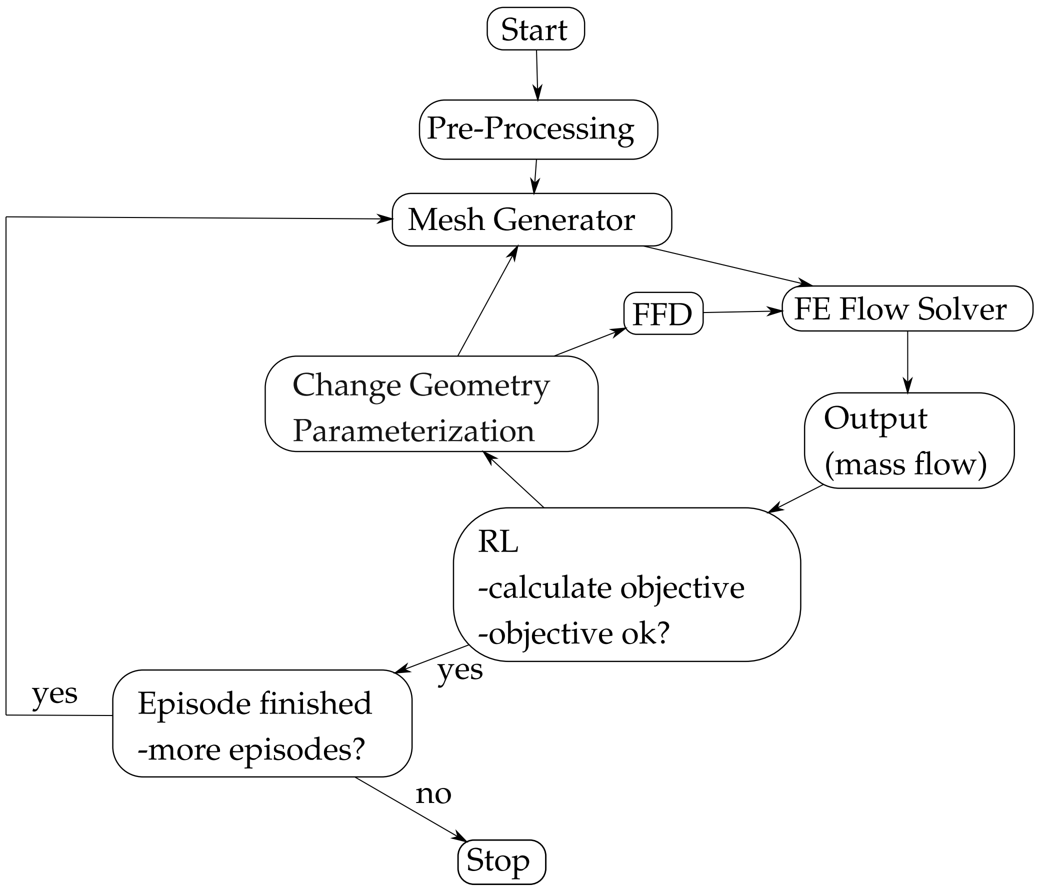


Figure 21: Interaction loop between the agent and the model of the analysis.

is an j-dimensional box that contains every point in the action space. A discrete action space defines the k actions that are possible and in each time step one of these discrete actions is selected [46].
The action space defines the characteristics of the possible actions of the agent. In this thesis, discrete actions were chosen. This gives the agent a finite number of actions to choose from and was empirically found by the author of this thesis that the used algorithms converge faster for discrete actions, in the experiments performed. The observation space for the tasks in this thesis is continuous, so all possible outcomes of the calculations are covered. For all tests, an increment δ is defined, and the agent can choose to alter a value by $\pm\delta$. Changing that value either changes directly the coordinates or changes the values of the control points for the FFD.

3.5. Rewards

For each of the experiments, a specific reward function had to be shaped which will be explained in detail in the corresponding subsections of Section 4. However, they all have similar characteristics. The rewards are designed in a way that the agent gets either a positive or negative reward after each step. Although it is possible to just give a reward after an episode of experiments is finished, much more training is needed as a lot of steps have to be performed (in the correct order). In the literature, this is referred to as the sparse rewards problematic [48, 49].

4. Results

In this chapter the results of the thesis are shown. First, an overview of the experiments performed is shown. On the example of the Matyas function, which is a test function for optimization, the basic mode of operation is discussed, we take a look at the reward shaping of this function and compare two different kinds of reward functions. The two different algorithms A2C and PPO are then evaluated for their performance, namely average time steps and cumulative reward per episode. In a hyperparameter study it is then analyzed how the algorithms can be optimized, apart from changing the reward function.
The emphasis of this thesis is shape optimization, which is performed on the course of three different experiments with varying DOF. First, a reward function is defined for this part of the thesis, then the three different experiments are evaluated for their performance. The focus on this part is to examine if the agents learn and which of them can achieve the goals of these experiments faster and more consistently. At the end of this chapter, the tests are evaluated for their runtime and some example shapes produced by these experiments are depicted and compared.

4.1. Overview

The aim of this thesis is to evaluate if shape optimization can be achieved by means of RL. The performance, namely time steps per episode and cumulative reward, of two different RL algorithms (A2C and PPO) were analyzed and compared in four experiments. In Table 6 an overview of the four experiments is shown. In the first experiment (Test I) the agent tries to find the minimum of a mathematical function, in this case the Matyas function, with as few steps as possible. The following three experiments, modify a simple geometry, which can be seen as an abstraction of the flow channel inside a profile extruder. The experiments Test II and Test III directly change the geometry by changing the coordinates of the corner points defining the geometry. Test IV changes the shape with a technique called FFD. The DOF for the tests increase. For each DOF one discrete action is needed. This implies that the test cases get more difficult for the agent as it has to evaluate more actions.

Matyas (Test I):
The Matyas function is used as a test function. Test functions are important to validate new optimization algorithms and to compare their performance. This experiment has nothing to do with shape optimization, but because of its short runtime and simplicity it helped to examine if the algorithms are working and if they can be improved by changing hyperparameters.
The advantage of this test function is that the minimum can be calculated analytically. The gradient of this test function gets relatively small close to the minimum, which

Type	Name	DV/DOF	Parameterization
Test function for optimization	Test I	2 DV/4 DOF	Coordinates of x and y can be altered by $\pm\delta$
Shape optimization	Test II	8 DV/16 DOF	Coordinates of the key points of the geometry can be moved vertically
	Test III	8 DV/32 DOF	Coordinates of the key points of the geometry can be moved vertically and horizontally
	Test IV	9 DV/36 DOF	Coordinates of the control points can be moved vertically and horizontally

Table 6: Overview of the examples for optimization. The key points of the geometry (A-H) can be seen in Figure 23.

increases the difficulty of finding said minimum. In Figure 22 this function is depicted. The dark blue color indicates parts of the function with small z values. The function is shown for x and y values of $\pm$ 3 for a better overview. For the function itself, it is not necessary to limit the values, however to speed up training the values of x and y were limited with $\pm$ 5 in the test.

Shape optimization (Test II - IV):
For the Tests II - IV the geometry depicted in Figure 23 had to be optimized.

The mass flow $\dot{m}_1$ is the input, the mass flows $\dot{m}_2$ and $\dot{m}_3$ are the outputs. The goal is to get a specific ratio between $\dot{m}_2$ and $\dot{m}_3$.

$$\dot{m}_{23} = \frac{\dot{m}_2}{\dot{m}_3} \tag{60}$$

In the Tests II and III, the coordinates of the corner points of the geometry $A - H$ could be modified by the agent. After changing the coordinates, the geometry is meshed and XNS runs the FEM simulation. In order to find out whether it makes a difference for the agent how many DOF are given, the experiment is split up into two separate experiments. In the first one (Test II) the coordinates of the points $A - H$ can be modified only vertically, i.e. along the y-axis. This gives us two DOF for each point, resulting in a total of 16 DOF for the whole experiment. In order to prevent the mesh from getting tangled, the amount the coordinates can change is limited, indicated by the red bars in Figure 23. The bars do not show the exact limits and are purely descriptive.
In Test III the coordinates of the points $A - H$ can be changed horizontally and vertically, as indicated by the green boxes. This leads to four DOF per point, resulting in

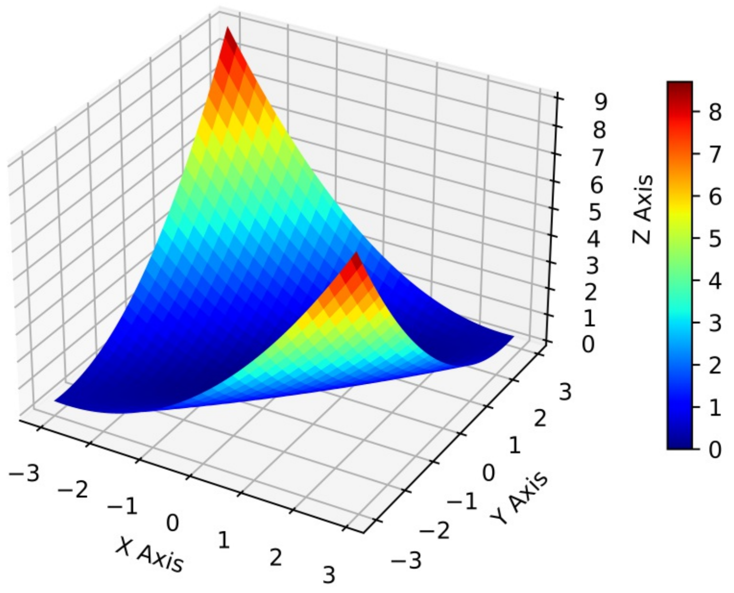


Figure 22: Matyas function.

