Automated test case generation for satellite on-board image processing using reinforcement learning

Master's thesis
in partial fulfilment of the requirements for the degree of
Master of Science

University of Münster
Computer Science Department

Supervisor:
Ulrike Witteck

First assessor:
Prof. Dr. Paula Herber

Second assessor:
Prof. Dr. Malte Schilling

Submitted by:
Rabea Ludorf

Münster, January 2026

List of Acronyms

ANN	Artifical Neural Network
CCD	Charge Coupled Device
DL	Deep Learning
DLR	German Aerospace Center
DQL	Deep Q-Learning
DRL	Deep Reinforcement Learning
EC	Equivalence Class
ESA	European Space Agency
FGS	Fine Guidance System
FPA	Focal Plane Assembly
GA	Genetic Algorithm
GCN	Graph Convolutional Network
ML	Machine Learning
MLP	Multilayer Perceptron
PLATO	PLAnetary Transits and Oscillations of stars
QUEST	QUaternion ESTimator
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SUT	Software Under Test

Contents

1 Introduction

In the satellite domain, on-board image processing algorithms must run for a long time without causing problems and they must be maintainable in space for several years. Therefore, such algorithms have to satisfy strict criteria, for example regarding mathematical accuracy and reliability in hard real time. Some of these algorithms are even important for achieving the mission goal. To ensure that all requirements are met, the algorithms must be thoroughly tested at an early stage of the development process. This is a challenge due to the large input range of the algorithms, as it is not always possible to test all possible inputs. Therefore, there are approaches to generate inputs that provoke critical behavior. One approach to automated generate test cases for on-board image processing algorithms is proposed by Witteck et al. [WGH21] in the form of a Genetic Algorithm (GA). However, GAs have some problems, which is why we propose a Reinforcement Learning (RL) approach to automated generate test cases for on-board image processing algorithms. We use RL to see if it can handle these problems better. To the best of our knowledge, there are no other approaches based on RL to automated generate test cases for satellite on-board algorithms.

1.1 Problem

Due to the large input range of on-board image processing algorithms, it is not feasible to test every possible input. However, it is important to select significant inputs that produce critical behaviour with respect to certain criteria in order to increase the robustness of such algorithms. In this thesis, the most important test criterion is the mathematical accuracy of the algorithm. However, due to the complexity of on-board image processing algorithms, it is often not obvious which inputs cause critical behaviour. In [WGH21], the authors propose an approach in which, firstly, the input range is reduced by equivalence class tests and, secondly, a GA is used to automatically search for the most important inputs in a reduced search space. The problem of GAs is that they often have a long runtime, and they do not necessarily find a global optimum but remain in a local optimum. In addition, for every execution of a GA a lot of configuration is required and the success of the algorithm depends partly on the configuration of the parameters. For example, according to the authors of the paper [WGH21], the tournament size must be large enough to generate significant selection pressure, otherwise the test cases will not develop further. Furthermore, the results of

GAs are not always easy to understand. These problems can lead to a loss of confidence in the results of the GA, in terms of whether the generated test inputs actually trigger mission-critical behavior. In addition, only someone with extensive knowledge of the configuration can execute the GA, and configuring the GA can be time-consuming.

1.2 Goal

The goal of this thesis is to present an approach which automated generates test cases to test on-board processing algorithms with regard to mathematical accuracy, which provoke mission-critical behaviour. The approach should be less susceptible to local optima and possibly find a global optimum. Another requirement of the proposed approach is that the user needs less knowledge during execution. Finally we want to compare the results of the new approach with the results of the GA from the previous mentioned paper [WGH21].

1.3 Proposed Solution

In this thesis we propose an approach which generates test cases for the robustness testing of satellite on-board image processing algorithms in an efficient and automated way.The key idea of our solution is a Deep Q-Learning (DQL) algorithm that receives feedback from an image processing algorithm and learns the properties of test cases that are most significant for robustness testing. We explain our approach by applying it to the Fine Guidance System (FGS) from the European Space Agency (ESA) mission PLAnetary Transits and Oscillations of stars (PLATO). We choose the FGS because it is a complex satellite on-board algorithm. Furthermore, Witteck et al. also use FGS as a case study in their paper, which allows us to directly compare the results of both approaches.

First, we will define an environment to encode the problem of test case generation for on-board image processing algorithms. In order to reduce the search space, we use the in [Wit+25] proposed Equivalence Class (EC) definitions. One state of our environment represents one test case. The input for the FGS algorithm is a combination of stars with a predefined number of stars. For this reason, a state of our environment is a combination of stars. The environment will be modified by the actions of one agent. A possible action is to replace one star of the combination with another star in favour of a higher reward. As part of the environment definition we also develop three different reward functions. Since the fitness function of the in [WGH21] proposed GA is well suited to find a local optimum, the reward functions will be based on this fitness function. Then, we propose two variants of a DQL approach. The principle behind our approach is that our environment initially starts with a randomly generated test input. Our RL agent then improves the generated test case

with regard to a specific test objective by selecting actions. During execution, the agent learns how to select actions in order to better achieve the test objective. Since we generate the initial test input randomly, it is important that the algorithm runs several times with different initial test inputs in order to reduce the influence of the initial test input on the final result. We achieve this by regularly resetting our algorithm to a new, randomly generated test input during the run.

We are less interested in the strategy learned by the RL algorithm, than in the final combination as a result. In the last part of our thesis, we evaluate the results of our novel testing approach and compare these results with the results of the GA of Witteck et al.

The implementation of the approach will take place in Python, as there is a large knowledge base and many libraries for machine learning in Python. Since the FGS is written in C++, we will use the library pybind11 to write a Python binding of the C++ code. Optionally, we will extend our concept to a multi-agent RL approach as presented in [Sco+21].

2 Fundamentals

In this chapter, we first provide an overview of the background of the case study and its purpose. Then, we explain the genetic approach [WGH21] by Witteck et al. and summarise how GAs work, as this is the approach we use to compare our results with. Next, we provide an introduction to Machine Learning (ML) and Deep Learning (DL). The introduction to ML covers the basics of RL with Q-Learning as example. Lastly, we introduce DQL, on which our approach is based.

2.1 Fine Guidance System of the PLATO mission

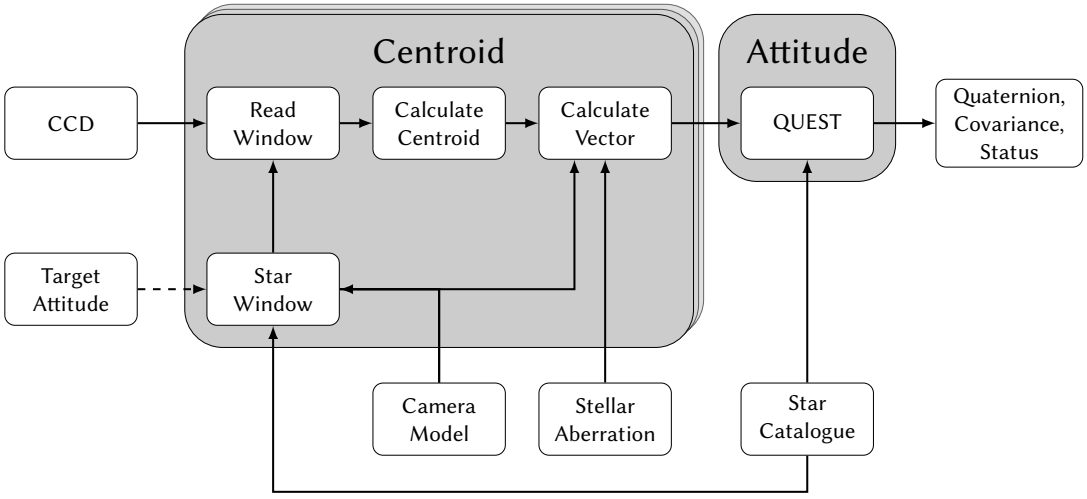


Figure 2.1: FGS algorithm overview [GWP21]

The main goal of this work is the generation of test cases for robustness testing of satellite on-board image processing algorithms. For this reason, we use an algorithm from the space domain as a case study. Namely the FGS from the ESA mission PLATO, developed by the German Aerospace Center (DLR) which is a mission-critical part of the mission [Rau+25]. The mission's purpose is to search for earth-like exoplanets orbiting in the habitable zone of sun-like stars. The satellite observes stars over a long period of time with very high precision using 26 different cameras. There are

two types of cameras: firstly, 24 normal cameras, which are used exclusively to collect scientific data during the mission, and secondly, two fast cameras, which also provide input for a precise attitude calculation of the satellite. The normal cameras have a slower cycle time of 25 s than the fast cameras, which have a cycle time of 2.5 s. To measure high precision data it is important to important to achieve a very high pointing stability of the satellite during this time. This is the responsibility of the FGS algorithm.

Each fast camera comprises four Charge Coupled Devices (CCDs) and is connected to a data processing unit that runs the FGS algorithm. The inputs for the FGS are the data from the CCDs as well as the target attitude, the corresponding star catalogue, the camera model and stellar aberrations, as shown in Fig. 2.1. First the FGS calculates the expected position on the CCD for each star in the star catalogue. For that it uses the target attitude and the camera model. The algorithm then uses the expected star position to read out sub-images from the CCDs for each star in the star catalogue. Each of the sub-images contains only one star. Next, the FGS calculates the centroid position for each sub-image. The algorithm then converts the centroid position into a direction vector in the boreseight reference frame, taking into account the stellar aberration. Finally, the QUaternion ESTimator (QUEST) uses the measured star direction vectors and reference directions from the star catalogue to calculate the attitude data.

During the development and testing phase of the FGS, several important limitations were identified. These include that the number of input stars of the FGS is 30 and non of these stars should appear twice in the input combination [WGH21].

2.2 Genetic Approach

In [WGH21] Witteck et al. present a genetic test approach, also using the PLATO FGS as a case study.

In general, GAs search for solutions to optimisation problems in a large search space. These algorithms simulate evolution within a population over several generations in order to optimise a solution which is encoded as an individual [Mir19]. Individuals consist of genes that represent parameters of the optimization problem. A population consists of a certain number of different individuals. Over several generations, a population changes due to evolutionary pressure based on the fitness value of individuals, which is calculated using a predefined fitness function. The fitness value describes how likely an individual will be selected into the next generation. This mechanism is called selection. In addition, genetic algorithms utilise two further evolutionary mechanisms, namely mutation and crossover to preserve the diversity of the population. Mutation

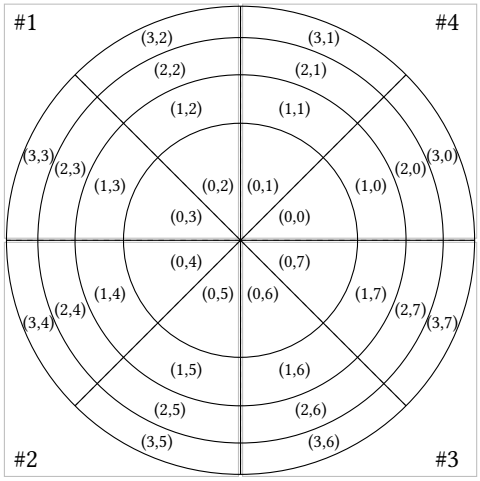


Figure 2.2: EC definition of FPA position [Wit+25]

0	1	0
1	2	1
0	1	0

Figure 2.3: EC definition of sub-pixel position [Wit+25]

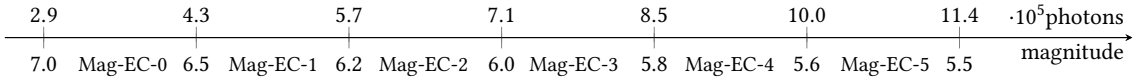


Figure 2.4: EC definition of magnitude [Wit+25]

is the exchange of genes with the gene pool and crossover is the exchange of genes between individuals in the population.

Their test suite consists of stars classified by three parameters: position on the Focal Plane Assembly (FPA), magnitude and sub-pixel position. First, they reduce the given test suite by defining ECs for each input parameter and a multidimensional coverage criterion. Second, they propose a GA to automated search in the reduced test suite for test cases which are relevant for robustness testing.

The latest EC definitions are from the new paper [Wit+25] by Witteck et al. Their definition of the ECs of the FPA position is divided into a radius and a sector and represents the area on the FPA where the star is located Fig. 2.2. The EC of the sub-pixel position indicates where on a sub-pixel the star is detected, divided into centre, edge and corner Fig. 2.3. Finally, the EC of the magnitude indicates the brightness of a star Fig. 2.4. It is important to note that the EC limits of the FPA position and the magnitude can be changed, but there are always three ECs of the sub-pixel position. The multidimensional coverage criterion defined in [WGH21] is satisfied if each EC combination is represented by at least one star.

In [WGH21], the goal is to find test cases that provoke mission-critical behaviour. The generated test cases are a combination of stars from a predefined star pool. This star pool is the test suit and thus the search space of the GA. A test case that contains 30 stars is represents an individual. Each

individual has a fitness value calculated based on the variance of the attitude estimation. The latest fitness function is defined in [Wit+25] as follows:

$$\text{fit} = \sqrt{\sigma_x^2 + \sigma_y^2 + \left(\frac{\sigma_z}{4}\right)^2} \tag{2.1}$$

Where σ^2 is the variance of the attitude estimation, expressed in Euler angles, in milliarcseconds (mas). Individuals with a high fitness value have a higher chance to be selected into the new generation. After mutation, i.e. the replacement of single stars in an individual with stars from the star pool, and crossover, i.e. the exchange of single stars between two individuals, the new fitness values for the individuals in the new generation are calculated and the selection process begins again until a predefined number of generations is reached.

2.3 Machine Learning

ML approaches simulate human learning. Two branches of ML are RL and DL. A specialisation of RL is Q-learning. There also exists combination of RL and DL, which is called Deep Reinforcement Learning (DRL). An example of such a combination is represented by DQL, which is the combination of Q-learning and DL.

2.3.1 Reinforcement Learning

RL is a broad field with numerous variations, and an overview is given in [KLM96]. Well-known areas of application are video games and robot control. The main components of RL are an agent

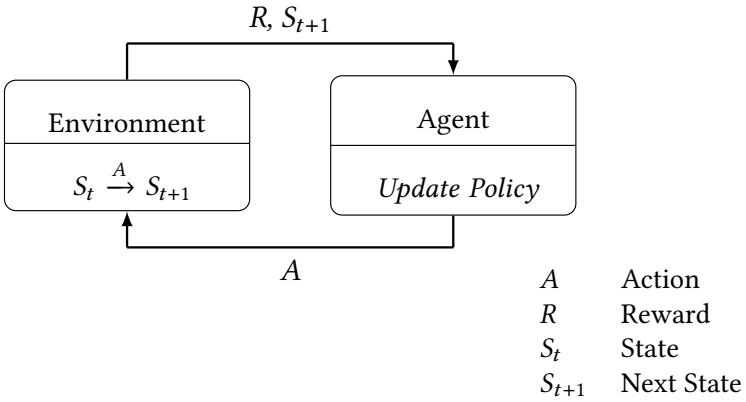


Figure 2.5: Basic RL workflow

and an environment. The environment encodes a given problem, such as a pathfinding problem. The environment has a specific state at any given time. The agent observes the states of the environment to map states to predefined actions in its policy. The general concept is that the agent selects actions from a predefined action space in the current state of the environment as shown in Fig. 2.5. The selected action influences the environment and thereby changes its state within the state space. As figure 2.2 depicts, the agent then receives feedback for each action in the form of a reward from the environment and the new state of the environment. The reward is calculated using a predefined reward function. Based on the feedback, the agent develops and improves a strategy, called policy, to find the optimal action in each environmental state.

