
Learning a State Representation for a Game Agent's Reactive Behaviour

Lernen einer Zustandsrepräsentation für das reaktive Verhalten einer Spielfigur

Bachelor-Thesis von Alexander Blank aus Bad Neustadt a.d. Saale

Tag der Einreichung:

1. Gutachten: Prof. Dr. Jan Peters

2. Gutachten: Dr. Oliver Kroemer



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Learning a State Representation for a Game Agent's Reactive Behaviour
Lernen einer Zustandsrepräsentation für das reaktive Verhalten einer Spielfigur

Vorgelegte Bachelor-Thesis von Alexander Blank aus Bad Neustadt a.d. Saale

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr. Oliver Kroemer

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15. September 2015

(Alexander Blank)

Abstract

In reinforcement learning, an agent interacts with its environment by taking actions and receiving rewards for those actions. A good example for such a task is a robot trying to clean up a park. The agent has to interact with multiple different objects and other agents in the park. To learn a behaviour in such a task it needs to be able to represent the state of his surroundings based on the distribution of objects he sees. Similar challenges can be found in arcade games wherein agents have to interact and avoid with objects in their environment. The goal of this thesis is therefore to learn the behaviour of a game agent. The agent will be presented with a view of the world consisting of a number of colored points in a 2D plane. Interactions such as slaying enemies and collecting gold result in rewards for the agent. The agent then has to learn a policy based on the distributions of the different object types in its surroundings. To learn such a policy, we use fitted Q-iteration. The Q-function computation is based on a variant of random trees which was modified into a representation that captures the key elements and conditions for action selection. We evaluate the parametrization of the approach and achieve better results than the standard grid-based state representation. We also explored and evaluated different representations for providing the agent with important global information, e.g. the location of a treasure in the game.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Problem Definition	3
2. Foundations	4
2.1. Reinforcement Learning	4
2.2. Random Forests	7
3. Experiments	12
3.1. Parameter Evaluation	12
3.2. Comparison of State Representations	17
3.3. Game with Additional Sword Actions	23
4. Conclusion and Future Work	26
Bibliography	29
A. Appendix	31
A.1. Parameter Evaluation: RMSE	31

Figures and Tables

List of Figures

1.1. An overview of our approach. We have a 2D game environment from which we extract the locations and type of objects in the current game view. This is then given as state to the policy learning part, where we apply a further state abstraction to form the basis of the policy. The state representation we use is a variant of random forests. Finally, we use fitted Q-Iteration to learn a behaviour.	2
2.1. The general steps of the fitted Q-iteration algorithm explained.	6
2.2. Two example tests inside the random forest structure. The blue test would evaluate as true, since there are less than 5 coins in the area. The test visualized in red would fail and the right child node in the tree would be chosen.	9
2.3. Different test sampling approaches. In random box we sample the 4 dimensions of the test. The fixed box is defined by its center and a certain Test width which defines the size of the square. An extension of this is the bounded box approach, where the test width is simply an upper limit for how far we go into each direction from the center. Finally, the classical approach simply chooses a split value and direction to generate the test.	10
3.1. When comparing different n_{min} there is no significant difference in the learning progress. We selected 30 as the minimum number of examples in a node.	13
3.2. The number of tests generated at each node in the tree evaluated. Using 50 tests resulted in the worst progression. Higher numbers perform better and therefore we select $K = 500$ as our final value.	14
3.3. The number of trees trained in each forest plays a big role in the convergence of the algorithm. The higher the number of trees the bigger the reward we converge to. We selected 30 Trees as the number for all further experiments.	15
3.4. The test width evaluated in the actual fitted Q-iteration. Really small tests perform worse in the beginning but converge to a higher reward collected. Too big tests perform as badly as expected. The final selection is a T_{width} of 1 for fixed and a T_{width} of 4 for bounded boxes.	16
3.5. Different Grid-sizes compared. Using a 10×10 grid worked the best out of all. Using a too fine grid of 21 performed the worst out of all configurations.	17
3.6. Execution of fitted Q-iteration with different state representations. The fixed test width random forests reaches the highest overall reward. The classical formulation of random forests fails to really improve much. The grid representation provides a very stable learning progress, but fails to reach the same levels as random forests with box tests.	18

3.7. Instead of using only the information from the local view in (a), we now also use information of the whole world (b). In (c) we visualize the minimap approach and the different objects spawned. The red circle means there is money in this area and the blue diamond means that the stack of coins is in the area. In (d) the arrow pointing towards the goal is shown. It is always close to the border of the local view of the agent.	20
3.8. Execution of the three exploration methods in comparison to the standard approach without any information about the whole game world. All methods perform worse than the normal version.	21
3.9. Comparison of different data set sampling methods. In the normal version we abandon the training data and model after each iteration. In the mixed approach we do 5 iterations where we only update the output of the random forest between each re-sampling of data. The latter approach proved to be more stable and reaches a higher reward.	23
3.10. Execution of fitted Q-iteration with different state representations in the 8-action environment. The grid representation fails to perform well in this setup. The classic version of regression trees is very stable and manages to collect a positive reward. Both the bounded and random box perform the best on this setup and the fixed box setup is extremely unstable.	24
A.1. Sampling 300 different tests at each node results in the lowest RMSE.	32
A.2. Turning a node into a leaf at around 20 samples remaining gives us the lowest overall RMSE.	32
A.3. The RMSE falls further the higher the number of trees is.	33
A.4. A test width of 2.5 results in the lowest RMSE.	33
A.5. Using different Grid sizes for the linear ridge regression results in different Errors. A size of 16×16 has the lowest RMSE.	34