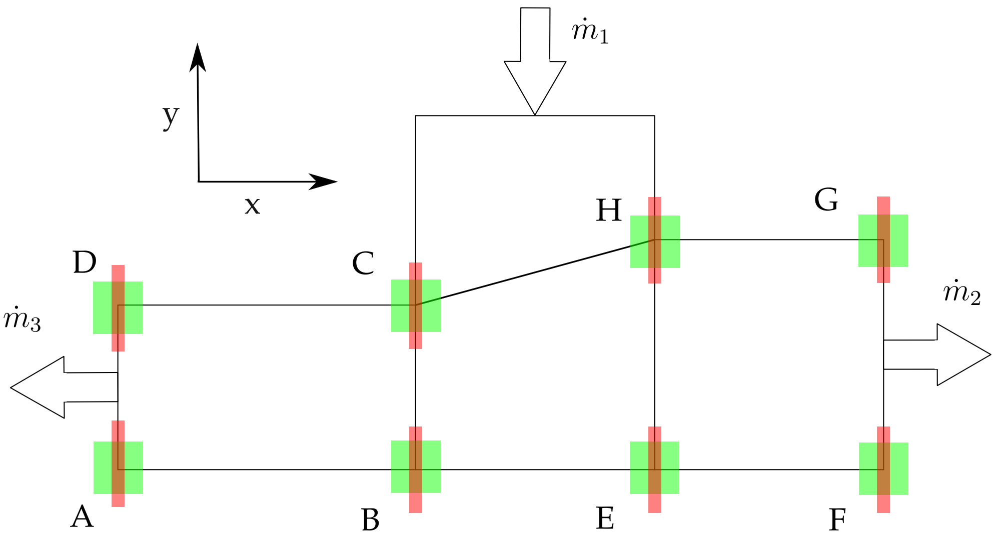


Figure 23: Geometry to be shape optimized.

a total of 32 DOF. In Section 4 it is analyzed if this affects the average time steps per episode and the cumulative reward per episode.
For the last experiment (Test IV) the geometry is altered by a FFD which is explained in detail in Section 3.1.

4.2. Matyas (Test I)

In this section the first experiment is explained in detail. First, we take a look on the basic mode of operation in terms of the code used. In the next subsection, two different reward functions are introduced, namely one with dynamic rewards and a function with constants. In Subsection 4.2.3 a sample episode is depicted to show the difference between a trained and untrained agent. Then the two different reward functions are compared to each other as well as the two agents among themselves. In Subsection 4.2.4, the learning rate and the clipping range is adjusted to evaluate if the performance can be improved by altering these values.

4.2.1. Basic mode of operation

Listing 1 shows the basic mode of operation for the Matyas function (Test I). The other examples are based on it and differ slightly. This is an overview of the code used, and stripped down to make it easier to comprehend.

```
LEFT_x=0
RIGHT_x=1
LEFT_y=2
RIGHT_y=3

def __init__(self, delta=0.1,max=5,min=-5,step_max=200):
  self.delta=delta
  self.max=max
  self.min=min
  n_actions=4
  self.action_space = spaces.Discrete(n_actions)
  self.observation_space = spaces.Box(low=self.min, high=self.max,
                                      shape=(2,), dtype=np.float32)
  self.step_max=step_max
def step(self, action):
  old_value_z=f(x,y) #Can be any function, in this example the Matyas
    function is used
  if action == self.LEFT_x:
    self.x_value -= self.delta
  elif action == self.RIGHT_x:
    self.x_value += self.delta
  elif action == self.LEFT_y:
    self.y_value -= self.delta
  elif action == self.RIGHT_y:
    self.y_value += self.delta
  self.x_value= np.clip(self.x_value, self.min, self.max)
  self.y_value= np.clip(self.y_value, self.min, self.max)
```

```
27    self.z_value=f(x,y) #Can be any function, in this example the
       Matyas function is used
28    done=False
29    self.step_count+=1
30    reward=* #the reward function is explained in the following section
31    logfile(self.x_value, self.y_value, self.z_value, reward, action)
32    return np.array([self.x_value,self.y_value]).astype(np.float32),
       reward, done, info
33  def reset(self):
34    self.step_count=0
35    self.y_value=randint(self.min*10,self.max*10)/10
36    self.x_value=randint(self.min*10,self.max*10)/10
37    self.z_value=f(x,y)
38    return np.array([self.x_value,self.y_value]).astype(np.float32)
```

Listing 1: Basic mode of operation, Matyas example.

Lines `1` to `4` define variables for the discrete actions that can be performed by the agent. The observation space is continuous and limited with the `min` and `max` values (line `12`). The maximum amount of steps `self.step_max` $= steps_{max}$ per episode are limited to reduce the calculation time in case a poor policy does not find a solution (line `14`). In the function `step` first the value of `self.old_value_z` $= z^{old}$ is calculated (line `16`). For the Matyas function, $f(x, y)$ is defined in Equation (61) but can be adapted to different functions as well. Then the agent can perform one of the following actions: Lower or increase the `self.x_value` $= x$ values or respectively the `self.y_value` $= y$ values (lines `17-24`). The function `np.clip()` limits the values of x and y (lines `25,26`). The value of `self.z_value` $= z$ (line `27`) is calculated to compare it to the value of z^{old} and choose the reward (line `30`) according to Algorithm 1. The value of `done` is set to `False` and is changed by the `reward`-function to `True` as soon as the epsiode is complete (line `28`). The shape of the return array is defined in line `13` with the operator of `shape=(2,)` and is 2 in this case as the two values x and y are returned.

The function `reset` is called after each episode. First the step counter is reset, then the values of x and y are reset to a random number within the boundaries to obtain a random starting position for z.

4.2.2. Reward shaping

One of the most challenging tasks in RL is finding a reward function for a specific task that makes the agent perform in the intended way of the programmer. Often the agent finds a way to exploit the reward function in order to obtain a high reward without finishing the task. To understand how to shape said reward function, a simple test function, in this case the Matyas function, is used. The Matyas function is defined as:

$$z = f(x, y) = 0.26 \cdot (x^2 + y^2) - 0.48 \cdot x \cdot y \tag{61}$$

The variables x and y can be modified by the agent. The goal is to find the minimum of this function.
In Figure 24 a sample episode is shown to illustrate how the agent works. The agent

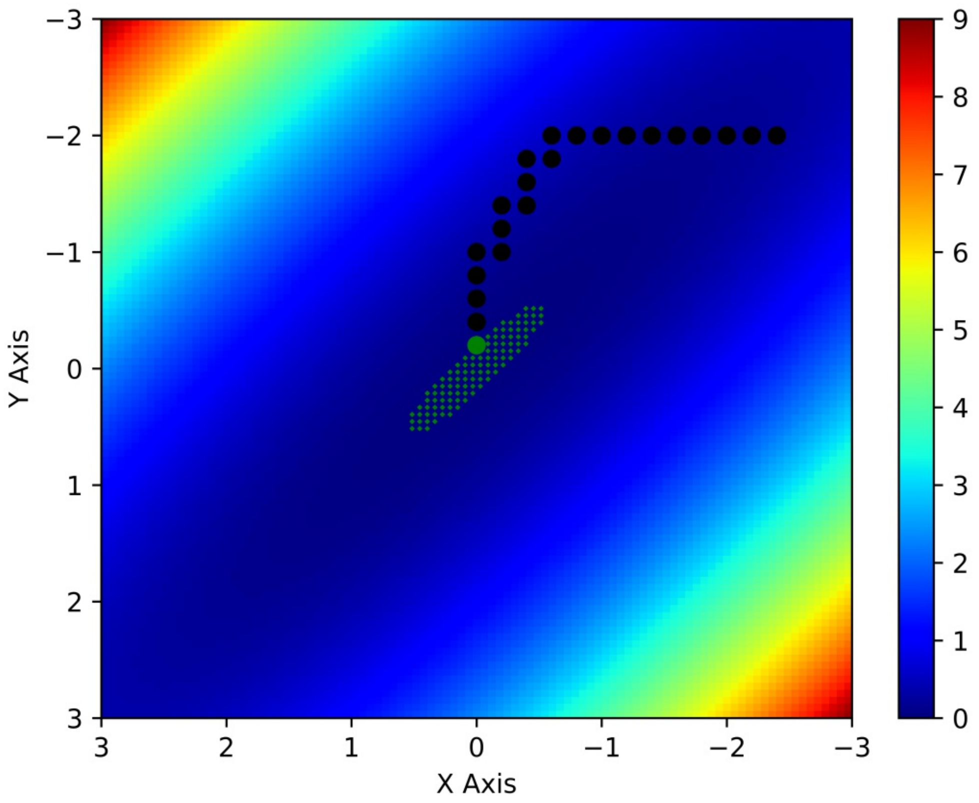


Figure 24: Sample episode of the Matyas function, the black dots represent the z values which are manipulated by the agent by altering the x and y values of the function. The distance between the dots is defined by $\delta = 0.2$. The small green dots represent the goal value of z.