The algorithm runs with a predefined number of episodes and a predefined maximum length for these episodes. At the start of the execution, the environment is reset to an initial state. The user determines which state is considered to be an initial state. Then, in each step of an episode, the agent performs actions that lead to a change in the state of the environment. For each action, the agent receives feedback in form of a reward. Depending on the implementation, the RL algorithm repeats theses steps until the environment has reached a predefined final state or the maximum number of steps per episode has been reached. After each episode, the environment is usually reset to an initial state again. Depending on the implementation, the initial state is always the same or one from a predefined set of initial states. An algorithms run consists of two overlapping phases, the exploration phase and the exploitation phase. During the exploration phase, the agent collects information about the state to action mapping through randomly selected interactions with the environment within the action space. The agent learns from the feedback it receives from the environment for each action. In the exploitation phase, the agent acts on the basis of the knowledge it has acquired. It is important to find a good balance between the two phases in order to enable effective learning. If the exploration phase takes too long compared to the exploitation phase, the agent cannot use the knowledge gained to improve the solution. If, on the other hand, the exploitation phase is too short, the agent can only make decisions based on the few insights gained and cannot learn new, better courses of action and improve its policy.

A possible solution to decide whether the agent should explore or exploit is the ϵ-greedy algorithm. By the means of the ϵ-greedy algorithm, the agent explores if a randomly generated number is less than ϵ. On the other side, the agent decides based on its strategy if the generated number is greater than ϵ. ϵ usually starts at $\epsilon = 1$, which represents one hundred percent probability for exploration. Then ϵ is typically reduced over time to decrease the probability of exploration and increase the probability of exploitation. The exploration and exploitation phases are balanced by a faster or slower reduction of ϵ.

Q-Learning One of the simpler approaches of RL is Q-learning. In this approach, the agent stores the knowledge it gains in a table which contains one value for each pair of possible state and action [WD92]. The value is based on the reward the agent receives from the environment after taking an action and is stored in the table in the cell associated with that action and the state. This stored knowledge is referred to as the Q-value. The Q-value is updated based on the Bellman equation for the optimal action-value function according to [SB+98]:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_a Q(S',a) - Q(S,A) \right] \tag{2.2}$$

The new Q-value is based on the current Q-value $Q(S,A)$, the reward obtained R, and the highest subsequent Q-value $Q(S',a)$. α is the learning rate and γ is the weighing factor of the highest subsequent Q-value. By considering the highest subsequent Q-value, it is possible to check whether the following possible actions could lead to a good solution. Since the goal is to obtain a high reward, the Q-value is higher for better actions. During the exploration phase, a Q-value for each observed state-action pair is stored in the table. In the exploitation phase, the agent selects the action with the highest Q-value for the current state. In this way, based on its current knowledge, the agent always chooses the optimal action in relation to its goal. The approach can be combined with the ϵ-greedy algorithm, then the agent explores when a randomly generated number is less than ϵ. The agent also acts randomly when it has no stored Q-values for the current state. Otherwise, it uses its knowledge and takes the action with the highest Q-value in the current state.

The Q-learning approach has the advantage of convergence: given sufficient time and memory, it always finds the global optimum. However, a major disadvantage is that its applicability is limited to smaller problems which can be encoded in a small state and action space. Otherwise, the table will become too large.

2.3.2 Deep Learning

DL is another variant of ML. The idea behind DL is to simulate human learning with layers of neurons that are combined into an Artifical Neural Network (ANN) [LBH15]. Areas of application include image and speech recognition and predictions in healthcare. ANNs receive data as input and learn a representation of this data in order to achieve a goal, for example, the classification of the input. An ANN consists of an input layer followed by any number of hidden layers and finally an output layer, as illustrated in Fig. 2.6. The number of hidden layers of an ANN corresponds to the depth of the network. In DL the depth of the network is normally at least two. Each layer has a predefined dimension corresponding to the number of neurons in the layer. The ANN in Fig. 2.6 for example has only one hidden layer with six neurons. In a fully-connected layer, each neuron is

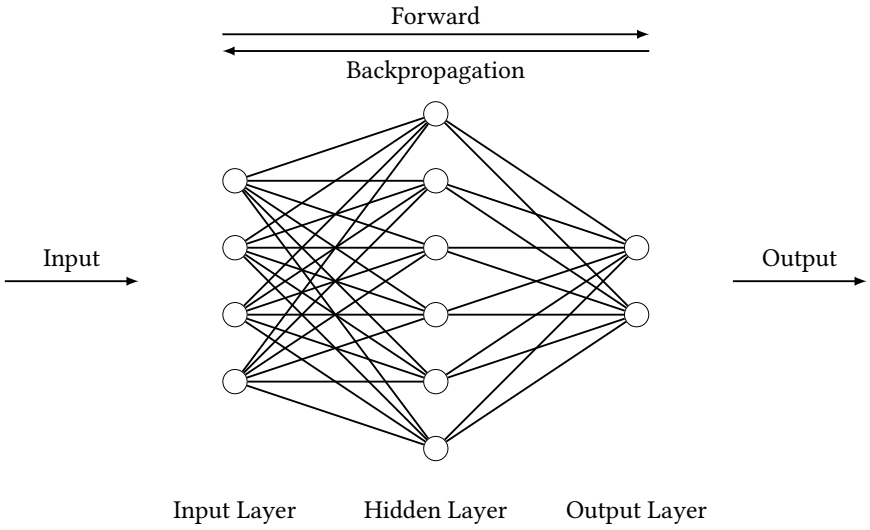


Figure 2.6: Artificial Neural Network

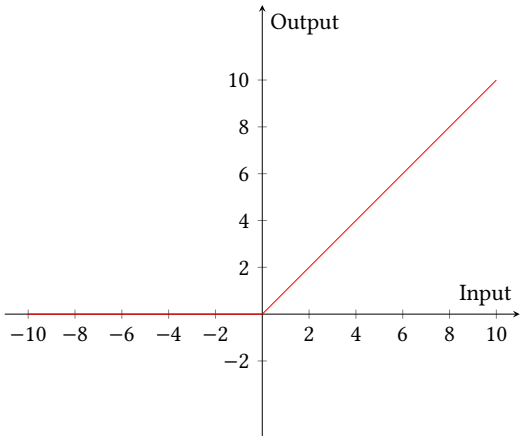


Figure 2.7: ReLU

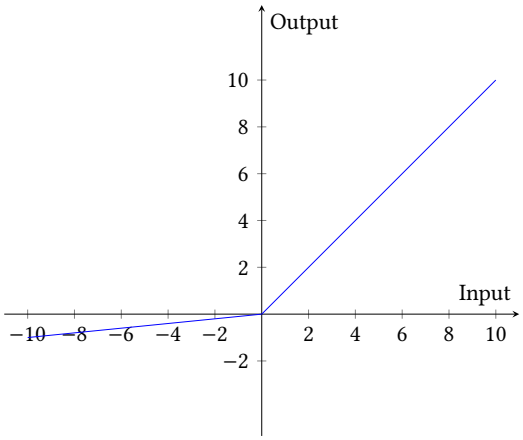


Figure 2.8: Leaky ReLU ($\alpha = 0.1$)

connected to every neuron in the next layer. ANNs with more than one layer and fully-connected layers are also called Multilayer Perceptrons (MLPs) [TM17].

During the forward process, the dimension of the input data of the ANN corresponds to the number of neurons in the input layer. The data flows from the input layer through the hidden layers to the output layer and is weighted between the individual layers. These weights are defined by the parameters of the ANN. Each hidden layer has an activation function, which ensures that the output of the layer is more than just a linear mapping of the input. The most popular function is Rectified Linear Unit (ReLU) [RAS20], shown in figure Fig. 2.7. However, ReLU does not take negative values into account. That might be problematic to solve some complex problems. For that case Leaky ReLU, shown in figure Fig. 2.8, is an alternative in which negative values are simply

scaled down by a constant factor α instead of being set to zero. The result delivered by the output layer is the predicted output of ANN. During training, this output is compared with a target output. For example, there is a training data set for classification, and an input is fed into the ANN for which the classification is known. Then the result of the ANN is the predicted output and the actual classification is the target output. A predefined loss function calculates the loss using the two outputs. A commonly used loss function is, for example, the mean squared error [Li23]. To improve the representation a backpropagation algorithm with the loss as input is used. The backpropagation algorithm adjusts the parameters of the different layers in order to reduce the feature loss.

Graph Convolutional Networks A special variant of ANNs are Graph Convolutional Networks (GCNs), which are specifically designed for graph-structured data. Because they regard the relationship, the edges between the nodes, as information [KW16]. Like ANNs GCNs also have an input layer, several hidden layers and an output layer, as well as activation functions. In contrast to ANNs, they work with graphs as input. A layer do not consist of neurons but of a representation of the input graph. While the structure of the graph always remains the same, the representation of the node features changes from layer to layer. The information flows between the edges of the graph, which are used to calculate an aggregation of all neighbouring information for one node. Then, the current features of the node are combined with the calculated weighted aggregation. In this way, the representation of a node is influenced by its direct neighbours after one hidden layer, and indirectly by its neighbours' neighbours after two hidden layers. We refer to the representation of each node after aggregation as node embedding. The task of the GCN is to adjust the weights of the aggregation in each layer. GCNs are used, for example, to analyse social networks, in biological information processing or in natural language processing [Ren+22].

2.3.3 Deep Reinforcement Learning

DRL is the combination of RL and DL. DRL makes use of both central ideas of the strategies. The agent and the environment and also ANNs. The combination of both strategies has existed since 2015 in various variants, the first of which was DQL [Li23]. Mnih et al. present in [Mni+15] a deep Q-network in the domain of Atari games.

Deep Q-Learning After Q-learning, DQL is the next step towards generalisation between similar states and actions and thus learning by transfer. The Q-table is replaced by an ANN, which learns an action-value function for calculating the Q-values [TYG17], and the target Q-value is calculated by a second network. Classic DQL has a single policy network called the Q-network. In addition, several techniques are added to stabilise the training, the target network that calculates the target Q-value,

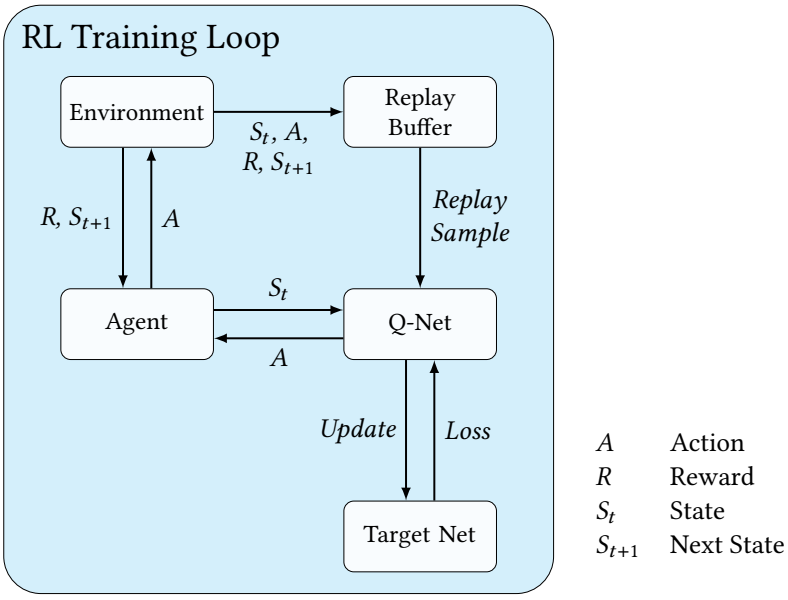


Figure 2.9: DQL training

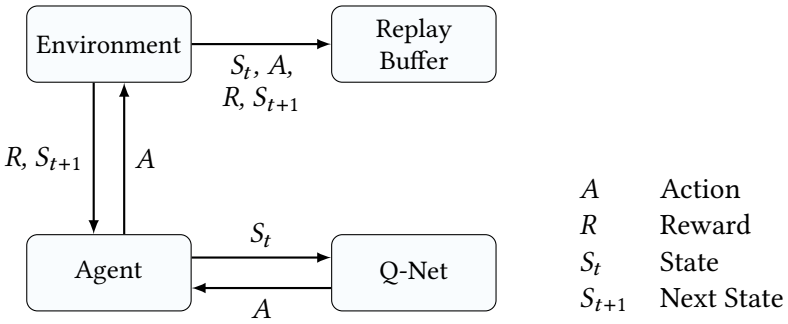


Figure 2.10: DQL experience collection

and the replay [Li23], as shown in Fig. 2.9. The target network has the purpose to "reduce the tight correlation between the Q-value and the target value"[Li23]. Whereas the replay is intended to break the close connection to previous experiences. In DQL, the interaction between the agent and the environment is the same as in Q-learning, and the interaction between the agent and the Q-network is very similar to the interaction between the agent and the Q-table in Q-learning. New are the replay buffer and the target network, as well as their interactions with the environment and the Q-network. The replay buffer is a structure that stores past interactions and their feedback according to the FIFO principle. Sample batches of these are later used for the replay, which is the actual training of the Q-network. The training consists of three different parts, which are performed with varying frequency: the experience collection, the replay and the update of the target network.

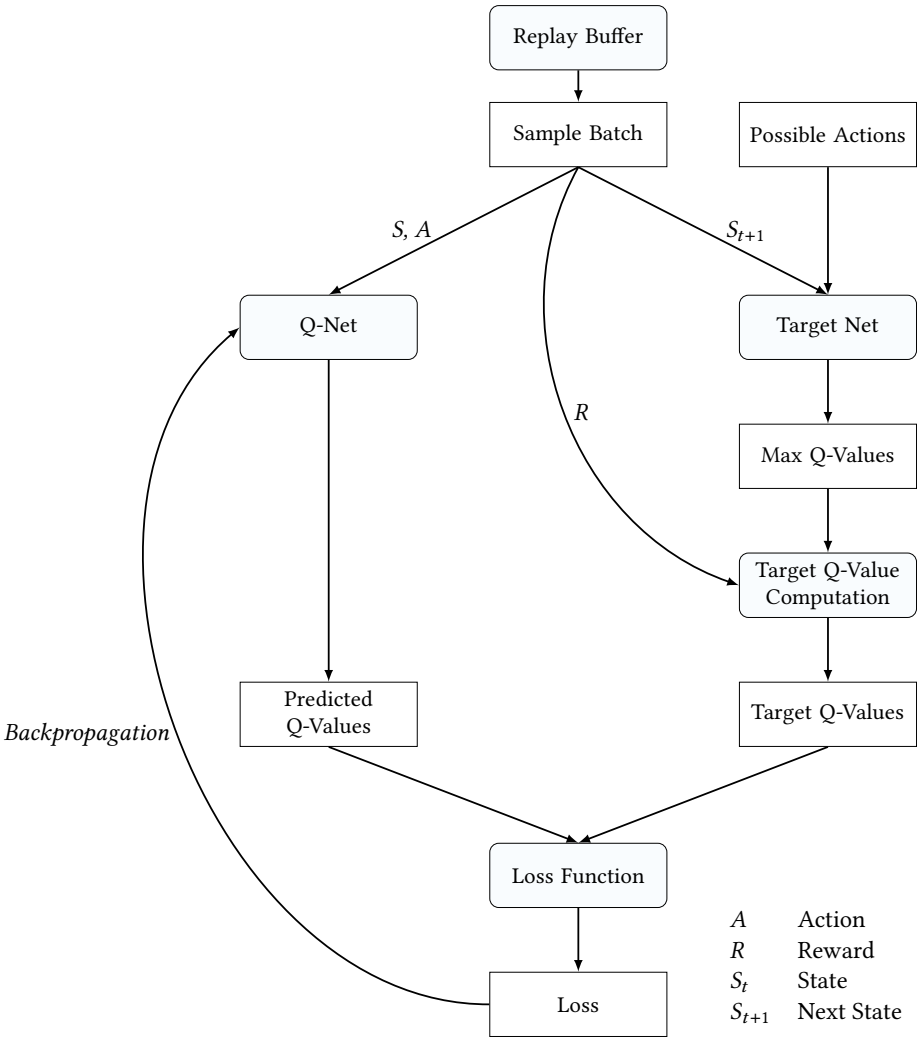


Figure 2.11: DQL replay

During the experience collection shown in Fig. 2.10, the agent selects either an action calculated by the Q-network depending on the current state of the environment or a random action from the action space at each step of the episode. The environment then changes its current state based on this action, and the agent observes the new state and receives a reward from the environment. At the same time, this tuple consisting of state, action, reward and next state is stored in the replay buffer. Depending on the size of the buffer, the oldest tuple is replaced.