List of Tables

1.1. The objects present in the game and their respective reward when the agent interacts with them in the environment by walking into them.	3
3.1. The best parameters resulting from the evaluation using RMSE. These values will be further evaluated on their actual performance when running the fitted Q-iteration.	13

1 Introduction

1.1 Motivation

The goal of this thesis is learning the behaviour of an autonomous agent based on its interaction with an environment. The agent could be a robot performing simple interactions with objects in different situations with varying number of objects and obstacles in the environment e.g. a robot that has to collect trash while avoiding obstacles in a park. It might also have to scare away magpies trying to steal trash from him. Since we do not know the locations and number of objects in the park, the robot has to learn a reactive behaviour based on the current local environment. In such problems we have a continuous environment with a manifold of different situations. The environment can also contain objects which are irrelevant for the optimal policy and the algorithm has to be able to deal with such a problem.

In arcade games we often encounter similar challenges. A games state changes rapidly when we take different actions and we want the agent to learn a policy that plays the game in a near optimal manner, while only observing the distribution of relevant objects like money or enemies in the space. In our case, the state is represented solely by the distribution of the game-objects in the 2D plane. These objects can be encountered in the game in varying numbers i.e. it is not previously known how many objects will be in which area. Actions in the game may include walking in different directions and interacting/attacking objects. As the game progresses a score is returned for certain state/action combinations, which is used as reward for our task.

Learning to play such an arcade game can be divided into three parts:

1. Mapping the observed scene into a set of objects with labels and poses.
2. Learn a state abstraction to form the basis of the policy.
3. Learn a policy for achieving a high score.

The first problem is difficult to handle for arbitrary games as one has to provide ways to reliably recognize the objects correctly and in any position. For example

game objects which are very close to each other often cause problems. In this thesis we have decided to focus on the latter two parts of the task.

In order to learn a policy that provides an optimal behaviour in a continuous state space where the number and location of objects in the area are previously unknown we use fitted Q-iteration. For this purpose we propose different state representations using a variant of random forests and thoroughly evaluate their performance.

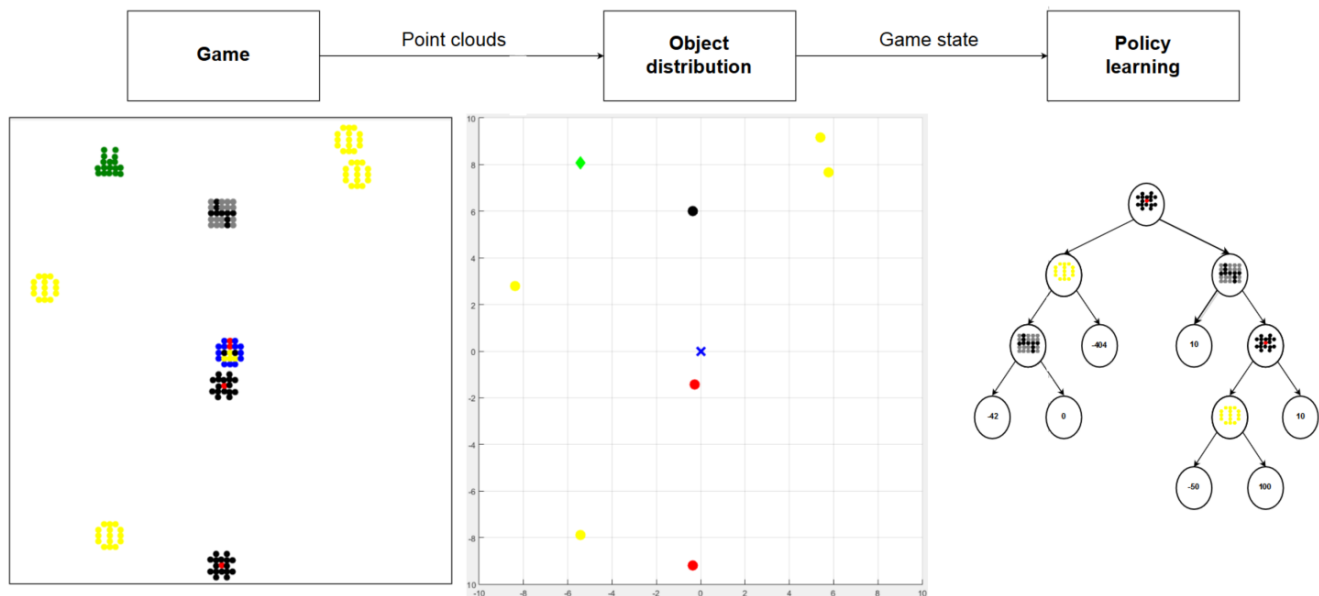


Figure 1.1.: An overview of our approach. We have a 2D game environment from which we extract the locations and type of objects in the current game view. This is then given as state to the policy learning part, where we apply a further state abstraction to form the basis of the policy. The state representation we use is a variant of random forests. Finally, we use fitted Q-iteration to learn a behaviour.