can modify each step either the x or y values by $\pm\delta$. The black dots are the z values calculated, the distance between the points is defined by δ. If the algorithm achieved a z value below a specified threshold (the green points in the picture) the episode ends. Two reward functions were shaped and evaluated for their performance. The first reward function is depicted in Algorithm 1 and used as a baseline, as it is relatively simple, and the rewards are constants. With this definition, the agent gets a positive reward r_t if the z value gets lower than z^{old}, a negative reward if it gets higher and a

Algorithm 1 Reward for the Matyas function (constant reward).

```
1: if $z - z^{old} < 0$ then
2:     $r_t \leftarrow 0.01$
3: else if $z - z^{old} > 0$ then
4:     $r_t \leftarrow -0.5$
5: if $|z| < 0.01$ then
6:     $r_t \leftarrow 5$
7:     $done \leftarrow True$
8: if $steps \geq steps_{max}$ then
9:     $done \leftarrow True$
```

big positive reward if it reaches the goal. Variables with the superscript ^{old} refer to the value of the previous step. An termination condition is defined in case the agents gets stuck, limiting the time wasted to the maximum number of steps per episode $steps_{max}$. The positive and negative reward are asymmetric, making it impossible for the agent to switch between higher and lower values of z to collect more reward.
The second reward function Algorithm 2 is a bit more sophisticated and uses variable rewards instead of constants.

Algorithm 2 Reward for the Matyas function (variable reward).

```
1: $delta^{old}_{matyas} = |z^{old} - z^{goal}|$
2: $delta_{matyas} = |z - z^{goal}|$
3: if $z - z^{old} < 0$ then
4:     $r_t \leftarrow \frac{|z-z^{old}|}{delta^{old}_{matyas}} + 0.2$
5: else if $z - z^{old} > 0$ then
6:     $r_t \leftarrow -\frac{|z-z^{old}|}{delta_{matyas}} - 0.5$
7: if $|z| < 0.01$ then
8:     $r_t \leftarrow 20$
9:     $done \leftarrow True$
10: if $steps \geq steps_{max}$ then
11:     $r_t \leftarrow -20$
12:     $done \leftarrow True$
```

In this algorithm a $delta_{matyas}$ and $delta^{old}_{matyas}$ are defined as the difference between the current value of z, z^{old} and the goal value of $z^{goal} = 0$. These variables are introduced to highlight the resemblance between this reward function and the one used in Algorithm 3 for the shape optimization. This is not related to the δ used as an increment in Listing 1.

As with the reward function above, the superscript ^{old} refers to variables of the previous step. If the agent makes progress, it gets a positive reward:

$$r_t = \frac{|z - z^{old}|}{delta^{old}_{matyas}} + 0.2 \tag{62}$$

The absolute value is used, so the reward cannot become negative. In the nominator the term $z - z^{old}$ increases, the greater the progress step is, rewarding the agent if it makes big steps towards the goal. The expression in the denominator $delta^{old}_{matyas}$ gets bigger, the closer the agent is to the goal. $delta^{old}_{matyas}$ was found empirically to work better than $delta_{matyas}$.
If the agent does not make progress, it gets a negative reward:

$$r_t = -\frac{|z - z^{old}|}{delta_{matyas}} - 0.5 \tag{63}$$

$$\tag{64}$$

This reward is also dynamic and gets a higher negative value, if the agent makes a bigger mistake or makes a mistake close to the goal. The constants $+0.2$ and -0.5 are added, so the reward does not get too small if the value of z is far from the goal value of $z^{goal} = 0$. If the agent reaches a value of z within a threshold of 0.01 it is rewarded with $r_t = 20$, and the episode is finished. If too many steps are needed, the episode is finished as well, but a negative reward of $r_t = -20$ is given.

4.2.3. Evaluation

In Section 4.1 Test I was introduced and a plot of the Matyas function in 3D (Figure 22) as well as in 2D (Figure 24) shown. The basic mode of operation regarding the code used was introduced in Subsection 4.2.1, now we take a look at how a trained and untrained agent act in general and what influence the choice of the reward function has. In Subsection 4.2.4, hyperparameters are altered and their influence on the performance is evaluated. The effects of the actions $0 - 3$ are shown in Table 7. Depending on the action, either the value of x or y is altered by $\pm\delta$. In order to show how the

Action	Effect
0	$x - \delta$
1	$x + \delta$
2	$y - \delta$
3	$y + \delta$

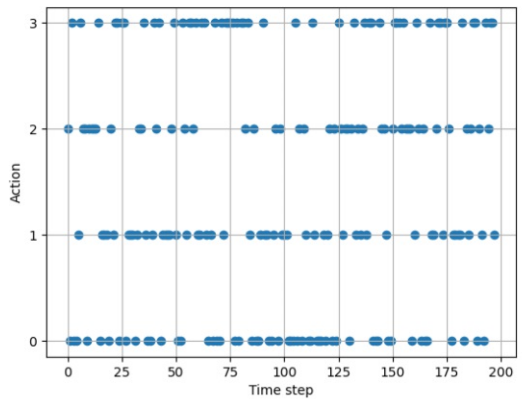
Table 7: Actions and their effect in the Matyas algorithm.

agent works, Figure 25 shows the actions, coordinates and rewards of an untrained and trained agent (on the basis of the PPO algorithm) with the baseline reward function Algorithm 1.

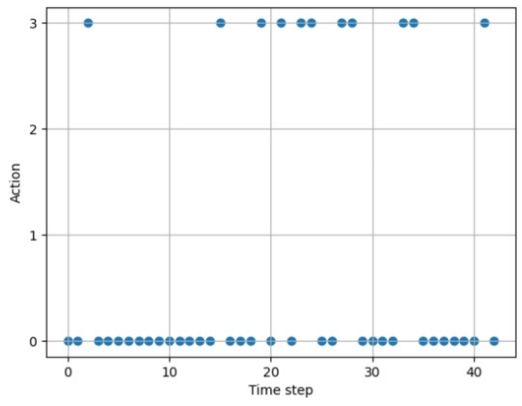
Figure 25a shows the actions of an untrained agent. On the y-axis the actions $0-3$ are depicted and on the x-axis the time steps. The untrained agent tries out all four possible actions in order to find the actions with the highest reward. In the beginning the actions are sampled relatively random as the agent has no prior experience on the amount of reward r_t each of the actions gives (depending on the state t). Figure 25b shows the actions of an agent that has been trained for 500 episodes. The most rewarding actions depend on the starting values of x and y, as well as the goal value of z. In this test case, the minimum is searched which lies at $z = f(x, y) = f(0, 0) = 0$. For the starting values of this episode the actions 3 and 0 are the most rewarding, different actions are not taken. Note that in this example the untrained agent never reaches the goal of $z < 0.01$ and ends the episode after the maximum amount of allowed steps $steps_{max} = 200$ is reached.
Figure 25c shows the effect of these random actions of the untrained agent on the variables x, y and z. On the y-axis the value of the coordinates are shown and on the x-axis the time steps. The goal is to find the minimum of z. The untrained agent is not successful in doing so, while the trained agent Figure 25d reaches this goal relatively fast after 42 time steps.
Figure 25e and 25f show the rewards for the untrained and trained agent. Getting closer to the goal of $z = 0$ gives a reward of $r_t = 0.01$, while getting further away gives a reward of $r_t = -0.5$. If the goal is reached within a limit, a reward of $r_t = 5$ is given. The untrained agent jumps between a positive and negative reward randomly, as it does not consistently get closer to the goal. The trained agent gets a reward of $r_t = 0.01$ at each step till it reaches the goal at step 42 and gets the reward of $r_t = 5$ at the end of the episode. The logarithmic scale shows the big difference in the reward for achieving the goal compared to the small rewards for making a step closer to the goal. This big difference is used, so the algorithm tries to achieve the goal as fast as possible, instead of making just a little progress each step and collecting more reward in the long run. This difference in reward shows also in the accumulated (undiscounted) return G_t for this episode. The total return for the untrained agent is $G_t^{untrained} = -46.5$ while the trained agent has a return of $G_t^{trained} = 5.43$.

To evaluate which of the two algorithms (PPO,A2C) performs better, they are compared in Figure 26. For the first evaluation, both algorithms are used with the standard hyperparameters provided by SB3 and the constant reward function Algorithm 1.