Depending on the implementation, the replay begins after the replay buffer is partially filled and occurs at every step or, for example, only every four steps. First, in classic replay, a batch of tuples is randomly selected from the replay buffer, with the batch size being predefined Fig. 2.11. The batches of states and actions are fed into the Q-network and the batch of next states and the batched possible actions are fed into the target network. The output of the Q-network is a batch of predicted

Q-values, while that of the target network is a batch containing all Q-values for the possible actions. The highest Q-value is extracted from the second batch for each state into a new batch. These maximum Q-values and the rewards from the replay sample batch are then used to calculate the target Q-values, based on this rule [TYG17], similar to the update rule of Q-learning:

$$Q_{Target}(S, A) \leftarrow R + \gamma \left[\max_{a'} Q(S', a') \right] \tag{2.3}$$

Where γ weights the influence of the next highest Q-value. Next, the loss function calculates the loss based on the predicted Q-values and the target Q-values. Finally, the calculated loss is used by the backpropagation algorithm to update the weights of the Q-network.

Unlike the Q-network, the target network is not propagated backwards for updating. There are two common options for updating the target network. Each training step of the Q-network in the form of a soft update or in larger intervals as a hard update. The soft update has the form [Li23]:

$$\bar{w} \leftarrow \rho w + (1 - \rho)\bar{w} \tag{2.4}$$

$\bar{w}$ are the weights of the target network, which are updated with the weights of the Q-network w and its previous weights. The weights are weighted with the update rate ρ. The hard update corresponds to setting ρ equal to one, all parameters are simply copied from the Q-network to the target network. The soft update stabilises training more than the hard update [Zhu+24].

Feature Embeddings In [WLL24] Wu et al. present an approach to learning deep embeddings for numerical and categorical features for tabular data. For categorical features, they first learn separated embeddings. Then they concatenate these embeddings with the numerical features and fed this combination into an MLP. In this way, the MLP learns relationships between the features.

3 Related Work

There are various interesting papers that deal with the problem of automated finding significant test cases for robustness testing of algorithms and similar problems, such as code coverage. One of the papers, which deals with the problem of automated finding significant test cases, is the aforementioned work [WGH21]. There, the authors present an approach that uses a GA to automated generate test cases for the robustness testing of satellite on-board image processing algorithms. They use the FGS algorithm of the ESA mission PLATO as a case study. First, they define ECs and a multidimensional coverage criterion to reduce a given test suite. Second, they propose a GA to automated search in the reduced test suite for test cases which are relevant for robustness testing. This means test cases that trigger mission critical behaviour in the sense of high noise results. The GA from the paper delivers good results at a local level. The problem with this type of algorithms is that they do not necessarily find the best global solution. In addition, the user need knowledge about the problem and the search space so that the algorithm delivers good results. In contrast, the aim of this work is to propose an approach that is more likely to find a globally optimised solution. Furthermore, the user should require less prior knowledge to configure the parameters necessary for executing our algorithm.

An alternative approach to GAs for finding solutions to optimisation problems in a large search space is RL. The authors of the paper [Maz+21] give a brief overview of RL and its application in the field of combinatorial optimization. As the authors describe, RL works with an agent which acts in an environment. The environment encodes the problem that the algorithm should solve and provides feedback to the agent after each action. This feedback is based on a reward function and can be positive or negative. The higher the reward, the better the action. In this way the agent learns what a good action is and optimises its behaviour in the next iterations. As a result, the agent learns to optimise a solution for the given problem. The strategy developed by the algorithm is referred to as a policy. In the context of combinatorial optimisation problems, on which the authors of [Maz+21] focus, a possible action, is to select a combination of parameters or one specific path in a graph. Based on the feedback, the agent learns factors to decide which combinations are better than others. One of the RL approaches, that finds solutions to optimisation problems in a large search space in context of test case generation, is presented by the authors of the paper [Mog+20]. In the paper, they propose a framework that generates test data for different

programmes such as dcraw and n-queen without knowledge of the source code. The approach is used to identify bottlenecks in the performance of programmes and stores the policy beyond one execution of the RL algorithm. In this way, the algorithm is reusable for similar programmes and increases the efficiency of finding bottlenecks. The authors of the survey [KLM96] on RL describe typical problems of RL. The problems are quite similar to the problems of GAs. With RL, it is also difficult to find the globally best solution, and the algorithm can also get stuck in a local maximum. An important trade-off in RL that the authors mention is therefore exploration versus exploitation. The first is the ability to explore the environment and find other new solutions that may be better than the current best solution. The second is the ability to improve the current solution until it has reached its maximum. To address the problem of finding a globally best solution Scott et al. propose in [Sco+21] a multi-agent RL apporach. In this paper, the authors present a method called BanditFuzz, which is a "multi-agent reinforcement learning (RL) guided performance fuzzer for state-of-the-art Satisfiability Modulo Theories (SMT) solvers" [Sco+21]. This method offers the possibility of comparing different SMT solvers and finding performance problems. Therefore, a fuzzer provides a test input and the RL algorithm attempts to maximise the difference in performance between two solvers. One of these solvers is one of a set of target solvers and the other is the reference solver. The algorithm behind this is based on RL. It is an interaction between two agents, which is why they call it multi-agent RL. The first agent maximises the current best test input and the second agent ensures that the algorithm finds more than just a local solution. The second agent monitors the learning of the first agent and decides whether in the next phase should be explored or exploited. The authors thus present an approach that improves the core problems of RL. However, this approach focuses on the comparison of the performance of SMT solvers and not on the generation of test cases for the robustness testing of satellite on-board image processing algorithms, which is the focus of this work.

4 Automated Test Case Generation using Reinforcement Learning

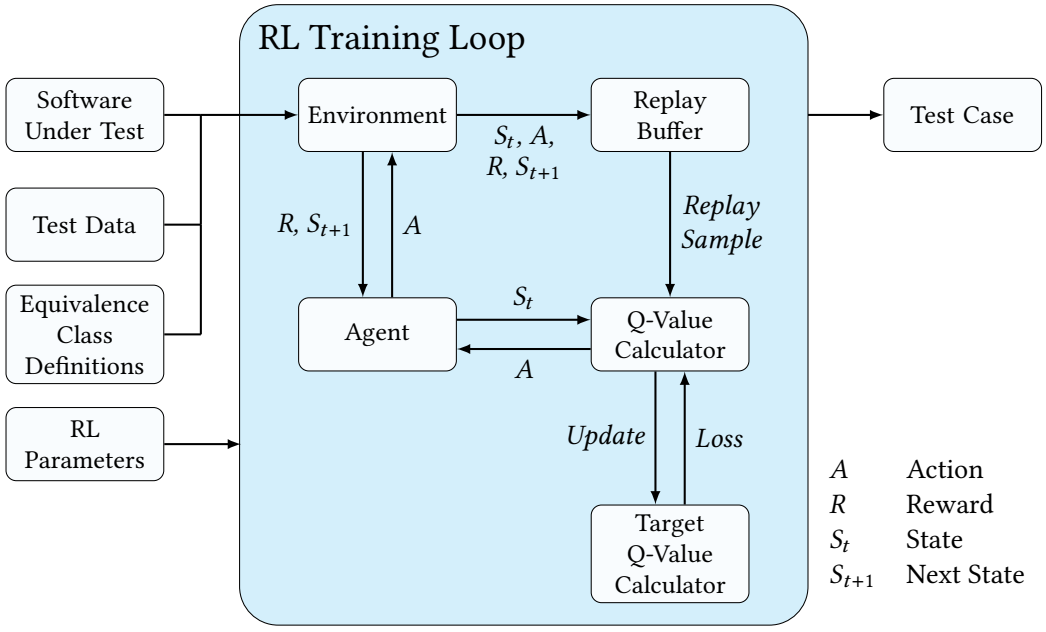


Figure 4.1: RL based test generation

The input domain of on-board satellite image processing algorithms is usually very large. This makes extensive testing nearly impossible. However, it is important to verify the correctness of such algorithms, as they are subject to strict functional requirements. Our goal is to present an approach for the automated generation of test cases for satellite on-board image processing algorithms using RL. The generated test cases should provoke mission-critical behaviour. The GA for automated test generation as proposed in [WGH21], probably often get stuck in a local optimum. This means it was not possible for the user to select a test case from the GA results that corresponds to a global worst-case test input. With our approach, we aim to reduce the probability of getting stuck. The challenges we face are adapting RL to the satellite domain in order to generate test cases, as well as the large search space and the definition of the reward function. The core idea of our approach is

to use RL in the form of DQL and define our environment as interface between the Software Under Test (SUT) and the agent. We present our test approach in the following chapter.

Like a typical RL approach our approach has an environment and an agent which interacts with the environment, as shown in Fig. 4.1. In our approach a state of the environment represents a test case. The environment assesses that test case by executing the SUT with the test case as input. The result of the SUT serves as the basis for the reward that the environment calculates and returns to the agent. This reward is intended to teach the agent to take actions that leads to an optimisation of the state of our environment and thus optimise the test case for a dedicated test goal. Our agent chooses between two options based on a random variable to select an action. Option one is that the agent selects the action randomly from the action space. Option two is that the agent feds the current state into the Q-value calculator. That Q-value calculator calculates the Q-value for every possible action. The agent then selects the action with the highest calculated Q-value. The Q-value calculator of Fig. 4.1 consists of multiple ANNs in order to reduce the search space and speed up the training. Our approach also uses replay and a target network to stabilise the training. The target network has the same structure as the Q-value calculator and we simple refer to it as target Q-value calculator.

Our approach takes the following inputs:

- SUT, in form of a satellite on-board image processing algorithm, to get information for the reward calculation
- Test data for running the SUT and initialising the state space
- EC definitions for specifying the boundaries of the state space and validating actions
- RL parameter to configure the RL training loop

The first three are direct inputs for the environment, and the last one is used to adjust the hyperparameters of the execution. The final output of the algorithm is either the worst or best test case found by the agent during execution, depending on the test objective.

4.1 Case Study: Fine Guidance System of the PLATO mission

To investigate the applicability of our approach we have chosen the FGS of the PLATO mission (Section 2.1) as a case study. Therefore, the FGS becomes our SUT as illustrated in Fig. 4.2. In the context of our case study, we define a test case as a combination of 30 stars.

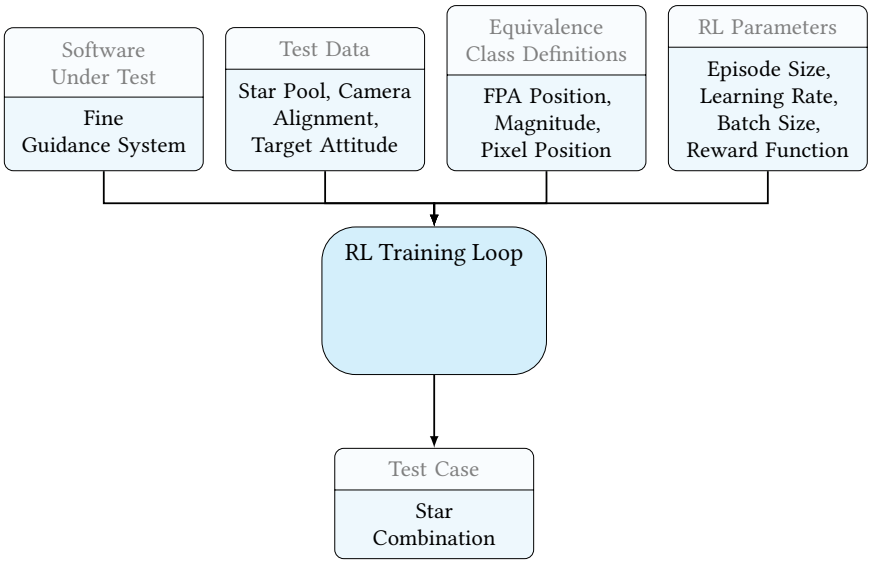


Figure 4.2: RL based test generation for the FGS

The inputs of the FGS are precalculated star directions for each star in a given star pool, as well as a camera alignment and a target attitude, see Section 2.1. The FGS uses the star directions to calculate the attitude quaternion for a combination of 30 stars. After calculating the quaternion, the FGS transforms it into the required reference frame. The star pool also contains the ID, FPA position, magnitude, and sub-pixel position for each star. Based on these parameters, the stars are allocated to predefined ECs, see Section 2.2. We assume that a star pool completely covers all EC combinations. Both the star parameters and the definitions of the ECs are input for our RL training loop, as shown in Fig. 4.2. We use these inputs to define our RL environment. Furthermore, we specify some RL parameters, such as the number of steps per episode or the reward function, as input parameters. This enables our test approach to search for star combinations that support robustness testing of the FGS.

In the following sections, we briefly explain how we apply our approach to our case study. We then define our environment and explain how our algorithm works.

4.2 Environment Definition

An environment for DQL consists of three parts: the definition of the state space, the definition of the action space, and the reward function. The state is the actual state of the environment, and the action is what changes the state of the environment. Finally, the reward is what the agent receives in order to update its strategy. To learn test cases that specifically trigger good or bad performance of the SUT we model our environment as follows. Our state space represents the

search space that contains every test case of the given test data. In the context of our case study a state represents a star combination. Our action space contains all actions from which the agent can choose to navigate through the search space and modify a test case. At last our environment uses the reward function to calculate the reward based on the result of the SUT. This reward is the feedback our agent gains to learn how to improve the test case by choosing actions.

In the next sections, we explain our environment definitions in more detail using our case study as example.

4.2.1 State Space

In our approach, the state space depends on the given star pool. One state is the combination of stars from the star pool and represents one test case. We describe each star by its star ID, star pool index, polar coordinates on the FPA, magnitude and sub-pixel position. Radius and angle on the FPA, magnitude and sub-pixel position are the features of a star. Each of these features is assigned to an EC based on the EC definitions. Due to the restrictions from [WGH21], a combination consists of exactly 30 stars without duplicates, i.e. the size of the combination is always $s_c = 30$. These assumptions significantly reduce the state space and thus accelerate training. For example, if we have a star pool with 192 stars, i.e. the size of the star pool is $s_{sp} = 192$, the state space now has the following size:

$$\frac{s_{sp}!}{(s_{sp} - s_c)!} = \frac{192!}{(192 - 30)!} \approx 2.887 \cdot 10^{67} \tag{4.1}$$

To further reduce the size of the state space and speed up the training, we consider any combination consisting of exactly the same stars, regardless of their order, to be the same combination. This is possible because the FGS is resistant to changes in order and outputs the same quaternion if the combination contains the same stars. We achieve this by arranging the stars within the combination according to their star pool indices and then assigning them additional local IDs. Since the size of a combination is $s_c = 30$, the local IDs ranges from 0 to 29. This reduces the size of the state space to:

$$\binom{s_{sp}}{s_c} = \binom{192}{30} = \frac{192!}{(192 - 30)! \cdot 30!} \approx 1.088 \cdot 10^{35} \tag{4.2}$$

This means our state is a structure of stars sorted according to the star pool index. Each star is assigned to a new local index based on its position in the combination. In Fig. 4.3, we illustrates our state definition. A row in the figure represents a star structure with local index, pool index

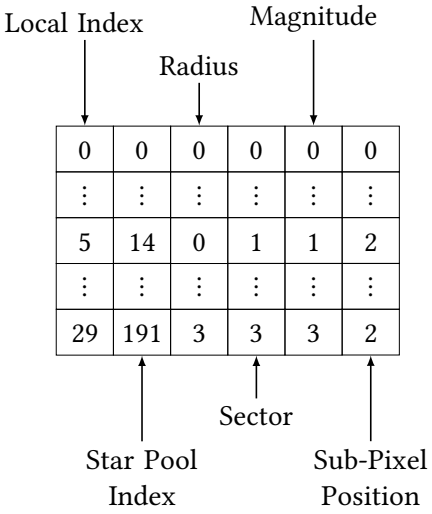


Figure 4.3: State example

and the ID for each EC the star is allocated to. Because the star ID only serves to identify the star regardless of its order within the star pool and is not relevant to the algorithm, we omit it from our representations. What our agent receives from the environment is merely an observation of the state with the information that our agent needs. Since only the EC combinations of the stars are relevant for our algorithm, the observation only consists of this data.