The rest of the thesis is organized as follows: In the first chapter we will further define the problem. Chapter 2 presents the foundations required to understand the used methods and introduces the different state representations. Afterwards, in Chapter 3 we first conduct experiments regarding the parameterization of the learning algorithm. Subsequently, we evaluate the performance of the algorithm on different difficulties of the game and additionally propose solutions which improve the stability of the learning process and explore different representations for providing the agent with important global information, e.g. the location of a treasure in the game. Finally, the Chapter 4 concludes the thesis with a recapitulation of the results and ideas for future work on this topic.

1.2 Problem Definition

The game environment used in this project is a randomly generated world filled with different objects. The size of the world is limited by a constant value $size_w$. This value gives an upper bound of how far the objects can be from the center. The objects are spawned uniformly within those limits. In the game itself, we always see a local section of the world which is centered on the agent and we can see objects in the area up to 10 units in the x and y directions. The objects inside the game are *money*, *grass*, *enemy* and *wall* as shown in Table 1.1.

The game initially supports four different moving directions as actions i.e. the set of possible actions is

$$A = \{a_{up}, a_{right}, a_{down}, a_{left}\}.$$

Executing one of those actions repositions the agent into the direction specified while also adding a random value between -0.1 and 0.1 to the position of the agent. The agent cannot walk through *walls*. Executing those actions returns a score from the game when interacting with objects i.e. when the agent collides with a object when walking into that direction. All sprites and the rewards for the actions are presented in Table 1.1.






Object	Collision	Sprite
Agent	/	
Money	100	
Wall	-50	
Grass	0	
Enemy	-300	

Table 1.1.: The objects present in the game and their respective reward when the agent interacts with them in the environment by walking into them.

The general goal of the game is getting the highest score possible. This goal can be achieved by avoiding the enemies while collecting as much money as possible and not getting stuck at walls. The positions of the objects in the space are given to the policy learner as the state. In order to generalize to different states, a suitable representation of the state has to be learned. We propose a representation using the properties of regression trees to generate different tests that capture varying numbers and locations of objects in the current state.

2 Foundations

In this section, we will discuss the algorithms used in our approach. In Section 2.1 we outline the basic reinforcement learning framework and describe the fitted Q-iteration. Afterwards, we introduce a variant of random forests for representing the object distribution in Section 2.2.

2.1 Reinforcement Learning

Reinforcement Learning is a discipline where an agent learns to behave in an optimal manner through trial and error. The agent interacts with the environment and receives a reward for performing suitable actions in certain states. These interactions together with the feedback received are used to learn which actions usually lead to a high reward in a certain state. This can be formalized as a Markov-Decision-Processes (MDP). While the game is actually a partially observed MDP, as the state of the objects beyond the screen are not observed, in this thesis we model it as an MDP and learn a policy only on the observed distributions. In Section 3.2.1, we additionally evaluate the case where the agent observes the entire world.

At the time t , the agent in a MDP is in a state $s_t \in \mathcal{S}$, where \mathcal{S} is the space of all possible states. The agent then executes an action $a_t \in \mathcal{A}$ from the action space \mathcal{A} . The agent chooses the action from a stochastic policy $\mathbf{a} \sim \pi(a_t|s_t)$. The agent then transitions from s_t to s_{t+1} according to a transition distribution $T(s_t, a_t, s_{t+1}) = p(s_{t+1} | s_t, a_t)$, which describes the probability of transitioning to state s_{t+1} from s_t when performing a given action a_t . An immediate reward signal $r_t = r(s_t, a_t)$, where $r(s_t, a_t) \in \mathbb{R}$ is returned by the system.

The usual format for one observation in discrete time is the current state s_t , the action taken a_t , the immediate reward r_t and the next state s_{t+1} of the environment after taking action a_t . The behaviour of the agent within the environment is denoted by the policy $\pi(a_t|s_t)$. The agent samples an action from π given

its current state. Finally, the goal is to find an optimal policy $\pi^*(a|s)$, which maximizes the expected long term reward $E[R|\pi]$, where

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1},$$

and the term $\gamma \in [0, 1]$ is a discount factor used to set the influence of future rewards.