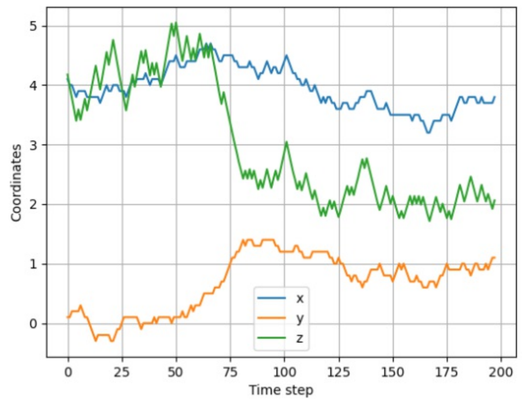
In blue the A2C algorithm is shown and in orange the PPO agent. The acronym `lr` stands for the learning rate and `cr` for the clipping range. The comparison is done by comparing the average length per episode and the cumulative reward per episode. The goal is to have few steps per episode and a high cumulative reward. If the reward func-



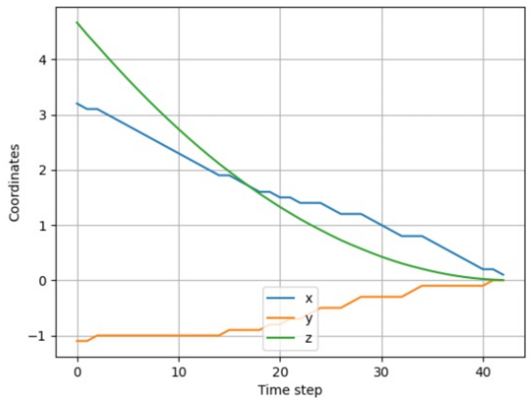
(a) Actions of an untrained agent. The discrete actions correspond to Table 7.



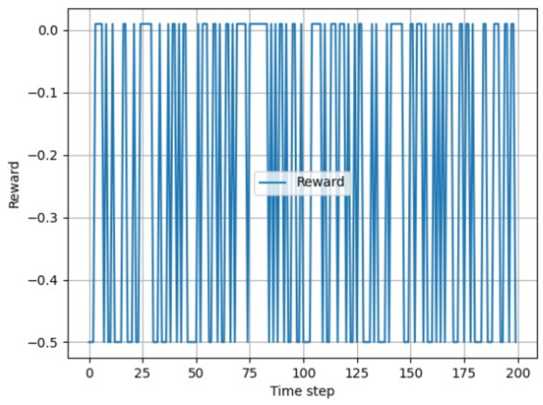
(b) Actions of a trained agent. The discrete actions correspond to Table 7.



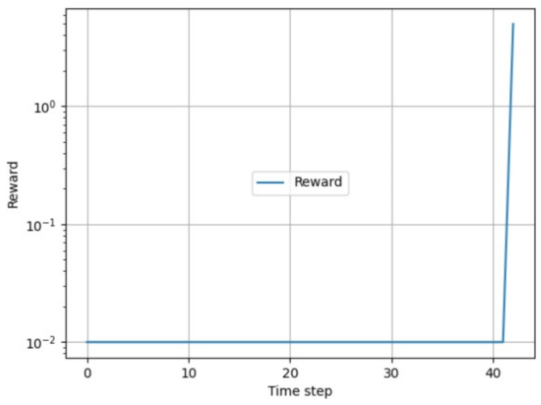
(c) Coordinates x, y, z of an untrained agent.



(d) Coordinates x, y, z of a trained agent.

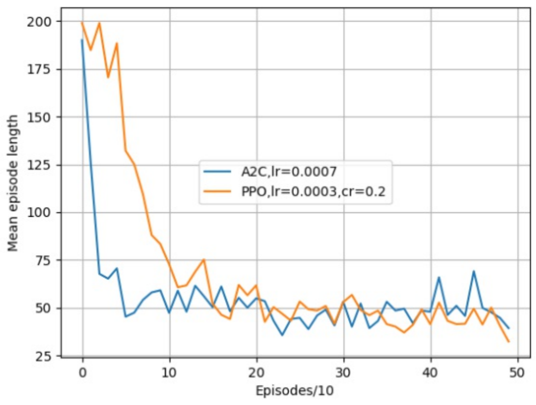


(e) Reward of an untrained agent.

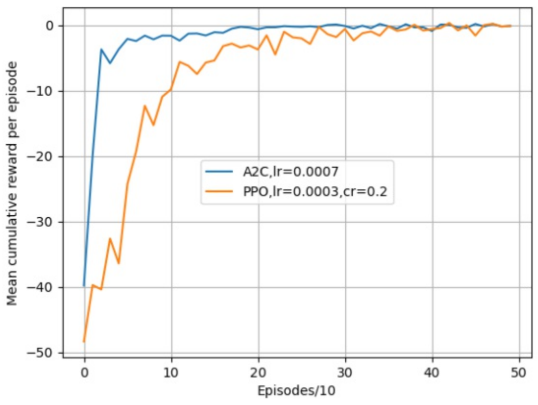


(f) Reward of a trained agent (logarithmic scale).

Figure 25: Comparison of an untrained and trained agent by the example of the Matyas function. Both algorithms are trained with the constant reward function Algorithm 1.



(a) Episode length averaged over 10 episodes.



(b) Cumulative reward per episode averaged over 10 episodes.

Figure 26: Comparison between the two algorithms applied to the example of the Matyas function.

tion is chosen appropriately and the agent learns, the cumulative reward per episode gets higher the lower the average time steps per episode are. This is true for this example. Both algorithms improve their behaviour during these first 500 episodes. The learning curve of the A2C algorithm is steeper, which means it learns a good policy faster, or in other words: the A2C agent is more sample efficient. An explanation could be that the learning rate is higher for the A2C algorithm, which allows the weights of the neural network to change more per step than with the PPO algorithm. Furthermore, the PPO agent uses a clipping range which limits how much the policy can change each step.
Another comparison between these two algorithms is depicted in Table 8:

At first a comparison is done by comparing how often the goal of $z < 0.01$ is reached. The table is split up into the episodes $0-100, 100-200, ..., 400-500$ to show the learning progress. The A2C agent stays fairly consistent with reaching the goal between 58-64% of the time, while the PPO agent can improve from 39% to around 70%. It is worth noting that this seems not to be very consistent, comparing the second to last and last time range. As a second key stat, the average cumulative reward per episode is looked at. The average reward increases for both algorithms, it seems that the A2C agent gets higher average cumulative rewards from episode 300 on. This is a bit surprising because the average time steps per episode are a bit higher than for the PPO agent. An explanation might be that because the A2C agent needs more time steps per episode, it can accumulate a higher average reward per episode. The reward per time step is higher for the PPO agent and increasing each time period. The absolute values of the reward depend on the reward function and can be compared only to algorithms with exactly the same reward function.

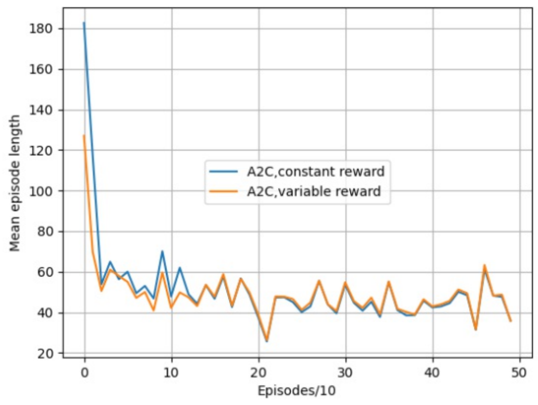
		Episodes				
Key stat	Algorithm	0-100	100-200	200-300	300-400	400-500
Goal reached	A2C	64	64	60	58	63
	PPO	39	66	71	60	70
Avg. cumulative reward -per episode	A2C	-3.9	3.8	4.8	4.8	4.8
	PPO	-25.5	-0.28	2.9	4.1	4.6
Avg. reward -per time step ($\cdot 10^{-3}$)	A2C	-49.2	69.6	104.3	100	92.8
	PPO	-171.1	-4.6	58.2	86.9	103.6
Average time steps	A2C	79.3	54.6	46.0	48.0	51.7
	PPO	149.0	61.0	49.8	47.2	44.4

Table 8: Comparison of some performance measures between the A2C and PPO algorithm applied to the example of the Matyas test function and a constant reward function.

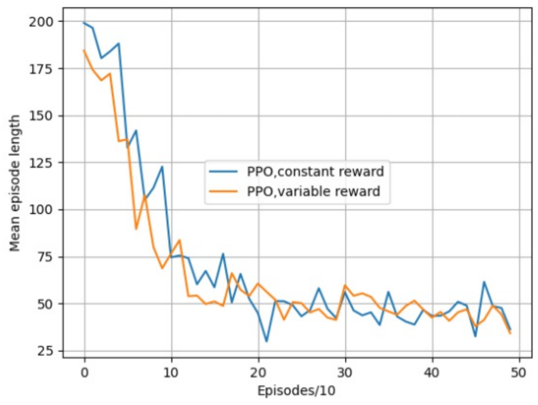
Next, we want to evaluate if changing the reward function to dynamic rewards, instead of constant, makes a difference. This is shown in Figure 27. As the rewards are altered, the time steps needed to complete the 500 episodes are compared. It is shown that both reward functions work and perform similarly. To examine if a difference is noticeable, some key measures are calculated and compared in Table 9. The total number of time steps needed for the variable reward function for the A2C agent is approximately 4.7% less and for the PPO 7.1% less respectively. When performing the experiment, the same seed for the pseudo random generator in the algorithm was used, which makes the experiments comparable. In the final performance, namely the average time steps needed per episode between the episodes $400-500$ both reward function perform similarly with a 1.7% performance loss for the A2C agent and a 6.7% improvement for the PPO agent. In all the related works that use a PPO agent [36, 38, 39] a dynamic reward function is used aswell. This led to the decision to use a dynamic reward function for the actual shape optimization in Section 4.3.