To be more flexible when multiple stars of the star pool are allocated to the same EC combination, we leave the choice of whether an EC combination should be unique in a state or not. Nevertheless, duplicates of stars are not permitted there either.

4.2.2 Action Space

In contrast to the state space, the action space we have defined is independent of the input. It would have been possible to define an action as exchanging a star from the current state with a star from the star pool that is not present in the current state. Assuming that we have exactly one star per EC combination and a star pool of 192, the number of possible actions in a state is then:

$$s_c \cdot (s_{sp} - s_c) = 30 \cdot (192 - 30) = 4860 \tag{4.3}$$

And the size of the complete action space would results in:

$$s_{sp} \cdot (s_{sp} - 1) = 192 \cdot 191 = 36672 \tag{4.4}$$

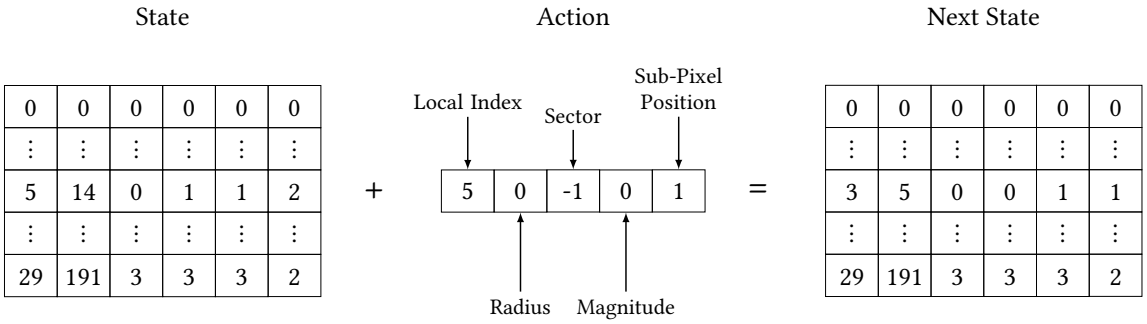


Figure 4.4: Step example

In this case, we have quite a large area of action, which makes learning difficult for the agent. In order to reduce the action space, we define an action as an exchange of two stars, but to limit the radius of action as follows. We define an action as consisting of two parts: The first part is the local index of the star within the current state that is to be replaced. The second part is a modification of the EC combination, defined as Δ. For the FPA position class, we consider the radius and the sector separately. The entire action is defined as a tuple with $(i, \Delta_R, \Delta_S, \Delta_M, \Delta_{SP})$, where i is the local index, Δ_R is the change of the radius EC ID, Δ_S is the change of the sector EC ID, Δ_{SP} is the change of the sub-pixel position EC ID and Δ_M is the change of the magnitude EC ID. Since the size of a star combination is always $s_c = 30$, $0 \leq i < 30$ applies. The Δ of the radius EC ID, the sector EC ID and the magnitude EC ID are relative changes: $\Delta_R, \Delta_S, \Delta_M \in \{-1, 0, 1\}$. While the change of the sub-pixel position EC ID is defined as absolute with $\Delta_{SP} \in \{0, 1, 2\}$. The reason for the different definitions is that the number of ECs of the FPA position and magnitude is variable, while the number of ECs of the sub-pixel position is always three. This means that the number of deltas $n_d = 4$ and there are three possible values for each delta, i.e. $p_d = 3$. Based on our previous definitions, the number of possible actions in a state is:

$$s_c \cdot p_d^{n_d} = 30 \cdot 3^4 = 30 \cdot 81 = 2430 \tag{4.5}$$

This is the size of our entire action space. The reason for this is that the definition of actions depends only on the size of the state s_c and not on the stars within a state. This is because the range of i corresponds to the combination size s_c and the first part of our action is only defined via the local index.

A step of the environment, shown in Fig. 4.4, means that the star with the local index i is replaced from the current state by a new star. To select a new star, the algorithm adds Δ_R, Δ_S and Δ_M to the corresponding EC IDs of EC combination of the star to be replaced. The algorithm directly exchanges the EC ID of the sub-pixel position of the star to be replaced with Δ_{SP}. Based on the resulting EC combination, the agent takes a star allocated to this combination and inserts it into the

next state. In doing so it is possible that the new star is invalid, there may be various reasons for this. Firstly, the star is already present in the combination. Secondly, the permissible number of stars per EC combination in a state has already been reached in this state. Thirdly, the EC combination is not represented in the star pool, as at least one calculated EC ID lies outside the range of the EC definitions. This problem is solved by the environment not allowing any change to the current state and returning the same state and reward as in the previous step.

4.2.3 Reward Function

The definition of the reward function is an important part of every RL approach. A meaningful definition guides the agent in the right direction to achieve the goal. Otherwise, the agent is unable to learn a stabilised strategy to achieve this goal. For this reason, we have tested various reward functions. In our approach our agent receives feedback after each step and not only at the end of each episode. This is so that the agent collects feedback more frequently, rather than just once per episode. It is also because our goal is not to optimise a path in an episode with a total reward, but to find optima during the episode. Each of the reward functions is based on the fitness function Eq. (2.1) proposed by Witteck et al. in [Wit+25]. To do this, our environment first passes the polar coordinates on the FPA, the magnitudes and the sub-pixel positions of the selected stars to the FGS, based on the current state. We use the result of the FGS to then calculate what we call the absolute reward using the fitness function. A higher absolute reward value means that the respective input is more likely to provoke mission-critical behaviour. Since our goal is to find the inputs with the highest values, i.e. the worst case scenario, our algorithm always attempts to maximise the value and our reward definitions are aimed at this goal.

We developed three variants of the reward function for our approach:

1. absolute reward which is the result we receive from the fitness function
2. difference reward, which represents the difference between the absolute rewards of the current and previous state
3. hybrid reward, which is a combination of the absolute reward and the difference reward

The hybrid reward weights the difference reward with a factor α and the absolute reward with a factor β. Previously, the hybrid reward function normalises both rewards by dividing them by their average:

$$\text{reward}_{\text{hybrid\_worst}} = \alpha \cdot \frac{\text{fit}(\text{FGS}(s_t)) - \text{fit}(\text{FGS}(s_{t-1}))}{\text{avg}(\text{reward}_{\text{difference\_worst}})} + \beta \cdot \frac{\text{fit}(\text{FGS}(s_t))}{\text{avg}(\text{reward}_{\text{absolute\_worst}})}. \tag{4.6}$$

We use the absolute reward because it represents the real value we want to maximise. The difference reward has the advantage of providing a relative value that reflects whether the test case has improved or deteriorated. The problem with this variant is that the agent optimises the difference rather than the actual fit. For this reason, we have developed our third reward function, the hybrid reward, which takes both other rewards into account.

To find the best cases, which are combinations with a low absolute reward, i.e. low FGS noise, we reverse our reward definitions. Therefore, we change the sign of the absolute reward and calculate the difference reward by subtracting the absolute reward of the previous state from the absolute reward of the current state. The hybrid reward is then:

$$\text{reward}_{\text{hybrid\_best}} = \alpha \cdot \frac{\text{fit}(\text{FGS}(s_{t-1})) - \text{fit}(\text{FGS}(s_t))}{|\text{avg}(\text{reward}_{\text{difference\_best}})|} - \beta \cdot \frac{\text{fit}(\text{FGS}(s_t))}{|\text{avg}(\text{reward}_{\text{absolute\_best}})|}. \tag{4.7}$$

4.3 DQL Algorithm

In an initial attempt, we used Q-learning, as it is an easy-to-implement algorithm that is relatively transparent in its decision-making. The problem with this is that the Q-table of our problem became too large to find a solution in acceptably time. The main factors for this are the large state and action space. Taking into account our definitions for the state space and the action space and their sizes calculated in Eq. (4.2) and Eq. (4.5), the Q-table had the following size:

$$\binom{s_{sp}}{s_c} \cdot (s_c \cdot p_d^{n_d}) \approx (1.088 \cdot 10^{35}) \cdot 2430 \approx 2.644 \cdot 10^{41} \tag{4.8}$$

This is the result of multiplying the size of our state space and the size of our action space, since the Q-table stores a value for each state-action pair. Since the biggest problem with Q-learning is that the agent does not learn to generalise across states and actions, we decided to use DQL because it is closest to Q-learning and better handles large state and action spaces.

We propose two variants of a DQL approach. The difference between the two variants lies in the Q-value calculator The first variant only considers node embeddings, while the second also considers the complete state embedding. Therefore we call the first variant Q-value calculator with node embeddings and the second variant Q-value calculator with state embedding.

In the following sections, we will first introduce our DQL training loop and then explain the two Q-value calculators in more detail.

4.3.1 DQL Training Loop

Algorithm 1 DQL Training Loop

Input: ϵ, $\epsilon_{initial}$, ϵ_{final}, ϵ_{decay_start}, ϵ_{decay_end}, episodes, episode_size, optimisation_start, batch_size, optimisation_frequency
Output: The state with the highest fit

```
replay_buffer ← []
highest_fit ← 0
highest_state ← []
steps_total ← 0
for episode ← 0 To (episodes - 1) do
    state, reward, fit ← Environment.reset()
    if fit > highest_fit then
        highest_fit ← fit
        highest_state ← state
    end if
    for step ← 0 To (episode_size - 1) do
        action ← Agent.select_action(state, ε)
        next_state, reward, fit ← Environment.step(action)
        if fit > highest_fit then
            highest_fit ← fit
            highest_state ← next_state
        end if
        replay_buffer.append((state, action, reward, next_state))
        if (step ≥ optimisation_start) & (step % optimisation_frequency == 0) then
            Agent.optimise_policy(replay_buffer, batch_size)
            Target_QV_Calculator.update(QV_Calculator, τ)
        end if
        state ← next_state
        if (steps_total ≥ ε_decay_start) & (steps_total ≤ ε_decay_end) then
            ε ← decrease_epsilon(ε_initial, ε_final, ε)
        end if
        steps_total ← steps_total + 1
    end for
end for
return highest_state
```

Since both variants use DQL, the main method of both is a training loop that combines the environment, agent, Q-value calculator, target Q-value calculator and replay. The training loop for the worst case search is shown in Algorithm 1. The length of an episode is always the same, namely the predefined number of steps, as it is not our goal to optimise the path to a specific goal or final state, but to find an optimal state. At the start of each episode, the environment resets the

Algorithm 2 ϵ-greedy Action_Selection

Input: ϵ, state
Output: An action, either randomly selected or the best one

```
1: r ← Random.generate_number([0,1))
2: if r < ε then
3:     action ← Environment.Action_Space.sample()
4:     return action
5: else
6:     q_values ← QV_Calculator.calculate_q_values(state, all_actions)
7:     best_action ← select_best_action(q_values, all_actions)
8:     return best_action
9: end if
```

state to a random state in the state space (line 6, Algorithm 1) in order to increase and vary the explored space. This also reduces the probability that our agent remains stuck in the same local optimum throughout the entire execution. In each step the agent first selects an action based on the current state and the current ϵ (line 12, Algorithm 1). For the action selection we use an ϵ-greedy algorithm, as shown in Algorithm 2, since it is an easy-to-implement algorithm that is frequently used in RL. If a randomly generated number r with $0 \leq r \leq 1$ is less than the current ϵ, then the environment randomly returns an action from the action space, regardless of the current state (line 3, Algorithm 2). Otherwise, the Q-value calculator uses the policy to determine the Q-values for all actions for the current state (line 6, Algorithm 2). The algorithm then selects the action with the highest Q-value. The ϵ-greedy algorithm thus gives us the opportunity to influence the probability of exploration and exploitation. We always start an execution with $\epsilon_{initial}$ equal to 1, so that the agent always explores in the first steps, as it has no knowledge yet. Over time, we reduce ϵ until it equals ϵ_{final} in order to reduce the probability of exploration and increase the probability of exploitation (line 25, Algorithm 1). We reduce ϵ linearly each step, starting from a predefined total number of steps, until the algorithm reaches another predefined total number of steps. Therefore, we count the total number of steps (line 27, Algorithm 1).

Next, our previously defined environment tries to execute the action returned by the ϵ-greedy algorithm, taking into account the predefined conditions of a valid combination. Then, the environment returns the next state, the reward calculated based on the reward function and the fit calculated by the fitness function (line 13, Algorithm 1). The fit is the FGS noise. If this fit is higher than the highest fit so far, it saves the fit as the new highest fit and the returned next state as the best state so far (line 14, Algorithm 1). This query is also performed immediately after the reset, in case the randomly selected initial state has the highest fit so far (line 7, Algorithm 1). After storing the highest fit and corresponding state, the replay buffer stores the combination of state, action performed, reward and next state as a tuple (line 18, Algorithm 1). When the replay buffer reaches its predefined size, the oldest entry is deleted.

Then, if a predefined number of steps has been reached and at a certain frequency, the agent first optimises its policy on a randomly selected batch from the replay buffer with a predefined batch size (line 20, Algorithm 1). Thereby the Q-value calculator can calculate better Q-values in the next iterations. The parameters of the target Q-value calculator are then updated by the parameters of the Q-value calculator (line 21, Algorithm 1). This serves to stabilise the training of the Q-value calculator. We describe the process in more detail in the next section, see Section 4.3.2.

Finally, the loop starts again with the next state as the state and a possibly reduced ϵ.

At the end of execution, the algorithm returns the combination stored as the *highest state*, i.e. the state with the highest reward, which represents the worst case found by the algorithm. Depending on the test objective, it is also possible to search for best cases. The algorithm then saves the lowest fit and the corresponding state.

4.3.2 Q-Value Calculator

The core of our approach is the Q-value calculator, which calculates the Q-values in order to select an action. We present two variants, which we refer to as the Q-value calculator with node embeddings and the Q-value calculator with state embedding. In both variants we use Leaky ReLU as activation function in each network so that negative values do not simply disappear. In the Q-value calculator with node embeddings, shown in Fig. 4.5, we initially use a GCN, that we call state encoder, to calculate node embeddings from our state. We choose a GCN because our input is a combination of stars and we want to use their neighbourhood as information. With GCNs, we can control how we define the neighbourhood for the flow of information. However, to use the state encoder, we must first convert our observation of the state into a graph. We define the graph as follows. Each star in the state is a node with the features FPA radius, FPA sector, magnitude and sub-pixel position. This means that the size of our graph always corresponds to the size of our state, i.e. 30. The edges are all undirected and defined by the neighbourhood definition. In our case, we define two nodes n_1 and n_2 as neighbours if they are allocated at least to the same EC for the radius or for the sector. If two nodes n_1 and n_2 are neighbours an undirected edge $e(n_1, n_2)$ exists. Additionally, each node has an edge to itself. The output of the state encoder are node embeddings. All possible deltas from the action space definition and the node embeddings are then the input for the next network, which we refer to as the action encoder. The action encoder has two parts. In the first part, we calculate the embeddings for all deltas. We calculate them separately for $\Delta_R, \Delta_S, \Delta_M$ and Δ_{SP}. In the second part, we combine the embeddings for $\Delta_R, \Delta_S, \Delta_M$ and Δ_{SP} with the corresponding delta embeddings like in [WLL24]. This allows us to generalise the actions to a greater extent. This means that we have a combined embedding for an action, whereby each action is already dependent on the state due to the node embeddings. The combination is then fed

into a MLP and converted into the input dimension required for the final network. We refer to this as state-action-embeddings, as shown in Fig. 4.5. In a last step an MLP, that we call Q-net, returns a calculated Q-value for each state-action-embedding.