The policy can be learned using a policy iteration approach. *Policy Iteration* algorithms alternate between policy evaluation and policy improvement phases. In the evaluation step, the value-function of all states is given by

$$V^\pi(s_t) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s \right],$$

which represents the expected discounted reward for starting in state s and using policy $\pi(a_t|s_t)$. This approach requires us to store a value for each possible state. Using this approach is not feasible in a continuous and high dimensional state space, because we would require a high amount of samples or an accurate model of the environment. On the other hand, the policy improvement part tries to improve the policy by taking actions with the highest quality. Sutton & Barto proposed the Q-learning algorithm [1], where the goal of the approach is finding the optimal value function by using the optimal Q-function for all state-action pairs. Where Q is defined as:

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + \gamma E_{p(s_{t+1}|s_t, a_t)}[V^\pi(s_{t+1})].$$

The Value function can then be expressed as

$$V^\pi(s) = \max_a Q^\pi(s_t, a_t).$$

In order to compute the Q-function for all state-action pairs, we iteratively collect samples and update the Q-function with

$$Q_{N+1}(s_t, a_t) = Q_N(s_t, a_t) + \alpha_t [r + \gamma \max_{a'} Q_N(s_{t+1}, a_{t+1}) - Q_N(s_t, a_t)].$$

If the state and action spaces are finite and small enough this can be solved in a tabular form. However, when dealing with continuous or large discrete state and action spaces the Q-function can no longer be expressed as a table with one

entry for each state-action pair. For this kind of problems an approximation of the Q-function has to be determined from a finite and generally sparse set of four-tuples (s_t, a_t, r_t, s_{t+1}) . Therefore, the fitted Q-iteration algorithm has been proposed by Ernst [2], which takes advantage of the generalization capabilities of regression algorithms and reformulates this problem into a sequence of regressions.

The Q-function is initialized as zero everywhere and in the first step an approximation is made by training a forest which has (s_t, a_t) as inputs and the instantaneous reward r_t as output. This first approximation is then used as policy in order to sample from the game again. In this thesis, we use an epsilon-greedy action selection policy that is defined by

$$\pi(a_t|s_t) = \begin{cases} \arg \max_{a \in A} Q(s_t, a) & \text{if } k \geq \epsilon \\ \text{random action} & \text{if } k < \epsilon, \end{cases}$$

where $k \in [0, 1]$ is drawn from a uniform distribution and ϵ is the term defining how likely we take an exploratory action. In the next iteration of the policy evaluation the output of the training set is updated using the value iteration approach with

$$Q_{N+1} = r_t + \gamma \max_a Q^\pi(s_{t+1}, a_t).$$

The Q-value of the next state s_{t+1} is drawn from the random forests of the previous iteration. This process is iteratively repeated until convergence. Unlike the original paper [2], we relearn the tree structure and re-sample the dataset after each iteration. The general steps of the algorithm are summarized by Figure 2.1.

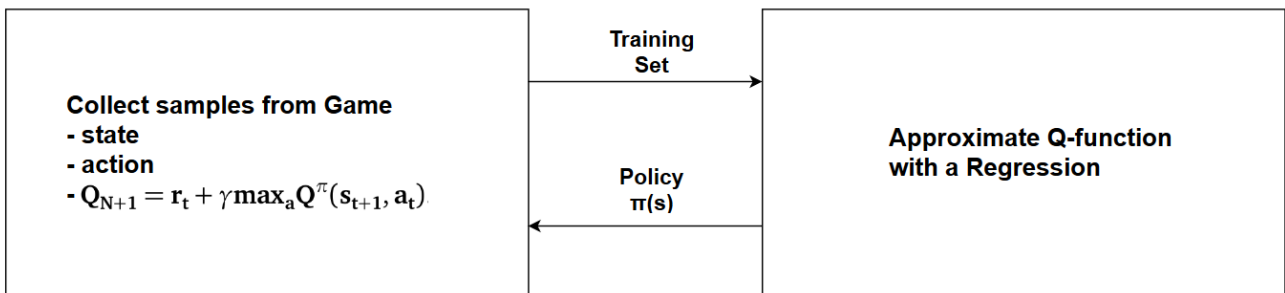


Figure 2.1.: The general steps of the fitted Q-iteration algorithm explained.

2.2 Random Forests

Tree based methods are a common approach for solving classification and regression problems. They are hierarchical structures consisting of nodes and edges connecting the nodes. Each node can have multiple children or none. A node without children is called a leaf. Each non-leaf node contains a test which decides the next node on the path down. In the classical formulation of decision trees, these tests are formulated as a boolean decision on an attribute i.e. if the value of a certain feature is smaller or bigger than some value x we split the data into the two parts depending on whether they satisfy this condition. The leaf node predicts the final output of the tree. The output value is a continuous real value in the case of regression trees.

A combination of multiple trees is called a forest. In the regression case the final output is formed by averaging over the output of all trees in the forest. This ensemble method works by growing many different trees from the same data set by randomizing the tree building process. For example, sampling the tests differently i.e. creating a completely random test at each node or selecting the best test out of a randomly created set. These random forests are an effective tool in prediction, because they are non-parametric and offer a great flexibility. They are also highly scalable to high-dimensional spaces and are robust to overfitting. One of the most popular formulations is the one from Breiman (2001) [3], where the randomness is brought in by choosing a random subset of the training set and also randomly sampling splitting directions.