4.2.4. Hyperparameter study

In this subsection the learning rate is adjusted for the A2C agent, for the PPO agent both the learning rate and the clipping range are altered and the influence on the performance is studied. We start by adjusting the learning rate for the A2C agent. Figure 28a shows the average episode length and Figure 28b the average cumulative reward per episode, for a total of 500 episodes. The learning rate is incremented by the



(a) Episode length averaged over 10 episodes, A2C agent.



(b) Episode length averaged over 10 episodes, PPO agent.

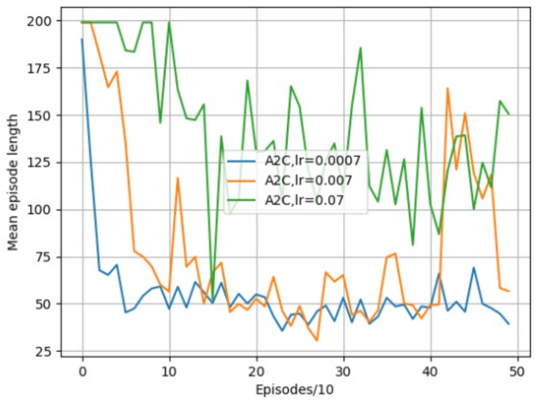
Figure 27: Comparison between a continuous reward function (Algorithm 1) and a variable reward function (Algorithm 2) on the example of the Matyas test function.

Agent	Reward	Total time steps	Average time steps needed for episodes 400-500
A2C	Constant	26315	46.26
	Variable	25064	47.05
PPO	Constant	36446	46.89
	Variable	33854	43.74

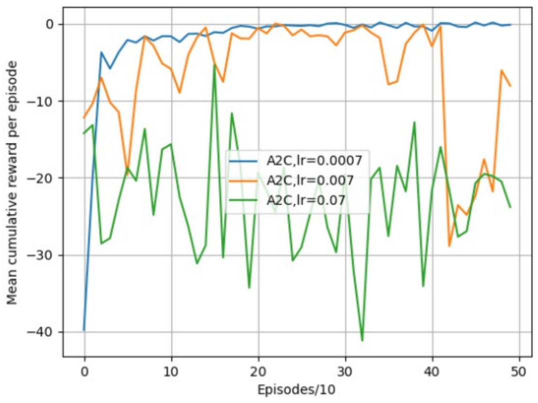
Table 9: Comparison of some key figures for a constant and variable reward function on the example of the Matyas function.

power of 10. We can see that increasing the learning rate from 0.0007 to 0.007 is enough to make the training unstable. The mean episode length as well as the average cumulative reward per episode fluctuate still after 400 episodes. Increasing the learning rate further, increases the episode length and decreases the average cumulative reward.
The same experiment was performed on the PPO agent, additionally we adjusted the clipping range. Figure 29a and 29b show that increasing the learning rate by a power of 10 still leads to a stable training, while further increasing it causes divergence. The progress in the first few episodes is a bit faster with the learning rate of $lr = 0.003$ instead of $lr = 0.0003$, but after 100 episodes there are still some peaks in the graph which indicate that a few episodes need a lot more time steps than others. With the learning rate of $lr = 0.03$ the agent learns an inappropriate policy which causes the agent to need the maximum amount of steps for a lot of episodes. This is shown in the average length as well as in the mean cumulative reward. Increasing the clipping range (Figure 29c, 29d) leads to a faster convergence towards the optimal policy. The

clipping range limits how much the policy can change each step, and in this simple example it is good for the algorithm to change the policy quickly. In a more complex environment like the following examples, this might not be true anymore, and the clipping range should not be increased. Based on the findings of this simple test case, the standard hyperparameters of SB3 are used. The PPO algorithm has a learning rate of $lr = 0.0003$ and a clipping range of $cr = 0.2$, the A2C algorithm has a learning rate of $lr = 0.0007$.

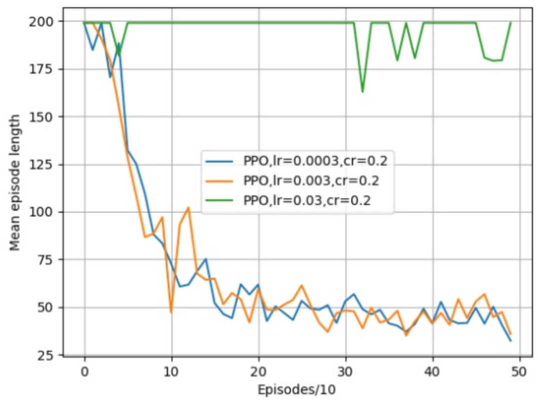


(a) Episode length averaged over 10 episodes.

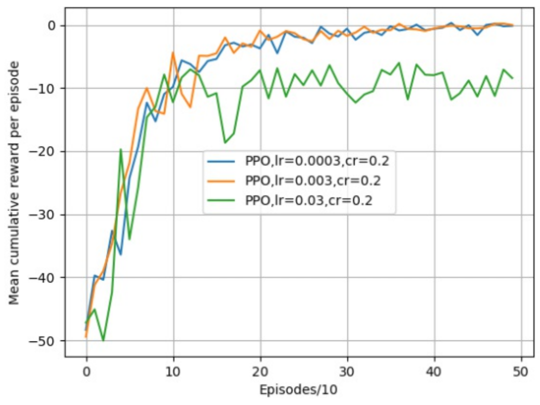


(b) Cumulative reward per episode averaged over 10 episodes.

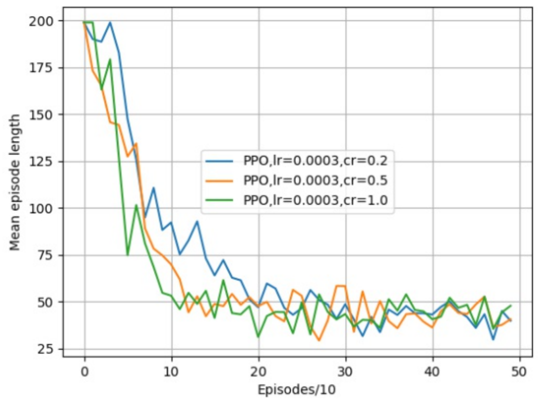
Figure 28: Hyperparameter study: modified learning rate of the A2C agent applied to the example of the Matyas function.



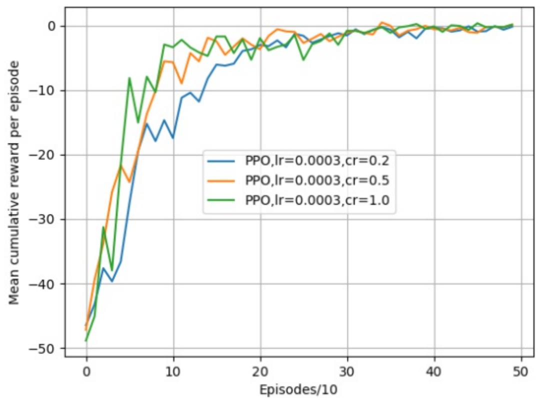
(a) Episode length averaged over 10 episodes.



(b) Cumulative reward per episode averaged over 10 episodes.



(c) Episode length averaged over 10 episodes.



(d) Cumulative reward per episode averaged over 10 episodes.

Figure 29: Hyperparameter study: modified learning and clipping range of the PPO agent applied to the example of the Matyas function.

4.3. Shape optimization

The aim of this section is to answer the core question of this thesis, namely if shape optimization can be achieved with a RL agent. First, a reward needs to be shaped for the three experiments. As it turned out, it was possible to use the same reward function for all three experiments. Then the experiments are evaluated for their time steps and cumulative reward per episode. Subsection 4.3.2 focuses on changing the geometry by directly changing the coordinates of the corner point of the geometry, while Subsection 4.3.3 changes the geometry by means of a FFD. In Subsection 4.5 some example shapes for the three different experiments are depicted and compared.

4.3.1. Reward shaping

The reward for the shape optimization (Tests II-IV) can be defined the same for all three environments. For the reasons mentioned above, a dynamic reward function is used, which is defined in Algorithm 3.