Our second variant, the Q-value calculator with state embedding, shown in Fig. 4.6, differs from our Q-value calculator with node embeddings in the final calculation of the Q-values. The state encoder additionally outputs a state embedding, which is an aggregation of the node embeddings. Then, an MLP, which we refer to as state projector, calculates a representation of the state embedding in the same dimension as the state-action embeddings. In a final step, we calculate the dot product of the state projection and the state-action embeddings. The result of the dot product is the Q-value for each combination. In this variant, the Q-value is therefore higher the more similar the embeddings are.

The advantage of the second variant is that we also take the entire state embedding into account and not just the node embeddings. The problem of the Q-value calculator with state embedding could be that we are using a kind of redundant information, since the node embeddings already contain information about the state.

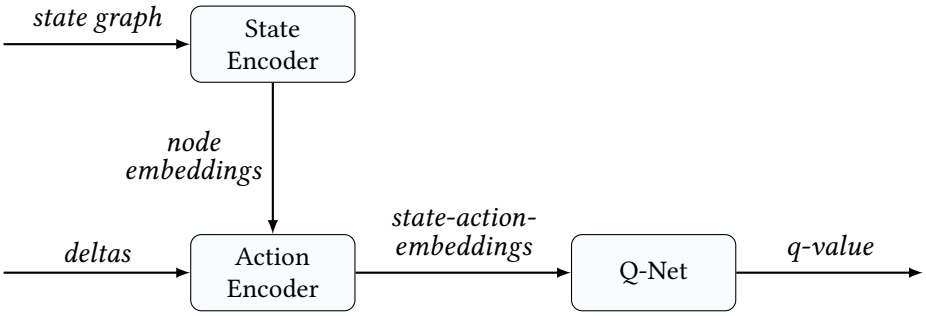


Figure 4.5: Q-Value Calculator with node embeddings

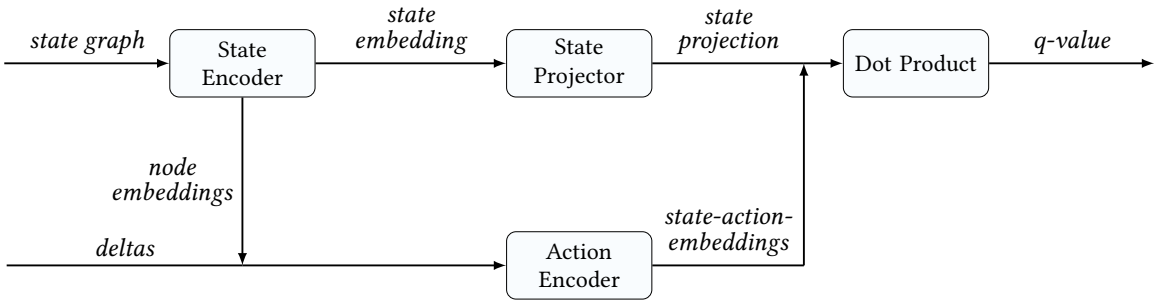


Figure 4.6: Q-Value Calculator with state embedding

The target Q-value calculator corresponds to the form of the Q-value calculator and is only used during replay or policy optimisation, see Algorithm 3. In this section, we consider our inputs to the

Algorithm 3 Policy Optimisation

Input: Replay_Buffer, batch_size

1: batch ← Replay_Buffer.sample(batch_size)
2: predicted_q_values ← QV_calculator.calculate_q_values(batch.state, batch.action)
3: next_q_values ← target_QV_calculator.calculate_q_values(batch.next_state, all_actions)
4: max_next_q_values ← max(next_q_values)
5: Target_q_values ← batch.reward + $\gamma \cdot$ max_next_q_values
6: loss ← Loss(predicted_q_values, target_q_values)
7: Backpropagation(QV_calculator, loss)

networks as batches. This means that when we simply write batch.state as input, we have actually entered a complete batch of states. At the beginning of the policy optimisation, we take a random sample, called a batch, of the tuples previously stored in the replay buffer (line 1, Algorithm 3). The Q-value calculator then calculates the predicted Q-values based on the current policy for the states and actions stored in the batch (line 2, Algorithm 3). The target Q-value calculator, on the other hand, calculates the Q-values for all actions of the next state (line 3, Algorithm 3). Then we calculate the target Q-values using the highest Q-value per batch element (line 5, Algorithm 3). We use the equation from Eq. (2.3) for this.

Next we calculate the loss on the deviation of the predicted Q-value from the target Q-value (line 6, Algorithm 3). In the final step, we use the loss to optimise the Q-value calculator through backpropagation (line 7, Algorithm 3). In this way, the parameters of the Q-value calculator are adjusted to minimise the feature loss. In contrast, the target Q-value calculator is not updated by backpropagation, but in our case by a soft update depending on its current parameters and the parameters of the Q-value calculator with the factor τ, see line 21 in Algorithm 1. By using a soft update, the target Q-value calculator does not change the target as quickly, which also makes learning more stable than with a hard update.

In this chapter, we have presented our approach for the automated generation of test cases for satellite on-board image processing algorithms, which applies RL in this field. In the following chapter, we present the conditions under which we evaluate our approach, present our results, and compare them with the results of the GA approach from Witteck et al.

5 Evaluation

In this chapter, we introduce the libraries we use and explain why we use them. We also describe the structure and number of layers for each network in our implementation. In the next step, we present the results of our experiments. Finally, we compare the results of our approach to the results of the GA from Witteck et al. in [WGH21].

5.1 Implementation

The implementation of our approach takes place in Python (version 3.10) under Linux. Since the FGS is written in C++ we also implemented a wrapper using pybind11 (version 2.13.6) to build a library which includes the functions we need to make the FGS executable in Python. The input in form of the star catalogue, camera alignment and target attitude are stored in a hdf5 file, so we use h5py (version 5.15.1) to read them in. For the implementation of our environment we use Gymnasium (version 1.2.2) and for the implementation of our DQL algorithm we use PyTorch (version 2.8.0) and PyTorch Geometric (version 2.7.0). We also use CUDA (version 12.8) to run our algorithm on the GPU, thereby reducing the runtime. We use a NVIDIA GeForce GTX 1650 with Max-Q Design. By using CUDA, our algorithm takes about 2.5 h for 10^6 steps. Our entire implementation comprises approximately 1,400 lines of code.

We implement the state encoder of the Q-value calculators as GCN with two graph convolutional layers and alternating two layers for normalisation. In the implementation of the Q-value calculator with state embedding, we additionally use global mean pooling after the network to aggregate the node embeddings to the state embedding. The action encoder of both variants consists of four PyTorch embeddings for the deltas followed by one layer for normalisation and then two linear layers for the entire action. The implementation of the Q-network of the Q-value calculator with node embeddings is a simple MLP with two layers. The same applies to the state projector of the Q-value calculator with state embedding.

Parameter	Catalogue 0	Catalogue 1	Catalogue 2
Star pool size	576	288	192
Radius EC number	8	4	4
Sector EC number	4	4	4
Magnitude EC number	6	6	4
Sub-pixel position EC number	3	3	3

Table 5.1: Test data configurations

	Number of samples	Catalogue 0	Catalogue 1	Catalogue 2
Best case	1×10^6	4.207	4.119	4.191
Worst case	1×10^6	6.754	6.413	6.374
Best case	2×10^6	-	-	4.207
Worst case	2×10^6	-	-	6.416

Table 5.2: FGS noise [mas] for random generation

5.2 Experimental Results

In our experiments, we use three different sets of test data: catalogue 0, catalogue 1, and catalogue 2. All three use the same camera alignment and target attitude. They differ in terms of the star pool and EC definitions. The size of the star pool and the number of ECs per star parameter are listed in Table 5.1. Each star catalogue covers every EC combination with exactly one star.

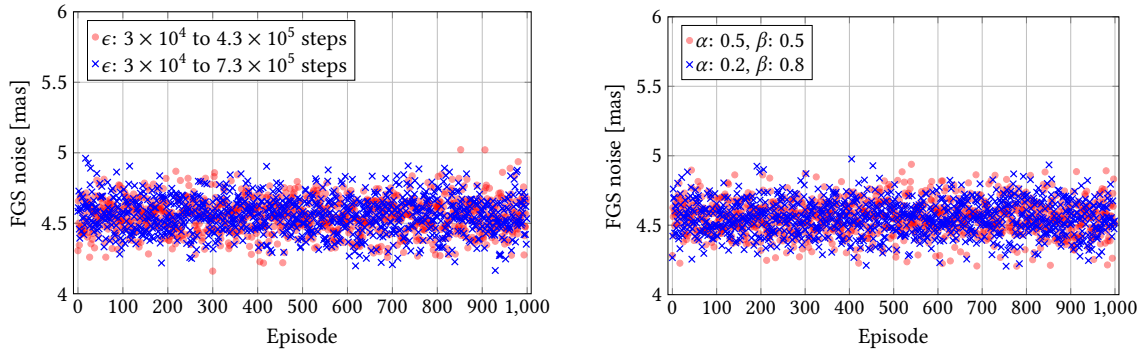
First of all, we ran our algorithm with an episode length of zero, which corresponds to a random algorithm without learning. We did this to establish a baseline for later comparison. We ran this random algorithm for each catalogue for 1,000,000 episodes. Due to time constraints and because it is the smallest catalogue with a smaller search space, we will focus more on catalogue 2 later on. For this reason, we also executed the algorithm with 2,000,000 episodes. All results are shown in Table 5.2.

We list the parameter values for our standard configuration that we use as input for our RL algorithm execution in Table 5.3. Unless otherwise specified, we run the algorithm for 1,000 episodes with an episode length of 1,000. In the first experiments, we test the influence of the epsilon decay period, different values for α and β for the hybrid reward function, our three different reward functions, and both presented DQL variants, the Q-value calculator with node embeddings and the Q-value calculator with state embedding. We are conducting these initial experiments for the star catalogue 0, as this catalogue has the largest search space and the other two catalogues represent smaller versions of the same problem.

Parameter	Value
Star combination size	30
Learning rate	1×10^{-4}
Initial ϵ	1.0
Final ϵ	0.1
Batch size	64
Optimization frequency	4
Replay start	2×10^4
Only different ECs	True
Seed	False
γ	0.99
τ	1×10^{-3}
Replay buffer size	1×10^5
Episodes	1000
Episode length	1000

Table 5.3: Standard configuration of the RL algorithm

In our first two experiments, the we use the first variant of our DQL algorithm, the Q-value calculator with node embeddings, to search for the best case. The diagrams in Fig. 5.1 present the minimums of the FGS noise found by the algorithm in each episode using the absolute reward function. Fig. 5.1a shows the results when we reduce ϵ linear from the initial ϵ to the final ϵ over different time periods. In order to compare the influence of reducing ϵ over a shorter period of time and a longer period of time, and thus the influence of the length of the exploration phase, we reduce ϵ linearly in the range of 3×10^4 to 4.3×10^5 steps for the first run of our experiment. In a second run, we decrease ϵ in the range from 3×10^4 to 7.3×10^5 steps. When evaluating the results, we focus on the results of the later episodes, since then ϵ is equal to 0.1 and our agent is more likely to select its actions based on the policy rather than acting randomly. As the figure shows, from around 610 episodes onwards, the algorithm finds a lower FGS noise more frequently in the second run than in the first run. For this reason, unless otherwise specified, we use the period of 3×10^4 to 7.3×10^5 steps for the ϵ decay in the following experiments. Fig. 5.1b presents the results for the hybrid reward function for different α and β: once for α and β equal to 0.5 and once for α equal to 0.2 and β equal to 0.8. As shown in Eq. (4.6) and Eq. (4.7) we use a reference value in our hybrid reward function to combine the difference and absolute reward value. To obtain such a reference value for the difference reward part of the hybrid reward, we calculate the average of the results of a previous execution of our RL algorithm, in which we used the difference reward. For that previous execution we use the same parameters as for the execution using the hybrid reward function. In the same way we obtain the reference value for the absolute reward by using the absolute reward function. The average we use for the experiment when searching for the best case to normalise the result of the difference reward function is approximately 0.041, and the average



(a) Results of different time periods of ϵ decay

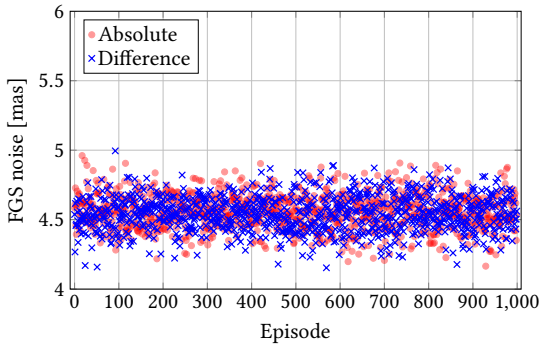
(b) Results of hybrid reward function with different weights

Figure 5.1: Results of different settings for the time period of ϵ decay and the hybrid reward function for best case search

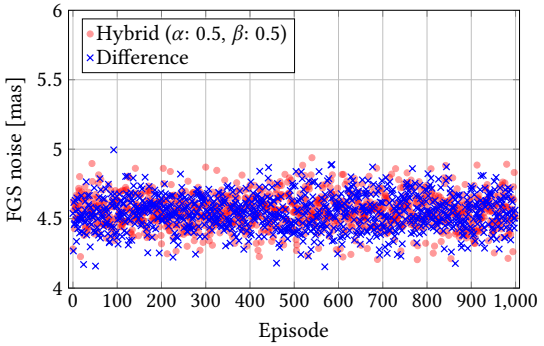
we use to normalise the absolute reward function is approximately 5.125. As the figure shows, from around 500 episodes onwards, our algorithm outputs slightly more frequent lower FGS noise for the first option than for the second option. Therefore, we use the first option with α and β equal to 0.5 to compare the results of the hybrid reward function with the results of the other two reward functions in our next experiments.

As next we ran our algorithm with the absolute reward function, the difference reward function, and the hybrid reward function. The results of the best case and the worst case search are shown in Fig. 5.2. Fig. 5.2a depicts the results of the search for the best case for the absolute reward and the difference reward. The figure shows that when searching for the best case from around 500 episodes onwards, both reward functions deliver similar results for the FGS noise. Fig. 5.2c shows, that the global maximum provided by the difference reward function at episode 907 with an FGS noise of 7.317 mas is higher than the global maximum provided by the absolute reward function. However, the diagram depicts that the absolute reward consistently delivers higher noise values than the difference reward function from around 500 episodes onwards. Since our goal is to find a global optimum, we have decided to focus on the difference reward function for the time being. Therefore, we only compare the results of our RL algorithm using the difference reward function with the results of the algorithm using the hybrid reward function. The results of the best case search are shown in Fig. 5.2b and the results of the worst case search are shown in Fig. 5.2d. As both figures depict, the comparison of the results closely resembles the comparison of the results of the absolute and difference reward functions, which is why we again decided to focus on the difference reward function.