The forest building process we used is called **Extra Trees** [4]. Each tree is learned by using the same training data \mathbf{TD} , which is a set of tuple (x_i, y_i) where x are the input and y are the output values. The training starts in a root node and a test is generated by creating a set of \mathbf{K} random tests which split the data into the two sets \mathbf{D}_a and \mathbf{D}_b . The final test is the one minimizing the relative variance of the output values y_i defined by

$$score = \frac{n_a}{n_a + n_b} var(\mathbf{D}_a) + \frac{n_b}{n_a + n_b} var(\mathbf{D}_b).$$

The test is then assigned to this node and the two children are created with their respective data set. This process continues until a node contains less than n_{min} samples. If this is the case, a leaf is created and is given the mean of all outputs in its respective dataset as value. This is done multiple times with the whole training set until we have a forest with \mathbf{M} independent random trees.

To obtain a value for an instance, we start at the root node and evaluate the instance x with the test. If the test is true, we evaluate the instance on the left node and if it is false we evaluate it on the right node. This is done until a leaf is reached. This evaluation is redone for every tree in the forest and the average of all results is returned as final output of the ensemble.

Using this non-parametric approach offers the ability to model any Q-function value, where the shape is a priori unknown. We chose this method due to its high computational efficiency, scalability to high-dimensional tasks and their robustness to irrelevant variables.

The structure of the forest depends on three key parameters. The number of trees M in each forest influences the smoothness of the output function and we will therefore evaluate it in the experiments. Afterwards, the minimum amount of examples n_{min} in a node has to be evaluated, because it has a high influence on the tree size. The number of tests K generated at each node will be evaluated in the experiments section, as the number controls the diversity between each tree in the forest [5]. The tree structure also has to be adapted in order to learn based on object distributions. For this purpose we propose a new representation for the tests in the next subsection.

2.2.1 State Representation of the Q-function

In our framework, the regression tree is used to approximate the Q-function. The input, i.e. the state of the game, is provided as a set of three-tuples

$$T = \{\langle x_1, y_2, type_1 \rangle, \dots, \langle x_n, y_n, type_n \rangle\}$$

where x and y are the coordinates of the object and the *type* is the kind of object present at that position (i.e. money, enemy, etc.). Each node in a tree has a test splitting the dataset into two child nodes. A test consists of:

- The 4 *dimensions* defining an area in the 2D plane $[x_{min}, x_{max}, y_{min}, y_{max}]$, where the x and y respectively are sorted.
- A type of object \overline{type} which is tested for.
- The number of objects z contained in the area.

The test is successful if

$$z > |type == \overline{type} \wedge x > x_{min} \wedge x < x_{max} \wedge y > y_{min} \wedge y < y_{max}|$$

is true i.e. there are less than z instances of \overline{type} in the area. Figure 2.2 shows two possible tests with its result in the evaluation. The test in blue is evaluated to true. On the other hand, the test in red fails. These two tests combined give the agent the information that going upwards is good. The blue test attracts the agent while the red test detracts it.

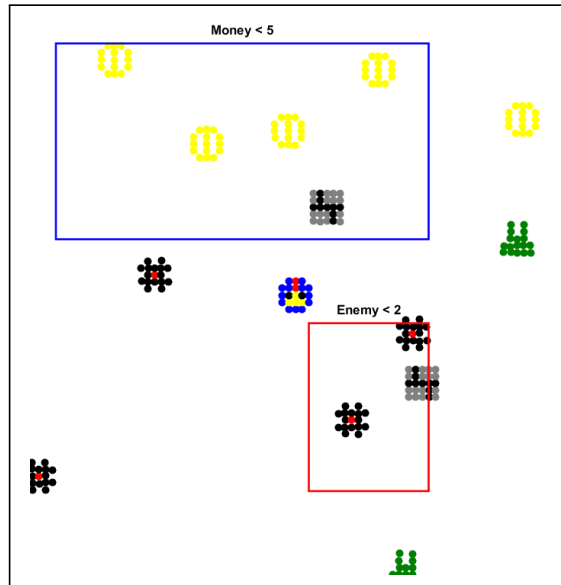


Figure 2.2.: Two example tests inside the random forest structure. The blue test would evaluate as true, since there are less than 5 coins in the area. The test visualized in red would fail and the right child node in the tree would be chosen.

The state representation for the agents behaviour is completely described as a combination of these tests in the tree structure. In this manner, we generalize to new states because the boxes generated for the tests are bigger than the samples we actually encounter and cover similar states. We also have a certain robustness towards irrelevant objects in the environment. In the example of our game world, the grass is almost completely ignored simply because we generate K tests and choose the one with minimal variance in the training data. Additionally, even if we have some tests which check for grass the influence will be irrelevant through averaging over the different trees, where the same test will not be found.