Algorithm 3 Dynamic reward function for shape optimization.

```
1:  $delta_{flow} = |\dot{m}_{23} - \dot{m}_{23}^{goal}|$
2:  $delta_{flow}^{old} = |\dot{m}_{23}^{old} - \dot{m}_{23}^{goal}|$
3:  if $error == 1$ then
4:      $r_t \leftarrow -5$
5:      restart episode
6:  if $delta_{flow} < delta_{flow}^{old}$ then
7:      $r_t \leftarrow \frac{|\dot{m}_{23} - \dot{m}_{23}^{old}|}{delta_{flow}^{old}} + 0.2$
8:  else if $delta_{flow} > delta_{flow}^{old}$ then
9:      $r_t \leftarrow -\frac{|\dot{m}_{23} - \dot{m}_{23}^{old}|}{delta_{flow}} - 0.5$
10: else
11:     $r_t \leftarrow -0.4$
12: if $|\dot{m}_{23} - \dot{m}_{23}^{goal}| < 0.02$ then
13:     $r_t \leftarrow 20$
14:     $done \leftarrow True$
15: if $steps \geq steps_{max}$ then
16:     $r_t \leftarrow -20$
17:     $done \leftarrow True$
```

This reward function is similar to the variable reward function Algorithm 2 for the Matyas function, except an abort condition is added. In some cases the calculation with XNS did not work, the reason was a tangled mesh which always lead to an error and an empty mass flow file. To resolve this issue, the variable $error$ is introduced. If the mass flow file is empty, the variable $error$ is set to 1 and the coordinates are reset

to a random point in the log file and a negative reward is given. With this method it is quite unlikely that the same error will occur within the next steps again and the calculation can continue. In this algorithm $delta_{flow}$ and $delta_{flow}^{old}$ are defined as the difference between the current mass flow ratio $\dot{m}_{23}$, $\dot{m}_{23}^{old}$ and the mass flow ratio it should achieve $\dot{m}_{23}^{goal}$. This is not related to the δ used as an increment in Listing 1.
As with the reward function mentioned earlier, the superscript ^{old} refers to variables from the previous time step. If the agent makes progress, it gets rewarded according to Equation (65).

$$r_t = \frac{|\dot{m}_{23} - \dot{m}_{23}^{old}|}{delta_{flow}^{old}} \tag{65}$$

$$delta_{flow}^{old} = |\dot{m}_{23}^{old} - \dot{m}_{23}^{goal}| \tag{66}$$

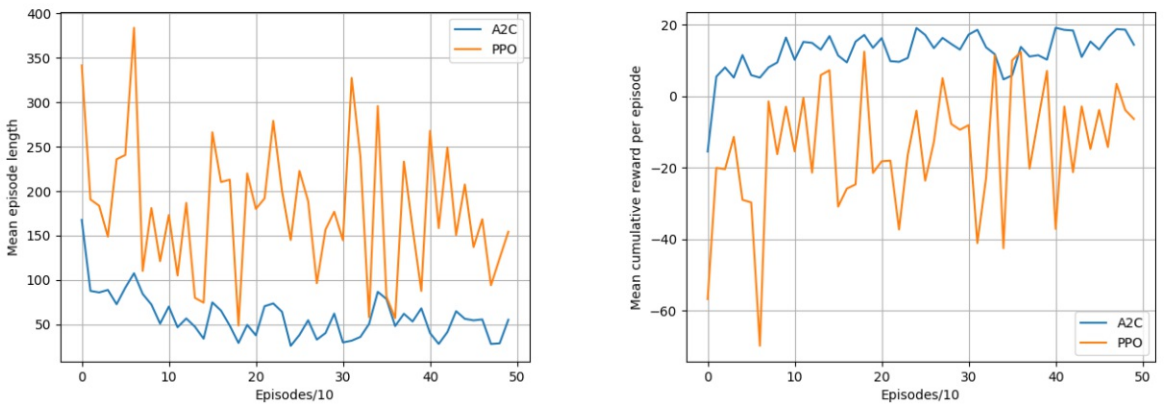
$$delta_{flow} = |\dot{m}_{23} - \dot{m}_{23}^{goal}| \tag{67}$$

The absolute value is used, so the reward cannot become negative. In the nominator the term $\dot{m}_{23} - \dot{m}_{23}^{old}$ increases, the greater the progress step is, rewarding the agent if it makes big steps towards the goal. The expression in the denominator $delta_{flow}^{old}$ gets smaller the closer the agent is to the goal, increasing the reward r_t. $delta_{flow}^{old}$ was found empirically to work better than $delta_{flow}$.
In case the mass flow ration does not change, the reward is set to $r_t = -0.4$. This happens if the coordinates of the points run into the limits of the clipping range. A clipping range is introduced to reduce the amount of times a tangled mesh is generated. In that case, the negative reward gets triggered as the reinforcement learning agent did not make progress.
If the agent reaches the specified mass flow ratio within a threshold of 0.02 it is rewarded with $r_t = 20$, and the episode is finished. If too many steps are needed, the episode is finished as well, but with a negative reward of $r_t = -20$.

4.3.2. Evaluation of Test II and III

For the direct change of the coordinates, two configurations are analyzed. First the coordinates can be adjusted only vertically which leads to 16 DOF (8 nodes that can be either increased or decreased). In the second example all nodes can be adjusted vertically and horizontally which leads to 32 DOFs. It is then analyzed if the DOF have an impact on the learning curve. The corner points of the geometry (A-H), which can be adjusted, are depicted in Figure 23.
Vertical coordinates only (Test II)
The results for Test II are depicted in Figure 30. Both algorithms could improve their performance over the course of 500 episodes. In this test the A2C algorithm showed a faster convergence towards a short episode length as well as a higher cumulative reward per episode.

In Table 10 some key stats for both algorithms are shown. The A2C agent reaches the goal between 98% and 100% for the periods evaluated. The PPO agent is slightly worse, but still reaches the goal between 95% and 97% of the time. For both algorithms the average cumulative reward per episode increases and the average time steps per episode decreases, which means the agents learn. On the example of the average time steps in the last period (episodes 400-500) we can see that the A2C agent needs about one third of the time steps per episode compared to the PPO algorithm. This has also influence on the total time needed for the 500 episodes, which will be discussed in Section 4.4.



(a) Mean episode length. (b) Mean cumulative reward per episode.

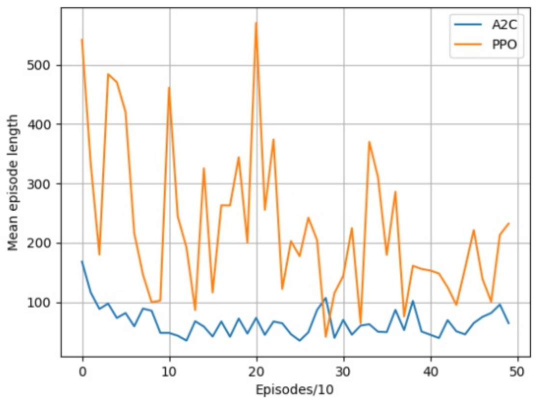
Figure 30: Test II: Comparison of the A2C and PPO agent with standard hyperparameters.

		Episodes				
Key stat	Algorithm	0-100	100-200	200-300	300-400	400-500
Goal reached	A2C	99	98	99	100	100
	PPO	97	97	95	97	97
Avg. cumulative reward -per episode	A2C	5.6	13.0	13.4	11.8	16.3
	PPO	-26.5	-11.3	-14.8	-10.3	-10.4
Average time steps	A2C	90.7	51.5	51.1	54.7	45.5
	PPO	213.2	157.4	181.7	171.2	171.4

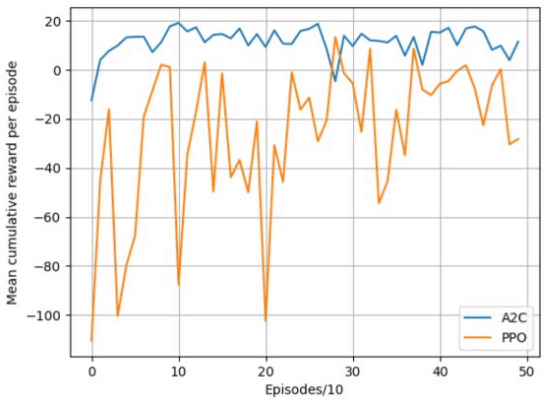
Table 10: Test II: Comparison of some performance measures of the A2C and PPO algorithm.

Vertical and horizontal coordinates (Test III)

As with Test II both algorithms could lower the average time steps needed for one episode and increase the cumulative reward per episode. The A2C agent outperforms the PPO agent in terms of time steps per episode and cumulative reward per episode, again by a great margin. This is even more apparent when comparing the key performance measures in Table 11. In the last 100 episodes, the A2C agent needs about 60% less time steps per episode. It should be noted that the average time steps as well as the cumulative reward fluctuates highly for the PPO agent, with some better episodes surpassing the A2C agent.



(a) Mean episode length.



(b) Mean cumulative reward per episode.

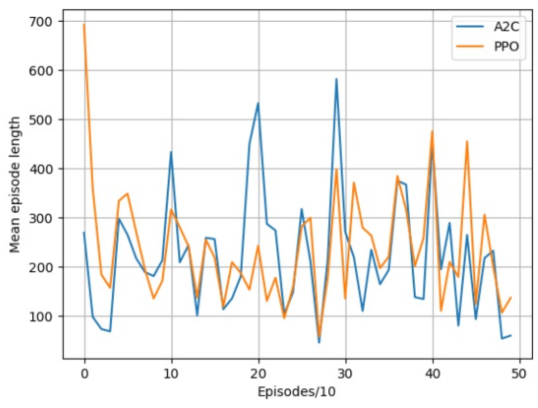
Figure 31: Test III: Comparison of the two algorithms.