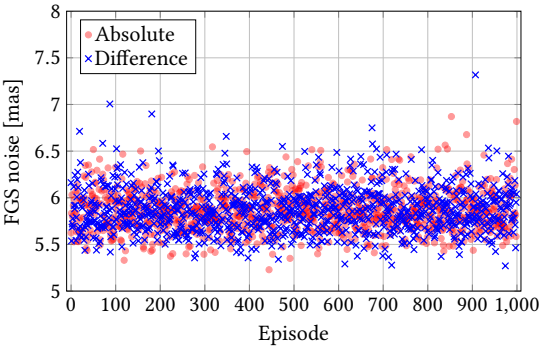
After deciding on the time period in which to reduce ϵ and the reward function to use, we ran both proposed variants to compare their results. Fig. 5.3 shows the results of our experiments for



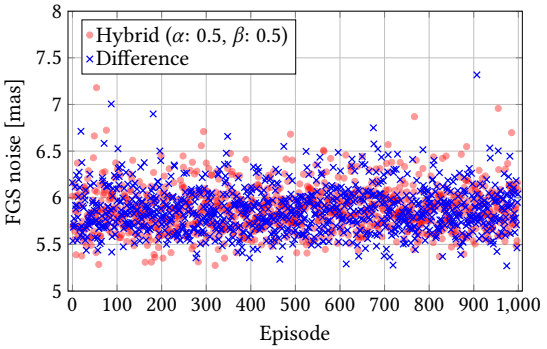
(a) Results of the absolute reward function and the difference reward function for the best case search



(b) Results of the hybrid reward function with α=0.5 and β=0.5 and the difference reward function for the best case search

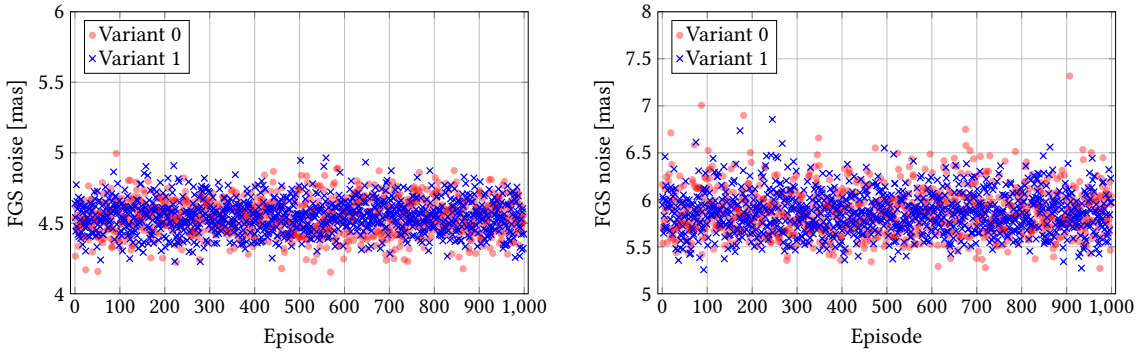


(c) Results of the absolute reward function and the difference reward function for the worst case search



(d) Results of the hybrid reward function with α=0.5 and β=0.5 and the difference reward function for the worst case search

Figure 5.2: Results of the three reward functions in comparison



(a) Results of the best case search

(b) Results of the worst case search

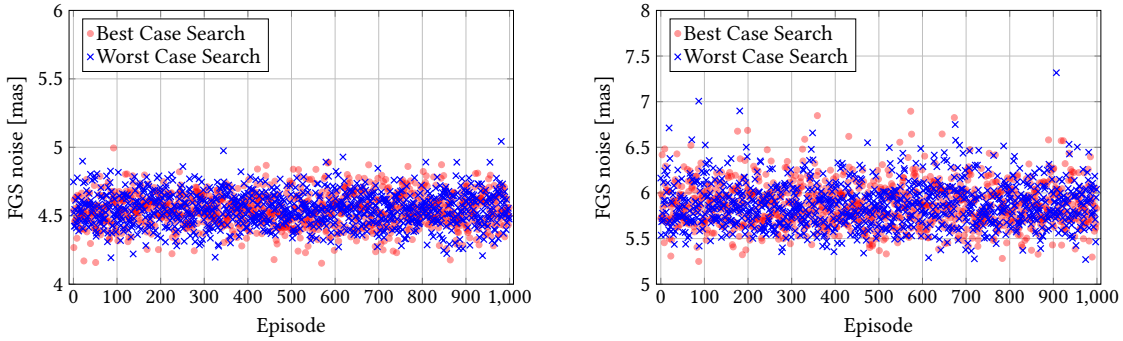
Figure 5.3: Results of the two proposed variants in comparison for the difference reward function

both Q-Value calculator variants. Fig. 5.3a depicts the results of the best case search and Fig. 5.3b the results of the worst case search. The first diagram shows that our algorithm by using the Q-value calculator with node embeddings finds a lower global minimum with 4.153 mas for the FGS noise than our algorithm using the Q-value calculator with state embedding with 4.228 mas. The diagram also shows that the our algorithm using the Q-value calculator with node embeddings consistently finds lower values than our the algorithm using the Q-value calculator with state embedding from around 450 episodes onwards. The worst case search diagram shows that the our algorithm using the Q-value calculator with node embeddings also finds higher FGS noise values and consistently higher FGS noise values in the later episodes from around 600 episodes onwards. Since both diagrams show that Q-value calculator with node embeddings achieves better results than Q-value calculator with state embedding, we will only use Q-value calculator with node embeddings in the following.

In summary, based on the experiments conducted so far, we have made the following decisions and are now using the following setting:

- reducing ϵ from 3×10^4 to 7.3×10^5 steps
- using the difference reward function
- using the Q-value calculator with node embeddings for the Q-value calculation

In the next experiments, we evaluate the results of our standard configuration and the settings from the previous experiments by running our algorithm to search for the best and worst cases for all three star catalogues. The results of star catalogue 0 are shown in Fig. 5.4. The left diagram depicts the minimum FGS noise of each episode provided by the algorithm during the best case search and during the worst case search, see Fig. 5.4a. By comparing these results, we expect that, the best case search will deliver better results than the search for the worst case. Instead, the diagram shows no



(a) Minimum FGS noise [mas] of each episode for best and worst case search

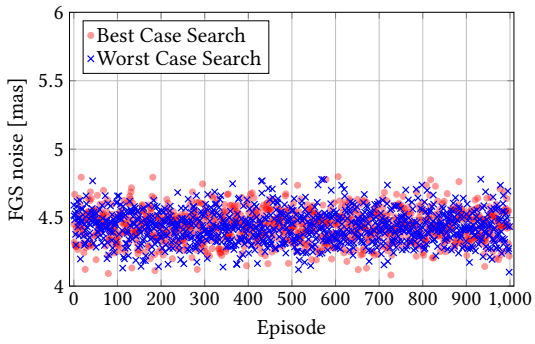
(b) Maximum FGS noise [mas] of each episode for best and worst case search

Figure 5.4: Results of Q-value calculator with node embeddings for star catalogue 0 using the difference reward function

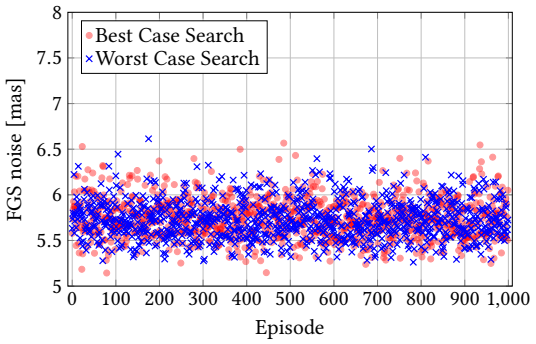
major differences in the minimum FGS noise. The right diagram shows the maximum FGS noise during best and worst case search per episode, see Fig. 5.4b. Here, we expect that searching for the worst case will yield higher values for the maximum FGS noise, but with the exception of one outlier, the results for both searches are quite similar.

We also ran the algorithm for star catalogue 1 and star catalogue 2 to see whether a smaller search space would affect the results. The results of both star catalogues are shown in Fig. 5.5. Fig. 5.5a and Fig. 5.5b show the results of star catalogue 1 for best and worst case search respectively. The results of the best and worst case search using star catalogue 2 are shown in Fig. 5.5c and Fig. 5.5d. In Fig. 5.5, too, the left sides represent the minimum FGS noise for each episode, while the right sides represent the maximum FGS noise for each episode. As all four figures show, the results are not really influenced by the search type. This reinforces the assumption that either the difference reward function is unable to guide the agent in the right direction to optimise the actual FGS noise, or that our agent is unable to learn from its experiences or is unable utilise its knowledge in order to achieve the goal. Therefore, in our next experiment, we will examine the relationship between what the agent actually learns and our goal.

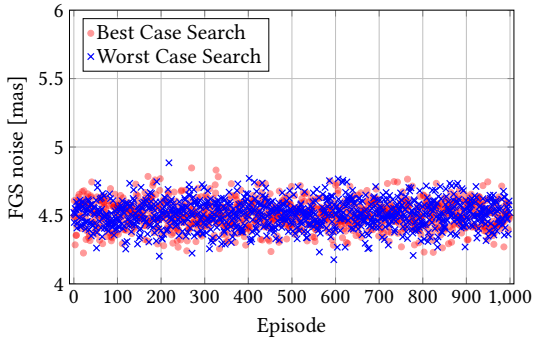
Fig. 5.6 shows the values calculated by the difference reward function and the actual FGS noise for catalogue 0 per episode. Fig. 5.6a shows the minimum FGS noise found by our algorithm using the Q-value calculator with node embeddings and using the difference reward function. The figure also shows the corresponding maximum values calculated by the difference reward function for each episode. Fig. 5.6b shows the maximum FGS noise found by our algorithm using the Q-value calculator with node embeddings and using the difference reward function. The figure also shows the corresponding maximum values calculated by the difference reward function per episode. Both figures illustrate how the calculated reward and the actual FGS noise calculated by the fitness



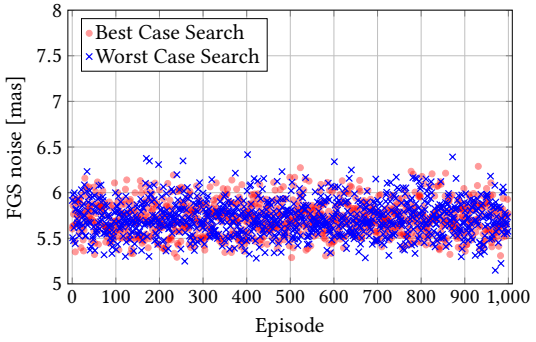
(a) Minimum FGS noise [mas] of each episode for best and worst case search for star catalogue 1



(b) Maximum FGS noise [mas] of each episode for best and worst case search for star catalogue 1

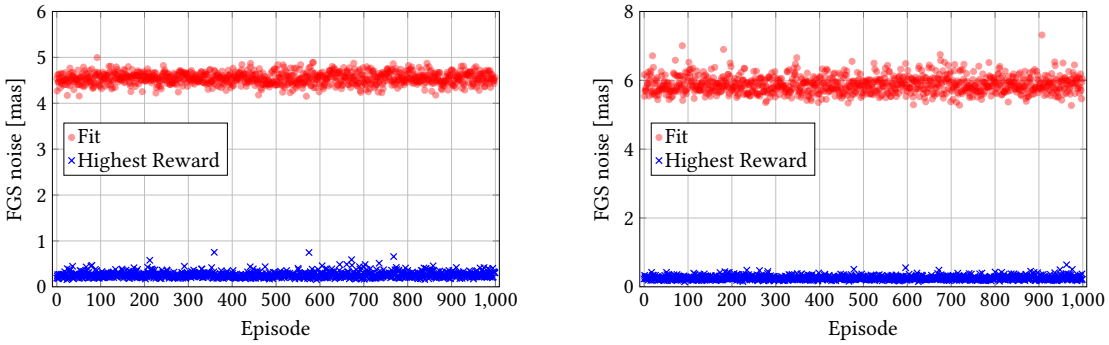


(c) Minimum FGS noise [mas] of each episode for best and worst case search for star catalogue 2



(d) Maximum FGS noise [mas] of each episode for best and worst case search for star catalogue 2

Figure 5.5: Results of Q-value calculator with node embeddings for star catalogue 1 and star catalogue 2 using the difference reward function



(a) Minimum FGS noise [mas] for the difference reward (here maximums) and the actual FGS noise during best case search

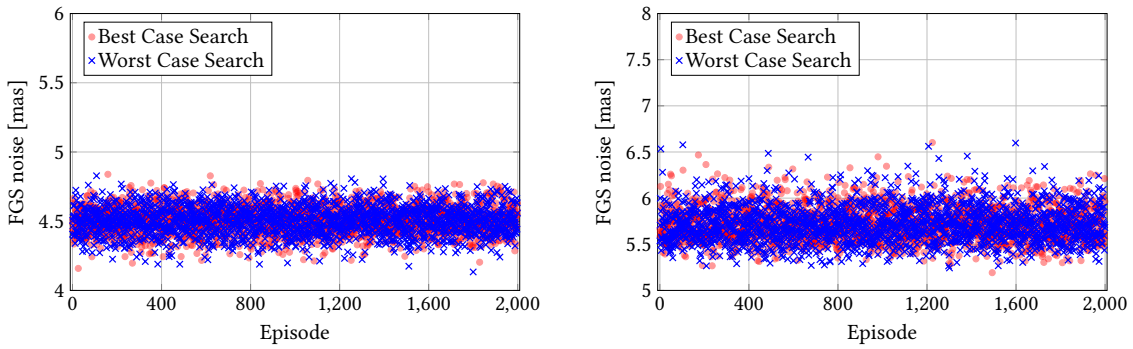
(b) Maximum FGS noise [mas] for the difference reward and the actual FGS noise during worst case search

Figure 5.6: Results of Q-value calculator with node embeddings for star catalogue 0 using the difference reward function as comparison between the calculated reward and the actual FGS noise

function are linked. As the figure on the left shows, the minimum turning points of the actual FGS noise and the maximum turning points of the difference reward are not always related to each other. For example, the differential reward has four clear maximum turning points, while the difference reward has no minimum turning points or distinctive points in each of these episodes. The same can be seen in the figure on the right, where not every maximum turning point in one graph corresponds to a maximum turning point in the other graph. We conclude from this that the difference reward function is not optimal for achieving our goal, as the optimum of the difference reward function does not necessarily correspond to the optimum of the actual FGS noise.

The results from the beginning, when we compared the three reward functions, do not seem to have been meaningful enough. For this reason, we ran our algorithm in the next experiments with the absolute reward function, as this value best reflects our goal. In the following experiments, we vary the number and length of episodes to determine whether our agent is able to find more expressive results when comparing results of the best case and worst case search. Since we want to see whether our agent can actually learn, we use the star catalogue 2 as a test set. We use it because if our agent learns very slowly and needs more time to learn, we can detect this tendency earlier in a smaller search space.

First we double the number of episodes to 2,000 and with that adjust the ϵ decay time to 3×10^5 to 1.46×10^6 steps. The results of that experiment are shown in Fig. 5.7. The left diagram depicts the minimum FGS noise found for each episode during the best and worst case search, see Fig. 5.7a. The right diagram the corresponding maximum values, see Fig. 5.7b. As the figures show, the search for the worst case delivers more minimums but also more maximums than the best case search.