This representation also allows us to generalize to arbitrarily many objects in the space e.g. it doesn't matter if there are three enemies or twenty enemies, the same test would capture both cases. The Q-function would look different with a higher number of enemies in the area, but the behaviour resulting from the tests will still work well. Each test generated is a different feature for the regression and is used as state for the Q-function. The size of those tests plays

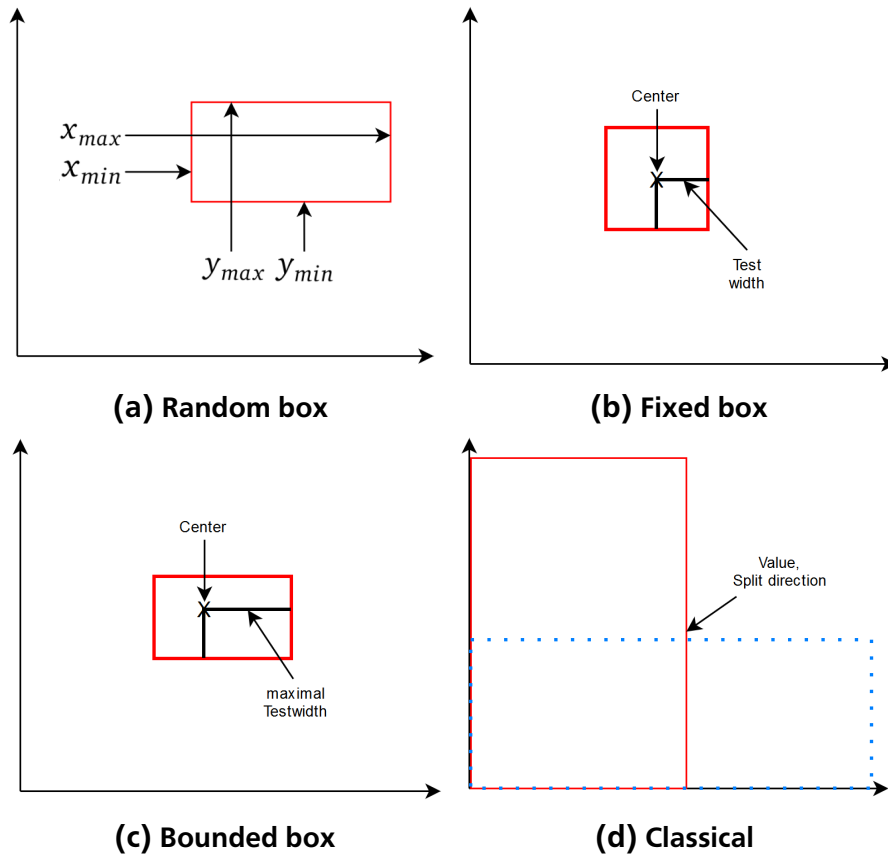


Figure 2.3.: Different test sampling approaches. In random box we sample the 4 dimensions of the test. The fixed box is defined by its center and a certain Test width which defines the size of the square. An extension of this is the bounded box approach, where the test width is simply an upper limit for how far we go into each direction from the center. Finally, the classical approach simply chooses a split value and direction to generate the test.

an important role in how finely we look at the state and how much we generalize between different states. Generating very big tests would lead to a badly defined behaviour since the tests are too general and the agent may not differentiate between an enemy 1 step or 5 steps away.

Therefore we propose multiple approaches to sampling the tests in order to control the trade-off between generalization and specialization. In the first formulation the 4 dimensions are sampled uniformly in the entire local game area. We will denote this representation as **random box**, since the shape and size of the tests is not limited in any way. Since the first version allows very big sizes of tests, the approximation of the Q-value can be too general. Therefore, the second representation uniformly samples the center of a test and creates a box of a specific size testwidth **TW** around it.

The width can be either be fixed, creating equal sized square tests or sampled from a uniform distribution where the upper limit is defined by **TW**. We evaluate both formulations called **fixed box** and **bounded box** in the experiment section. Finally, the last approach is essentially the classical way of creating random tests in regression trees. We choose a random split value and direction (i.e. x or y direction). Figure 2.3 shows the parameters we sample and the form of the tests for each approach.

3 Experiments

In this section, we describe the experiments for evaluating the learning of the game agent and discuss their results. Since the Q-function regression is essential for the success of the algorithm, we evaluated the performance of different parameter settings according to their accuracy when estimating the Q-value in the appendix. It is important to note that the policy learned with the lowest *root mean squared error* (**RMSE**) is not necessarily the best policy, but it gives us a feasible starting point for setting the parameters. Using the parameter settings from this evaluation, we compare different parameters on their actual performance when running the algorithm in Section 3.1. Next, in Section 3.2 we compare the different state representations with their best parametrization to each other on the game and do experiments on the encountered issues. Finally, Section 3.3 deals with the evaluation of the performance on a more complex version of the game.