		Episodes				
Key stat	Algorithm	0-100	100-200	200-300	300-400	400-500
Goal reached	A2C	98	100	100	100	100
	PPO	87	89	97	93	98
Avg. cumulative reward -per episode	A2C	7.7	14.7	11.5	11.1	12.5
	PPO	-44.5	-34.2	-24.4	-18.5	-10.4
Average time steps	A2C	90.2	52.0	61.5	62.2	64
	PPO	298.0	250.7	230.0	198.2	158.5

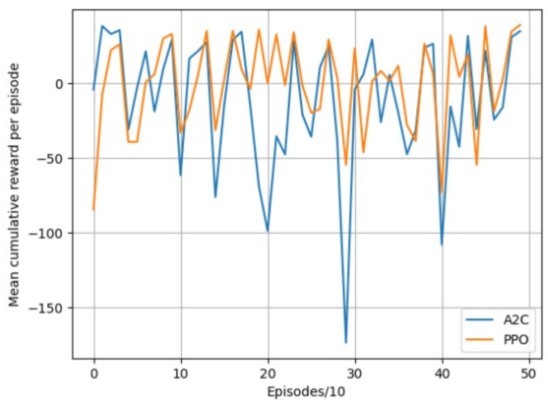
Table 11: Test III: Comparison of some performance measures of the A2C and PPO algorithm.

4.3.3. Evaluation of Test IV

In this experiment the shape of the flow channel inside the profile extruder is altered through FFD. This method was used as an alternative to directly changing the coordinates, as the mesh got tangled a few times when directly changing the coordinates of the corner points of the geometry. This was then minimized by adjusting the clipping range. In order to evaluate if the algorithms can learn, the agent is trained for 500 episodes and the mean length and cumulative reward per episode are plotted. Like in the other experiments, this is done for the A2C algorithm and the PPO algorithm. Figure 32a and 32b show a comparison of these two values for the agents. This time both agents could improve their policy as well, but the average time steps still fluctuated strongly even after 400 episodes. Both agents have problems to find a good policy. The experiment was done again with the standard hyperparameters, as each run of 500 episodes need roughly two days on the Rheinisch-Westfälische Technische Hochschule (RWTH) Compute Cluster. Adjusting the learning rate to a lower level and/or having a lot more episodes might improve the policy. The maximum episode length was set to 1000 time steps to accelerate the training.



(a) Mean episode length.



(b) Mean cumulative reward per episode.

Figure 32: Test IV: Comparison of the A2C and PPO agent with standard hyperparameters.

To make the two algorithms better compareable some statistical values are calculated in Table 12. The A2C algorithm shows no significant improvement over the first 500 steps in all performance measures. This time, both algorithms perform similarly, which can also be seen in the total time needed for the completion of the 500 episodes.

4.4. Runtime

The RL-algorithm in general is really fast, so a lot of steps and episodes can be calculated in a short period of time. This means that the agent can learn a policy for the best actions in a short period of time. As the test scenarios get more complicated and the FEM-solver has to be used the time per step increases which makes it harder for the agent to learn the best actions as the total time for learning a policy is limited. In

		Episodes				
Key figure	Algorithm	0-100	100-200	200-300	300-400	400-500
Goal reached	A2C	95	94	89	95	93
	PPO	88	90	88	89	88
Avg. cumulative reward -per episode	A2C	10.2	-10.7	-35.1	-7.5	-11.7
	PPO	-5.9	3.6	0.3	-3.7	2.5
Average time steps	A2C	186.5	236.9.5	264.9	230.6	195.1
	PPO	283.7	211.8	201.1	265.4	231.6

Table 12: Test IV: Comparison of some key figures between the A2C and PPO algorithm.

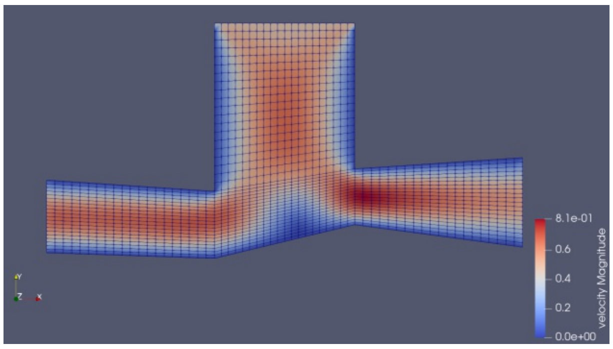
Table 13 there is a comparison of the computation times of the different models. All models except for the Matyas function (Test I) use 8 threads for the parallelization of the simulation and a relatively coarse mesh with 1600 elements. An interesting observation is that the A2C needs less total time for all four performed experiments. An explanation could be that the learning rate has a higher standard value than for than the PPO agent, which makes it more sample efficient if it can find a good policy. The PPO agent additionally uses a clipping range, which limits how much the policy can change per time step. A more thorough look on the performance will be given in the respective chapters for the experiments.

Average time per time step				
Algorithm	Test I	Test II	Test III	Test IV
A2C	$0.0018\ \frac{s}{step}$	$1.38\ \frac{s}{step}$	$1.31\ \frac{s}{step}$	$1.43\ \frac{s}{step}$
PPO	$0.00165\ \frac{s}{step}$	$0.97\ \frac{s}{step}$	$1.52\ \frac{s}{step}$	$1.51\ \frac{s}{step}$
Total time needed for 500 episodes (d-hh:mm:ss)				
Algorithm	Test I	Test II	Test III	Test IV
A2C	$0-00:00:50$	$0-11:17:17$	$0-12:18:25$	$1-22:17:50$
PPO	$0-00:00:58$	$1-07:27:47$	$1-23:46:28$	$2-01:56:56$

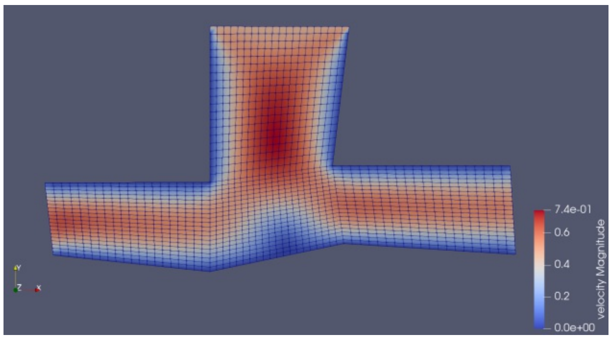
Table 13: Comparison of the runtimes for the four experiments with standard hyperparameters.

4.5. Comparison of the optimized shapes

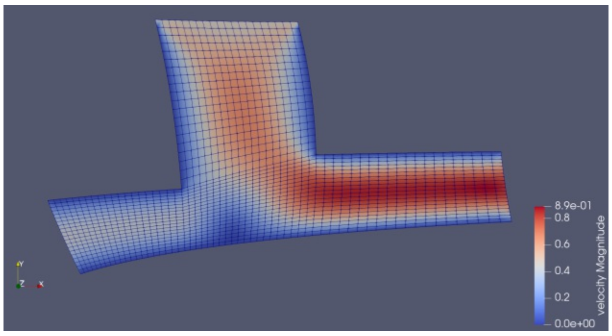
Figure 33 shows how the flow channel for the extruder can look like when optimized with the three different methods. This is just an example and varies a lot, as the only constraint is the mass flow ratio $\dot{m}_{23}$. All three geometries have been optimized to achieve the same mass flow ratio $\dot{m}_{23}$. In Test II, Figure 33a the corner points of the geometry could be adjusted only vertically so the lines limiting the entry of the flow channel are parallel. In Test III, Figure 33b the corner points could also be moved horizontally, so the lines do not have to be parallel anymore, although it is possible depending on the actions the agent chooses. The shape for the FFD, depicted in Figure 33c is constructed with B-Splines of the order two, because of that the lines are not straight. This makes the deformations smooth, and the shape looks more natural.
It is difficult to draw an exact conclusion regarding which shape parameterization is target-oriented for a real application. In the tests, it has been shown that fewer DOF can help the agent learn a good strategy quickly. The FFD test case had problems learning a good strategy because too many DOF were used. However, the shapes created look more natural and the big advantage with this parameterization is that arbitrary shapes can be optimized without much adjustment in the code. One approach to solve this problem would be to limit the DOF, either with another free-form or by limiting the DOF of the DV.



(a) Test II: Sample shape with a mass flow ratio of $\dot{m}_{23} = 1$.



(b) Test III: Sample shape with a mass flow ratio of $\dot{m}_{23} = 1$.



(c) Test IV: Sample shape with a mass flow ratio of $\dot{m}_{23} = 1$.

Figure 33: Comparison of the shapes produced by the three different experiments (Test II-IV) for a mass flow ratio of $\dot{m}_{23} = 1$.