(a) Minimum FGS noise [mas] of each episode for the best and worst case search

(b) Maximum FGS noise [mas] of each episode for the best and worst case search

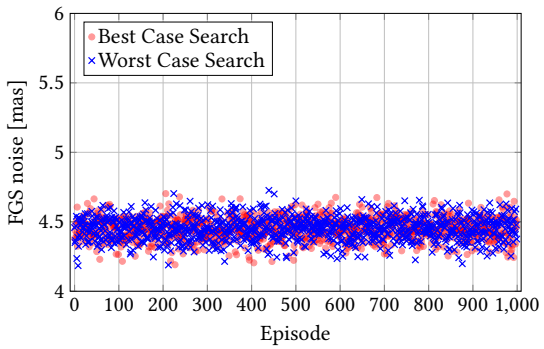
Figure 5.7: Results of Q-value calculator with node embeddings for star catalogue 2 using the absolute reward function, 2,000 as number of episodes and 3×10^5 to 1.46×10^6 steps as ϵ decay time

In the next experiment we use again 1,000 as number of periods but double the length of the episodes to 2,000. This gives the agent more time to explore or optimise a combination for one episode. The results for that experiment are shown in Fig. 5.8. Fig. 5.8a on the left side shows again the minimum FGS noise for each episode during the best and worst case search. Fig. 5.8b on the right side shows the maximum FGS noise for each episode during the best and worst case search. The diagrams show that the algorithm finds similar minimum FGS noise during worst case and best case search. But the algorithm finds significantly higher maximum values during worst case search than during best case search from around 550 episodes onwards. This could be an indication that the longer episodes help the agent to learn.

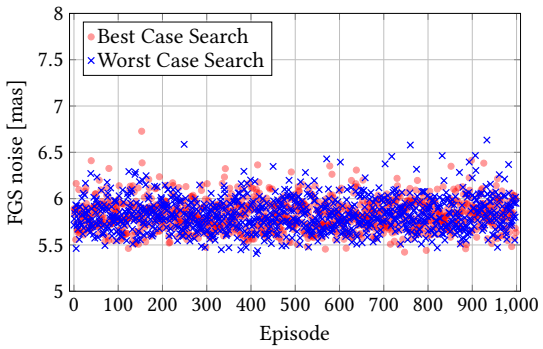
We also increased our learning rate from 10^{-4} to 3×10^{-4} in combination with the higher episode length in the next experiment. The results are shown in Fig. 5.9 and, as before, divided into Fig. 5.9a and Fig. 5.9b. The figures show lower minimum FGS noise for the best case search than in the previous experiment. The maximum FGS noise for the worst case search, on the other hand, are lower than in the previous experiment from around 550 episodes onwards. One possible reason could be that the exploration phase is still too short and there are not yet any good examples from which to learn.

In Table 5.4 we list the best results provided by our algorithm using the difference reward function and the standard configuration for all three star catalogues. In comparison to the results of the random algorithm, see Table 5.2, our algorithm delivers for each star catalogue better results for the lowest and the highest FGS noise except of the lowest FGS noise for star catalogue 2.

The best results of our algorithm using the absolute reward function for catalogue 2 are shown in Table 5.5. When we compare the results of our algorithm, which was run with the standard

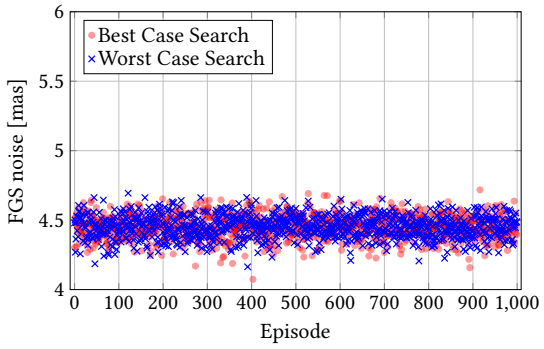


(a) Minimums of each episode found by the best and worst case search

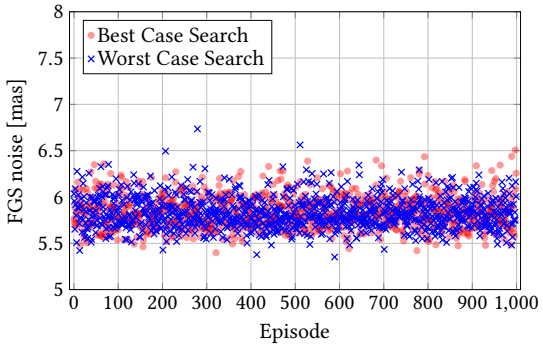


(b) Maximums of each episode found by the best and worst case search

Figure 5.8: Results of Q-value calculator with node embeddings for star catalogue 2 using the absolute reward function and 2,000 as length of episodes and 3×10^5 to 1.46×10^6 steps as ϵ decay time



(a) Minimums of each episode found by the best and worst case search



(b) Maximums of each episode found by the best and worst case search

Figure 5.9: Results of Q-value calculator with node embeddings for star catalogue 2 using the absolute reward function and 2,000 as length of episodes, 3×10^5 to 1.46×10^6 steps as ϵ decay time and 3×10^{-4} as learning rate

	Catalogue 0	Catalogue 1	Catalogue 2
Best case	4.153	4.081	4.226
Worst case	7.317	6.614	6.417

Table 5.4: FGS noise [mas] using standard configuration and distance reward function

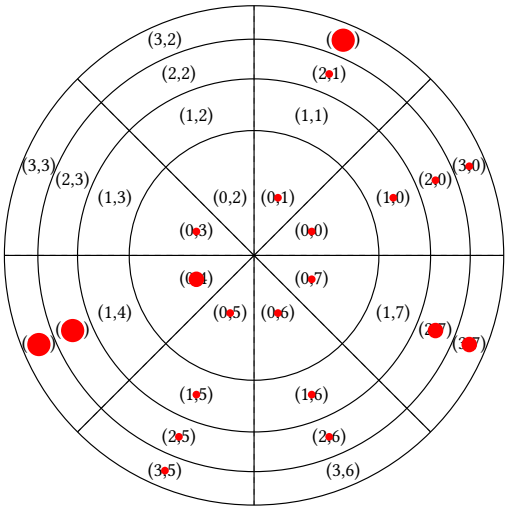
Parameter	Best case	Worst case
Standard configuration	4.165	6.871
2×10^3 episodes	4.159	6.597
2×10^3 steps	4.191	6.632
2×10^3 steps, learning rate 1×10^{-3}	4.074	6.735

Table 5.5: FGS noise [mas] using absolute reward function and catalogue 2

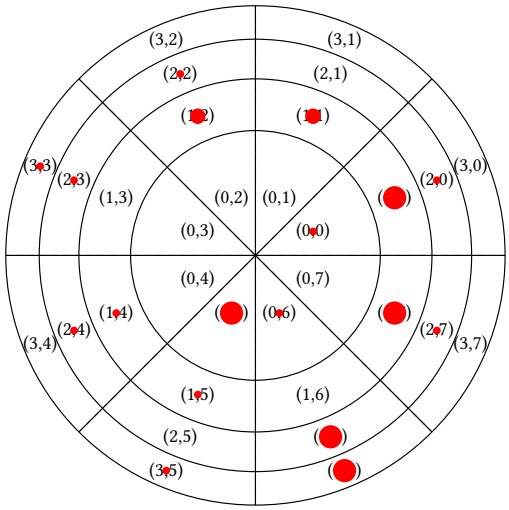
configuration, with the results of the random algorithm with 1×10^6 iterations, our algorithm delivers better results. Even when we compare the other results in this table with the results of the random algorithm from Table 5.2, where the random algorithm was run for 2×10^6 iterations, our algorithm outperforms the random algorithm.

Next we illustrate the star combinations of the best results generated by our algorithm by mapping the stars to their allocated EC on the FPA and show the distribution of stars over the EC of the magnitude. For star catalogue 0 were the best results found by using the difference reward function a FGS noise of 4.153 mas for the best case search and 7.317 mas for the worst case search, see Table 5.4. Fig. 5.10 illustrates for all FPA EC the amount of allocated stars that are contained in the star combination with highest FGS noise. The left figure illustrates the star distribution for the best case search. As Fig. 5.10a shows, the ECs that are covered by the stars in the combination are distributed across the FPA in three accumulations, with the ECs of the outer ring being particularly well represented. Only the ECs located on the CCD at the top left are almost not represented. In contrast to that the distribution of the ECs on the FPA for the worst case search, see Fig. 5.10b, shows more larger dots, which means that some classes have more stars. This means that the stars are closer together for the worst case. Especially the ECs on the CCD at the bottom right is well covered. For the best case, eleven ECs are not represented and for the worst case, fourteen ECs are not represented. Fig. 5.10c and Fig. 5.10d show the distribution of the stars in the star combination over the ECs of the magnitude in the star combinations. As the figures show, in the star combination for the best case are the highest magnitude EC IDs more represented, while in the star combination for the worst case the lower EC IDs are a little more represented. This means that stars with a lower magnitude occur more frequently in the star combination for the best case than in the one for the worst case. The combination for the worst case instead contains more stars with a higher magnitude than the combination for the best case.

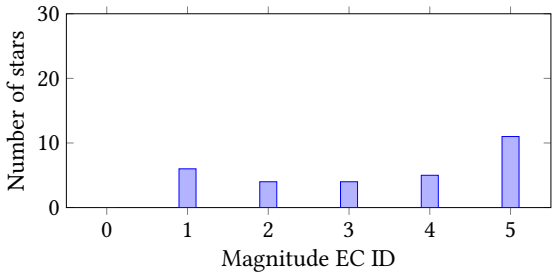
The star combinations from the best case search for star catalogue 1 are shown in Fig. 5.11. On the left side are the figures for the star combination with 4.081 mas, Fig. 5.11a and Fig. 5.11c. On the right are the illustrations of the ECs covered by the stars of the star combination that provokes an FGS noise of 6.614 mas. The ECs covered by stars of the combination for the lowest FGS noise are again more widely distributed on the FPA than the ECs covered by stars of the combination for



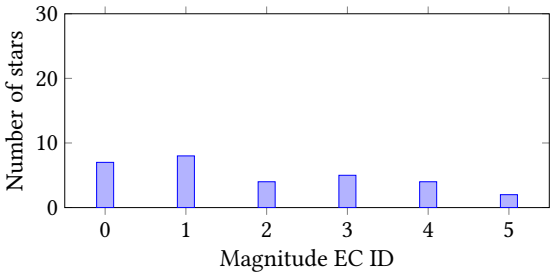
(a) Coverage of the FPA ECs for the best star combination for best case search



(b) Coverage of the FPA ECs for the best star combination for worst case search



(c) Coverage of the magnitude ECs for the best star combination for best case search

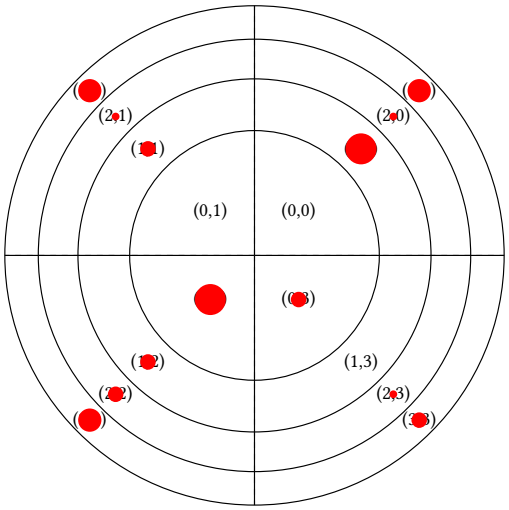


(d) Coverage of the magnitude ECs for the best star combination for worst case search

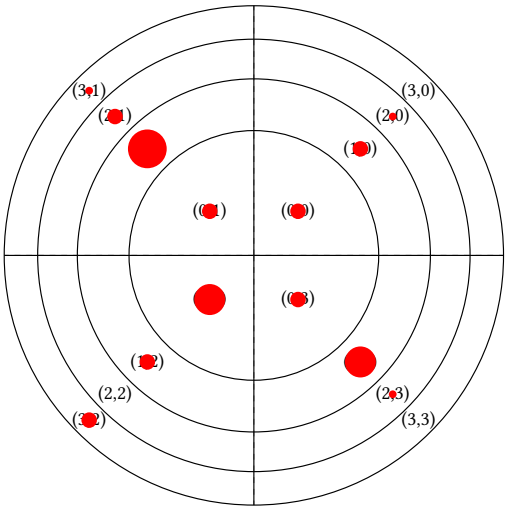
Figure 5.10: Distribution of stars of the best star combination over FPA ECs and magnitude ECs for the best case and the worst case found by our algorithm using Q-value calculator with node embeddings and using the difference reward function and the standard configuration for star catalogue 0

the highest FGS noise. The same trend towards lower star magnitudes for a lower FGS noise and towards higher magnitudes for a higher FGS noise can also be seen again in the distribution of stars over the ECs of the magnitude.

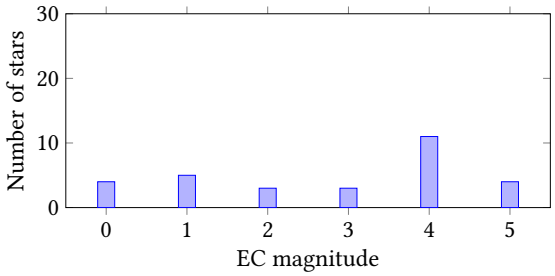
At last we show the results of the star catalogue 2 in Fig. 5.12. This time the results are from Q-value calculator with node embeddings using the absolute reward function and for the best case search the best value is reached with the episode length equal 2,000 and the learning rate equal 3×10^{-4}, see Table 5.5. The results for the lowest FGS noise with 4.074 mas are shown in Fig. 5.12a and in Fig. 5.12c. The highest FGS noise is 6.871 mas for star catalogue 2 and the EC coverage of that star combination are illustrated by Fig. 5.12b and Fig. 5.12d. That highest FGS noise was found with the standard configuration, see Table 5.5. The distributions of stars over the ECs for catalogue 2 show



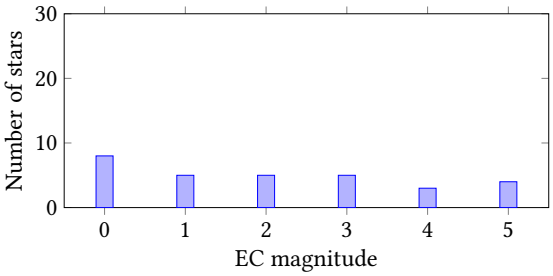
(a) Coverage of the FPA ECs for the best star combination for best case search



(b) Coverage of the FPA ECs for the best star combination for worst case search



(c) Coverage of the magnitude ECs for the best star combination for best case search



(d) Coverage of the magnitude ECs for the best star combination for worst case search

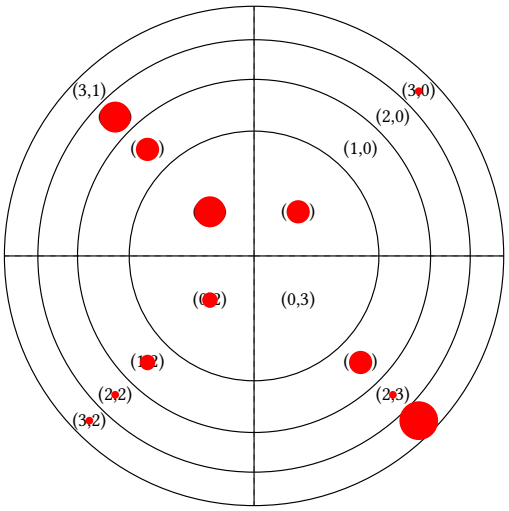
Figure 5.11: Distribution of stars of the best star combination over FPA ECs and magnitude ECs for the best case and the worst case found by our algorithm using Q-value calculator with node embeddings and using the difference reward function and the standard configuration for star catalogue 1

the same trends as for the other two star catalogues. If we compare the star combinations from catalogue 0 and catalogue 2 for the worst case, the results are reflected in the figures. The result of catalogue 0 for the worst case is 6.614 mas which is lower than the result for catalogue 2 with 6.871 mas. We can see that the stars of the combination from catalogue 2 are closer together than the stars of the combination from catalogue 0. This is the same result which Grießbach et al. in [GWP21] present. They also present that the magnitude of the stars for combinations with lower FGS noise is often lower than the magnitude of the stars for combinations with higher FGS noise.