3.1 Parameter Evaluation

We apply the fitted Q-iteration with the different parameters described in Table 3.1. They have been selected to test different interesting cases resulting from the evaluation with the RMSE in the appendix. In each iteration we collect 20,000 samples. Starting from a random policy, we run the fitted Q-iteration for 20 iterations and compare the reward collected after each iteration to each other. Although the training has been done using an epsilon greedy action selection with $\epsilon = 0.2$, the evaluation is performed using a deterministic action selection process that always chooses the action a that returns the highest Q-value. We also use the same reference world in each iteration of the evaluation. The world currently used for training and evaluation changes every 50 steps in order to capture the behaviour in many different situations. The plots presented always begin in iteration 0, where we use a random policy that collects a reward of about -100.000. Each parametrization was run 5 times and the presented results are the average over these 5 runs. The standard parameter settings used in this evaluation are $n_{min} = 30$, $K = 300$ and $M = 15$.

Table 3.1.: The best parameters resulting from the evaluation using RMSE. These values will be further evaluated on their actual performance when running the fitted Q-iteration.

	n_{min}	K Tests	M Trees	Test width (TW)	Gridsize N
Test	10,20,30	50, 300, 500	5, 15, 30	1, 2.5, 4	10,15,21

3.1.1 Minimum Number of Examples in a Leaf

This experiment focuses on finding a good value for n_{min} for the tree training. This value influences the structure of the tree. Choosing a low value increases the depth of the tree because much more leaves are required to represent the training data. On the other hand, a really high value creates a shallow tree with very general tests. Since the shape of the Q-function is previously unknown, but in many applications has a very spiky form, choosing a high value leads to higher errors. Using a lower n_{min} has proven successful for our task. Looking at the results presented in Figure 3.1, we see that when we have a low number of examples, there is no significant difference in the results. Using values of n_{min} bigger than 50 led to an extremely unstable learning and the reward would often decrease after an iteration. The final choice of $n_{min} = 30$ was made because the learning starts with a very high progress in the first few steps when using this value and additionally it leads to the smallest trees, which decreases computational time.

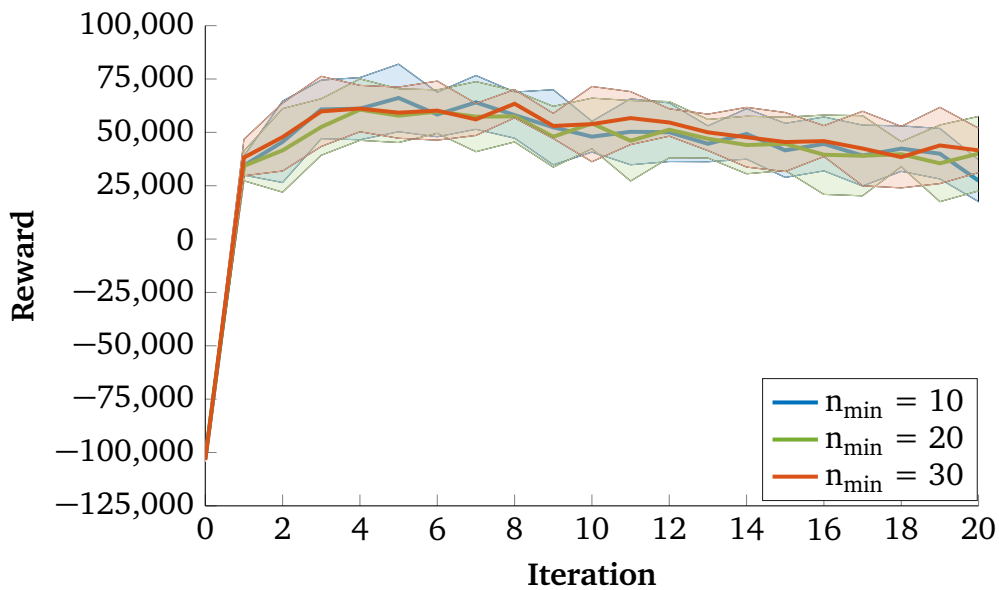


Figure 3.1.: When comparing different n_{min} there is no significant difference in the learning progress. We selected 30 as the minimum number of examples in a node.

3.1.2 Number of Tests in each Node

The next parameter we investigated is the number of tests K created at each node in the forest. The number represents how many different random splits we generate at each node in the tree building process. It is therefore responsible for the trade-off between the randomness of the trees and finding the best split at the cost of computational time. In the literature it is often suggested to choose a value depending on the number of features m you have, i.e. a good value often is \sqrt{m} [3]. In our case, each unique test in combination with an object and a threshold is basically a feature. Hence, we cannot use this recommendation and we need to evaluate it empirically.

It has to be mentioned that in case the tests generated are not sufficient to determine a split of data, we repeat the process up to ten times before turning the node into a leaf. We also do not want too many tests in order to keep the difference between trees higher. Creating too many tests would result in very similar tree structures, which leads to worse generalization. We therefore, according to the results in Figure 3.2, select $K = 500$ as the final parameter. This value is very stable in the first few iterations and provides good results.