5. Conclusion and Outlook

The aim of this thesis is to evaluate if shape optimization through reinforcement learning is possible and if the number of time steps needed for the optimization can be improved by training an agent for a fixed number of episodes. The mass flow ratio of a geometry, which can be seen as an abstraction of the flow channel inside a profile extruder, had to be changed to a pre-defined value. Two different methods for parameterizing the geometry are introduced — one directly changes the coordinates of the corner points of the geometry, while the other one changes the geometry through FFD. In the first simple test function, the hyperparameters of the RL algorithms are altered, and their performance is evaluated. This is also used to establish an appropriate reward function. This test function showed that a dynamic reward function performs better than a constant reward function. The hyperparameters should not be altered, as the small improvement that can be achieved by adjusting them does not justify the risk of an unstable training.
Three tests for the shape optimization problem were performed, two of them directly change the geometry, while the last one changed the geometry with FFD. The tests were performed in order of increasing DOF. It can be shown that the difficulty for the RL agent increases with the number of DOF. The reason is that the RL agent has to evaluate each action for its expected reward, and each action corresponds to one degree of freedom in our experiments.
In the two tests that directly change the geometry, the agents could improve their policy over the training episodes. The A2C agent performed better on average than the PPO agent. In the test with the FFD, the performance of both agents stayed fairly constant over the training episodes and no difference between the agents in terms of time steps per episode or cumulative reward per episode was observed.
Directly changing the geometry has the advantage that fewer DOF are needed. A disadvantage is that the so generated mesh can get tangled easily, and the shapes produced look very angular. The experiment with the FFD had the most DOF, and would have needed more training episodes to improve its policy. Nevertheless, the shapes produced by the FFD look the most natural and the big advantage with this parameterization is that arbitrary shapes can be optimized without much adjustment in the code.
To further speed up the time steps needed per episode, the hyperparameters can be optimized, as well as the reward function. A better trained agent can be created by training the agent longer. A different approach would be using an algorithm, which is more sample efficient, as most of the calculation time is spent in the FEM simulation. The parameterization can be adjusted, as the DOF make a huge difference in the calculation time. A second criterion for a good shape is needed apart from the mass flow ratio to introduce some constraints for a shape, that is close to a real-world application. Finally,

if enough progress is made in the two-dimensional test cases, a three-dimensional test case can be created, evaluated, and tested as a real-world model.

A. Appendix

A.1. OpenAI Gym interface

```
import gym
from gym import spaces

class CustomEnv(gym.Env):
  """Custom Environment that follows gym interface"""
  metadata = {'render.modes': ['human']}

  def __init__(self, arg1, arg2, ...):
    super(CustomEnv, self).__init__()
    # Define action and observation space
    # They must be gym.spaces objects
    # Example when using discrete actions:
    self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)
    # Example for using image as input (channel-first; channel-last
 also works):
    self.observation_space = spaces.Box(low=0, high=255,
                                        shape=(N_CHANNELS, HEIGHT,
 WIDTH), dtype=np.uint8)

  def step(self, action):
    ...
    return observation, reward, done, info
  def reset(self):
    ...
    return observation  # reward, done, info can't be included
  def render(self, mode='human'):
    ...
  def close (self):
    ...
```

Listing 2: OpenAi gym interface

References

[1] H Ettinger, J. Sienz, John Pittman, and I Szarvasy. Parameterization techniques for two-and three-dimensional automated optimization of pvc extrusion dies. In *Polymer Processing Society (ed.) PPS 18*, page 62, 06 2002.

[2] Hongjing Ji, Osama Alfarraj, and Amr Tolba. Artificial intelligence-empowered edge of vehicles: Architecture, enabling technologies, and applications. *IEEE Access*, December 2017.

[3] Jörg Frochte. *Maschinelles Lernen*. Hanser, München, second edition, 2019.

[4] Chire. : Website https://commons.wikimedia.org/wiki/file:k-means_convergence.gif [online], May 2017.

[5] Josh Achiam. : Website https://spinningup.openai.com [online], June 2021.

[6] Jiexin Xie, Zhenzhou Shao, Yue Li, Yong Guan, and Jindong Tan. Deep reinforcement learning with optimized reward functions for robotic trajectory planning. *IEEE Access*, 7:105669–105679, 2019.

[7] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[8] Richard S. Sutton and Andrew G.Barto. *Reinforcement learning: An Introduction*. MIT Press Cambridge, Massachusetts, second edition, 2018.

[9] Rajendra Koppula. : Website https://www.manifold.ai/exploration-vs-exploitation-in-reinforcement-learning [online]. Manifold, 2020.

[10] Julien Vitay. : Website https://julien-vitay.net/deeprl/ [online], 2021.

[11] Arthur Juliani. : Website https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2 [online], December 2017.

[12] Lila Weng. : Website https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html [online], February 2019.

[13] I. Szarvasy, J. Sienz, John Pittman, and E. Hinton. Computer aided optimisation of profile extrusion dies. *International Polymer Processing*, 15:28–39, 03 2000.

[14] Stefanie Elgeti, Markus Probst, Christian Windeck, Marek Behr, W. Michaeli, and Christian Hopmann. Numerical shape optimization as an approach to extrusion die design. *Finite Elements in Analysis and Design*, 61:35–43, 11 2012.

[15] H.J. Ettinger, J. Sienz, John Pittman, and Andrey Polynkin. Parameterization and optimization strategies for the automated design of upvc profile extrusion dies. *Structural and Multidisciplinary Optimization*, 28:180–194, 01 2004.

[16] Roland Siegbert, Johannes Kitschke, Hatim Djelassi, Marek Behr, and Stefanie Elgeti. Comparing optimization algorithms for shape optimization of extrusion dies. *PAMM*, 14, 12 2014.

[17] Geoffrey E. Hinton Alex Krizhevsky, Ilya Sutskever. Imagenet classification with deep convolutional neural networks. *Communication of the ACM*, 60:84–90, 2017.

[18] Rupali Roy. : Website https://towardsdatascience.com/understanding-the-difference-between-ai-ml-and-dl-cceb63252a6c [online], April 2020.

[19] R. Gross. *Psychology: The Science of Mind and Behaviour 6th Edition*. Hodder Education, 2012.

[20] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019.

[21] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Alan Apt, 1995.

[22] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning - with Applications in R*. Springer, New York, NY, 2013.

[23] Demis Hassabis, Shane Legg, Lila Ibrahim, Koray Kavukcuoglu, and Colin Murdoch. : Website https://deepmind.com/research/case-studies/alphago-the-story-so-far [online], June 2021.

[24] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *The Journal of artificial intelligence research*, 4:237–285, 1996.

[25] Andrej Andreevic Markov. *Theory of algorithms*. Works of the Steklov Mathematical Institute. Israel Program for Scientific Translations, Jerusalem, 2. impression. edition, 1962.

[26] Yuxi Li. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018.

[27] Kristopher De Asis, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton. Multi-step reinforcement learning: A unifying algorithm, 2018.

[28] Abhishek Suran. : Website https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f [online], July 2020.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[30] J. Langford S. Kakade. Approximately optimal approximate reinforcement learning. *ICML*, 2:267–274, 2002.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[32] Yoshua Bengio. Practical recommendations for gradient-based training of deeparchitectures. *arXiv:1206.553v2*, 2012.

[33] Yuhuai Wu, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman. : Website https://openai.com/blog/baselines-acktr-a2c/ [online], August 2018.

[34] Amanda Lampton, Adam Niksch, and John Valasek. Reinforcement learning of a morphing airfoil-policy and discrete learning analysis. *Journal of Aerospace Computing, Information, and Communication*, 7(8):241–260, 2010.

[35] Amanda Lampton, Adam Niksch, and John Valasek. Morphing airfoils with four morphing parameters. pages 2008–7282, August 2008.

[36] Jonathan Viquerat, Jean Rabault, Alexander Kuhnle, Hassan Gharaieb, Aurelin Larcher, and Elie Hachem. Direct shape optimization through deep reinforcement learning. *arXiv*, 2020.

[37] Xinghui Yan, Jihong Zhu, Minchi Kuang, and Xiangyang Wang. Aerodynamic shape optimization using a novel optimizer based on machine learning techniques. *Aerospace Science and Technology*, 86:826–835, 2019.

[38] Runze Li, Yufei Zhang, and Haixin Chen. Learning the aerodynamic design of supercritical airfoils through deep reinforcement learning. *CoRR*, abs/2010.03651, 2020.

[39] Xinyu Hui, Hui Wang, Wenqiang Li, Junqiang Bai, Fei Qin, and Guoqiang He. Multi-object aerodynamic design optimization using deep reinforcement learning. *AIP Advances*, 11(8):085311, 2021.

[40] J. Austin Cottrell, Thomas J. R Hughes, and Yuri Bazilevs. *Isogeometric analysis : toward integration of CAD and FEA*. J. Wiley, Chichester, West Sussex, U.K. ; Hoboken, NJ, 2009.

[41] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *ACM Computer Graphics, Siggraph*, 20(4):151–160, 1986.

[42] Olgierd C Zienkiewicz, Robert L Taylor, and Perumal Nithiarasu. *The finite element method for fluid dynamics*. Butterworth-Heinemann, Oxford [u.a.], 7. edition, 2014.

[43] Doug McLean. Continuum fluid mechanics and the navierstokes equations. In *Understanding Aerodynamics*, pages 13–77. John Wiley & Sons, Ltd, Chichester, UK, 2012.

[44] Jean Donea and Antonio Huerta. *Finite Element Methods for Flow Problems*. John Wiley & Sons Incorporated, [Place of publication not identified], 2003.

[45] Lutz Pauli. *Stabilized Finite Element Methods for Computational Design of Blood-Handling Devices*. PhD thesis, RWTH Aachen, 2016.

[46] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[47] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[48] Brandon Brown and Alexander Zai. *Deep Reinforcement Learning in Action*. Manning Publications Co. LLC, New York, 2020.

[49] Joshua Hare. Dealing with sparse rewards in reinforcement learning. *CoRR*, abs/1910.09281, 2019.