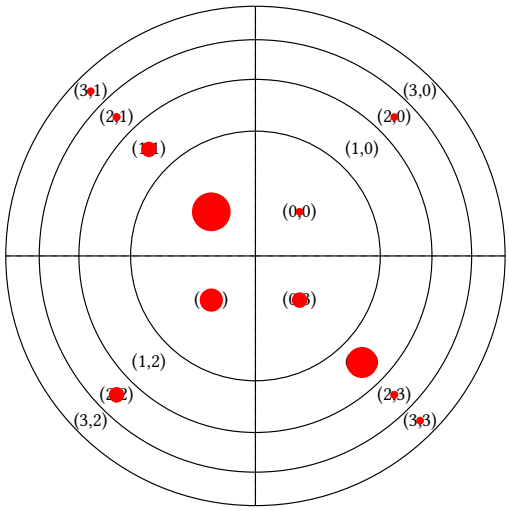
Although we reduced the search area and worked with the smallest catalogue, we were unable to achieve a significant improvement in the results. We have also increased the number of steps to achieve better results through more exploration. This also did not bring us any significant

improvement in results. Perhaps it is not the number of steps that is the problem, but our definition of the action space in which only minor changes are permitted. Or perhaps the way in which we use a separate embedding for each parameter in delta slows down the learning process of our algorithm. Since the star combinations show for the magnitude good results, the question arises as to whether our agent has learned more about the influence of the magnitude than about that of the position. That could be because it is easier to learn about the ECs of the magnitude because it is a smaller space. The position on the FPA is more complex because it consist of two values.

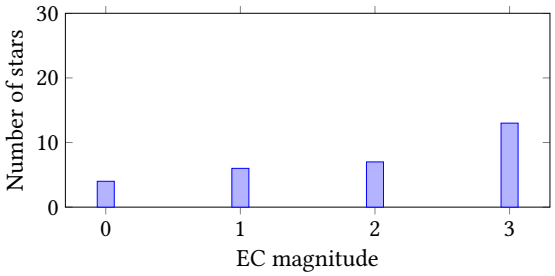
In the next section we compare our results with the results of the genetic algorithm presented by Witteck et al. in [WGH21].



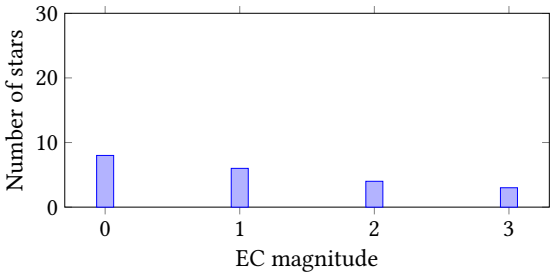
(a) Coverage of the FPA ECs for the best star combination for best case search



(b) Coverage of the FPA ECs for the best star combination for worst case search



(c) Coverage of the magnitude ECs for the best star combination for best case search



(d) Coverage of the magnitude ECs for the best star combination for worst case search

Figure 5.12: Distribution of stars of the best star combination over FPA ECs and magnitude ECs for the best case and the worst case found by our algorithm using Q-value calculator with node embeddings and using the absolute reward function for star catalogue 2; For the best case search the episode length equal 2,000 and a learning rate equal 1×10^{-10} and for the worst case search the standard configuration and 1,000 as episode length

	Catalogue 0	Catalogue 1	Catalogue 2
Best case	5.374	5.270	5.279
Worst case	6.496	6.525	6.414

Table 5.6: FGS noise [mas] using C++ random algorithm for 2×10^6 iterations

5.3 Comparison with the Genetic Algorithm

We compared the results of our approach with a random algorithm which is based on our implementation. The results for that random algorithm for the three catalogues are listed in Table 5.2. The results of our random algorithm are the best values from one execution with 1×10^6 iterations for each star catalogue and from one execution for catalogue 2 where we used 2×10^6 iterations. In order to compare the results of our approach with the results of the GA presented in [WGH21] we firstly compare the results of the random executions of both algorithms for the same star catalogues. The results of the random algorithm based on a randomised version of the GA that only performs a random selection of individuals without evolving them from one execution with 2×10^6 iterations are shown in Table 5.6. As the results show our random algorithm delivers for each star catalogue clearly lower values for the lowest FGS noise than the random algorithm based on the randomised version GA. Also our algorithm finds only lower values for the highest FGS noise except for the highest FGS noise of star catalogue 0. We explain this difference in results by the different programming languages since our approach is implemented in Python and the GA is implemented in C++. According to these results, the starting combinations of our approach are more likely to have a lower FGS noise.

In Table 5.7 we show the best results of the GA for each star catalogue. For catalogue 0 and catalogue 1, we list the results for 100 generations, as we only ran the algorithm for these catalogues with 10^6 steps. For star catalogue 2, we list the results for 200 generations, as we also ran our algorithm for this catalogue with 2×10^6 steps. When we compare the results with those of our approach, it becomes clear that the GA delivers significantly better results across the board regardless of the settings of our algorithm. The difference between the lowest found FGS noise is not that significant as the difference between the highest found FGS noise. This is probably because the difference between the FGS noise of the start combination to the FGS noise of the best possible combination is smaller than the difference between FGS noise of the start combination and the FGS noise of the worst possible combination.

In summary, the results of our approach are better than the results of both random algorithms, but do not yet match the results of the GA. The randomly selected starting combinations could influence this, as our approach may not see properties of combinations with high FGS noise. Then our algorithm does not learn how to optimise another combinations towards a higher FGS noise.

	Catalogue 0	Catalogue 1	Catalogue 2
Best case	3.455	3.527	3.729
Worst case	21.355	17.908	12.897
Generations	100	100	200

Table 5.7: Results of the GA

It is also possible that our algorithm needs more time to learn because of the same reason, even though we have already increased the total number of steps. However, the assumption that our agent does not see combinations that allow it to learn how to achieve the goal is refuted by the fact that our algorithm does not optimise towards combinations with a lower reward either. Even if our environment has a higher probability of starting with a combination with a lower FGS noise. With the current runtime of our implementation, our approach is slower than the implementation of GA, which has a runtime of approximately 25 minutes for an execution with 100 generations, a population size of 100, and 100 repetitions. In terms of our objectives, our approach does not reduce the knowledge that the user needs to execute the RL algorithm.

6 Conclusion and Outlook

In this thesis, we have presented a DQL approach for the automated generation of test cases for satellite on-board image processing algorithms. The biggest challenge of test case generation for that domain is the large search space. Due to the large search space it is not possible to test each possible test case. It is also not possible to easy identify significant test cases. For this reason, algorithms are used for automated test case generation. A successful approach so far is the GA proposed by Witteck et al. in [WGH21]. The problem with that result is that the algorithm often get stuck in a local optimum instead of finding a global optimum.

In this thesis, we have proposed an RL approach for the automated test case generation. To this end, we first defined an environment to encode the problem of automated test case generation in the satellite domain so that we could apply RL. For the definitions we used the EC definitions from [Wit+25]. Therefore we defined a state space, an action space and three reward functions: the absolute reward function, the difference reward function and the hybrid reward function. All three reward functions are based on the result of the fitness function, the fit, from [WGH21], as this approach yielded good results. Then we proposed two variants of our DQL approach. One uses what we call the Q-value calculator with node embeddings, the other uses the Q-value calculator with state embedding.

In order to evaluate our approach we used the FGS from the PLATO mission as case study. We conducted four initial experiments to make the following decisions:

- over which time reduce ϵ
- which α and β use for the hybrid reward function
- which reward function to use
- which variant to use

Based on the results of the experiments, we decided to use a longer period to reduce ϵ, and thus a longer exploration phase. We also decided to compare a more balanced hybrid reward function with the other two reward functions, rather than one that weights the difference reward lower than the absolute reward. Of our three reward functions, we initially focused on difference reward

function and we decided to use our approach with the Q-value calculator with node embeddings. However, we have found that the difference reward function is not optimal for achieving the goal of generating test cases with the lowest possible or highest possible FGS noise. We thought its not optimal because the signals that the difference reward function gives to the agent do not match the respective goal. Therefore, we decided to further test our approach with the absolute reward function, as it gives the fit as a reward when searching for the worst cases and the negative fit as a reward when searching for the best cases. Since we still did not achieve significant results we experimented with the episode length, the number of steps per episode and the learning rate in order to speed up the training. However, the results have not improved significantly.

When we compare our results with the results of random algorithms implemented in C++ and Python we achieve better results. Also the distribution of stars over the ECs of the FPA position and the magnitude mostly matched the expected results. However, our algorithm seems to learn better about the influence of the magnitude than about the influence of the FPA position of the stars in a combination. We also compared our results with those of the GA, but our algorithm does not achieve similar results.

In the future we plan to improve the runtime of our algorithm. We also developed ideas to optimise the learning process of our approach. One idea is to reduce the number of networks by removing the separate embeddings for the parameters of delta and combining them into a maximum of one embedding. With that we will simplify the training process and thus speed it up. It is also possible to allow larger steps for actions to enable faster exploration. Another idea is to redefine our action space by making the actions completely independent of the state space. For that, we define the actions by replacing one star from the star pool with another, taking into account the global ID rather than the local ID. In this case, we can also define restrictions for the permitted actions by masking invalid actions for the respective states. For example, an action that replaces a star that is not included in the current combination with another star from the star pool that is also not included in the current combination. In addition, we can limit the scope of actions by masking actions whose assigned EC IDs are outside a predefined range.

These are ideas to improve the problem encoding but we could also improve the RL training loop by using prioritized experience replay [Li23] or t-soft update [KI21]. If we are able to improve our learning process the next problem might be getting stuck in local optimum for that a possible solution could be applying the multi-agent approach from [Sco+21].

Otherwise, if we are still not able to improve the learning process with theses steps, it might be interesting to apply other RL algorithms to the problem, such as a policy gradient approach. But in case our learning approach has improved, it could be interesting to see whether other policy gradient approaches achieve similar or better results.

List of Figures

List of Tables

List of Algorithms

Bibliography

[GWP21] Denis Grießbach, Ulrike Witteck, and Carsten Paproth. "The fine guidance system of the PLATO mission". In: *International Conference on Space Optics — ICSO 2020*. Ed. by Bruno Cugny, Zoran Sodnik, and Nikos Karafolas. Vol. 11852. International Society for Optics and Photonics. SPIE, 2021, 118523H. DOI: 10.1117/12.2599604. URL: https://doi.org/10.1117/12.2599604 (cit. on pp. 5, 46).

[KI21] Taisuke Kobayashi and Wendyam Eric Lionel Ilboudo. "T-soft update of target network for deep reinforcement learning". In: *Neural Networks* 136 (2021), pp. 63–71 (cit. on p. 52).

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285. DOI: https://doi.org/10.1613/jair.301. URL: https://www.jair.org/index.php/jair/article/view/10166 (cit. on pp. 8, 18).

[KW16] Thomas N. Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016) (cit. on p. 12).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444. DOI: https://doi.org/10.1038/nature14539 (cit. on p. 10).

[Li23] Shengbo Eben Li. "Deep Reinforcement Learning". In: *Reinforcement Learning for Sequential Decision and Optimal Control*. Singapore: Springer Nature Singapore, 2023, pp. 365–402. ISBN: 978-981-19-7784-8. DOI: 10.1007/978-981-19-7784-8_10. URL: https://doi.org/10.1007/978-981-19-7784-8_10 (cit. on pp. 12, 13, 15, 52).

[Maz+21] Nina Mazyavkina et al. "Reinforcement learning for combinatorial optimization: A survey". In: *Computers & Operations Research* 134 (2021), p. 105400. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2021.105400. URL: https://www.sciencedirect.com/science/article/pii/S0305054821001660 (cit. on p. 17).

[Mir19] Seyedali Mirjalili. "Genetic Algorithm". In: *Evolutionary Algorithms and Neural Networks: Theory and Applications*. Cham: Springer International Publishing, 2019, pp. 43–55. ISBN: 978-3-319-93025-1. DOI: 10.1007/978-3-319-93025-1_4. URL: https://doi.org/10.1007/978-3-319-93025-1_4 (cit. on p. 6).

[Mni+15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533 (cit. on p. 12).

[Mog+20] Mahshid Helali Moghadam et al. “Poster: Performance Testing Driven by Reinforcement Learning”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 402–405. DOI: 10.1109/ICST46399.2020.00048 (cit. on p. 17).

[RAS20] Andrinandrasana David Rasamoelina, Fouzia Adjailia, and Peter Sinčák. “A Review of Activation Function for Artificial Neural Network”. In: *2020 IEEE 18th world symposium on applied machine intelligence and informatics (SAMI)*. IEEE. 2020, pp. 281–286 (cit. on p. 11).

[Rau+25] Heike Rauer et al. “The PLATO mission”. In: *Experimental Astronomy* 59.3 (2025), p. 26. DOI: https://doi.org/10.1007/s10686-025-09985-9 (cit. on p. 5).

[Ren+22] Haotian Ren et al. “Graph convolutional networks in language and vision: A survey”. In: *Knowledge-Based Systems* 251 (2022), p. 109250 (cit. on p. 12).

[SB+98] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998 (cit. on p. 10).

[Sco+21] Joseph Scott et al. “BanditFuzz: Fuzzing SMT Solvers with Multi-agent Reinforcement Learning”. In: *Formal Methods*. Ed. by Marieke Huisman, Corina Păsăreanu, and Naijun Zhan. Cham: Springer International Publishing, 2021, pp. 103–121. ISBN: 978-3-030-90870-6 (cit. on pp. 3, 18, 52).

[TM17] Hind Taud and Jean-Franccois Mas. “Multilayer perceptron (MLP)”. In: *Geomatic approaches for modeling land change scenarios*. Springer, 2017, pp. 451–455 (cit. on p. 11).

[TYG17] Fuxiao Tan, Pengfei Yan, and Xinping Guan. “Deep Reinforcement Learning: From Q-Learning to Deep Q-Learning”. In: *Neural Information Processing*. Ed. by Derong Liu et al. Cham: Springer International Publishing, 2017, pp. 475–483. ISBN: 978-3-319-70093-9 (cit. on pp. 12, 15).

[WD92] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3 (1992), pp. 279–292. DOI: https://doi.org/10.1007/BF00992698 (cit. on p. 10).

[WGH21] Ulrike Witteck, Denis Grießbach, and Paula Herber. “A Genetic Algorithm with Tournament Selection for Automated Testing of Satellite On-board Image Processing”. In: *Software Technologies*. Ed. by Marten van Sinderen, Leszek A. Maciaszek, and Hans-Georg Fill. Cham: Springer International Publishing, 2021, pp. 134–157. ISBN: 978-3-030-83007-6 (cit. on pp. 1–3, 5–7, 17, 19, 22, 31, 33, 47, 48, 51).

[Wit+25] Ulrike Witteck et al. “Explainable Input Partitioning for Robustness Testing of Satellite Image Processing using Principal Component Analysis [Manuscript submitted for publication]”. In: (2025) (cit. on pp. 2, 7, 8, 25, 51).

[WLL24] Yuqian Wu, Hengyi Luo, and Raymond ST Lee. “Deep Feature Embedding for Tabular Data”. In: *International Conference on Neural Information Processing*. Springer. 2024, pp. 102–116 (cit. on pp. 15, 29).

[Zhu+24] Yanfei Zhu et al. “UAV Path Planning Based on Random Obstacle Training and Linear Soft Update of DRL in Dense Urban Environment”. In: *Energies* 17.11 (2024), p. 2762 (cit. on p. 15).

Declaration of Academic Integrity

I hereby confirm that this thesis, entitled "Automated test case generation for satellite on-board image processing using reinforcement learning" is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media and AI Tools, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited. I am aware that plagiarism is considered an act of deception which can result in sanction in accordance with the examination regulations.

_____

(date, signature of student)

I consent to having my thesis cross-checked with other texts to identify possible similarities and to having it stored in a database for this purpose.

I confirm that I have not submitted the following thesis in part or whole as an examination paper before.

_____

(date, signature of student)