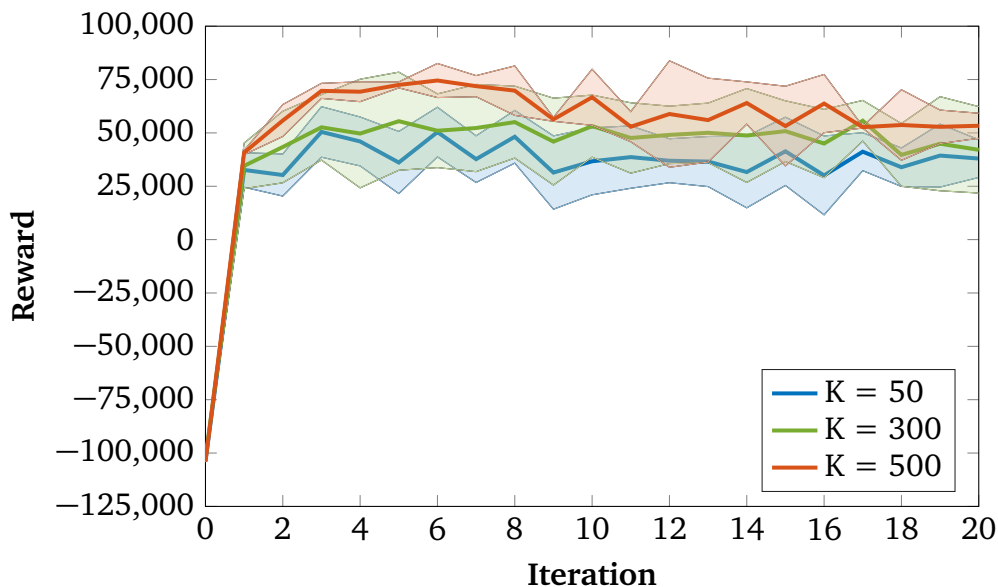


Figure 3.2.: The number of tests generated at each node in the tree evaluated. Using 50 tests resulted in the worst progression. Higher numbers perform better and therefore we select $K = 500$ as our final value.

3.1.3 Number of Trees

The last important parameter for random forests is the number of Trees M in each forest. Increasing the number of trees does not overfit the data and should therefore be as high as possible. We try to find a value which is sufficiently large such that the accuracy does not increase much more when further increasing the number of trees. It is also noteworthy that while they are a powerful tool for prediction, the training of random forests is computationally expensive. Using a high number of trees is advantageous, but at the cost of high model learning time. In Figure 3.3 we can see how the reward collected behaves when increasing the number of trees. As expected, using more trees results in better learning progression. Creating even more than 30 trees didn't lead to an significant increase and therefore we select $M = 30$ as the final value.

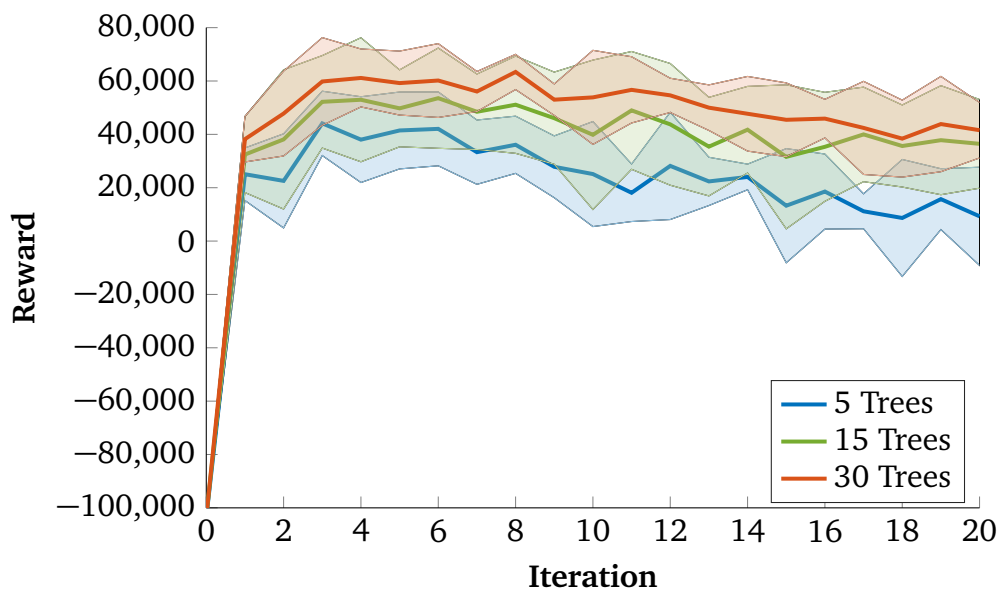


Figure 3.3.: The number of trees trained in each forest plays a big role in the convergence of the algorithm. The higher the number of trees the bigger the reward we converge to. We selected 30 Trees as the number for all further experiments.

3.1.4 Test Width

Here we evaluate the different test widths used for representation where we sample the center of the test and create a box around it by going T_{width} into every direction, namely **fixed box** and **bounded box**. This creates a lot of equal sized tests and has proven to perform well on the actual learning task. The results received with different test widths (TW) are shown in Figure 3.4. For the fixed box we receive the highest reward using a really small test size of 1. The convergence is less stable than using a test size of 2.5 but it was consistently

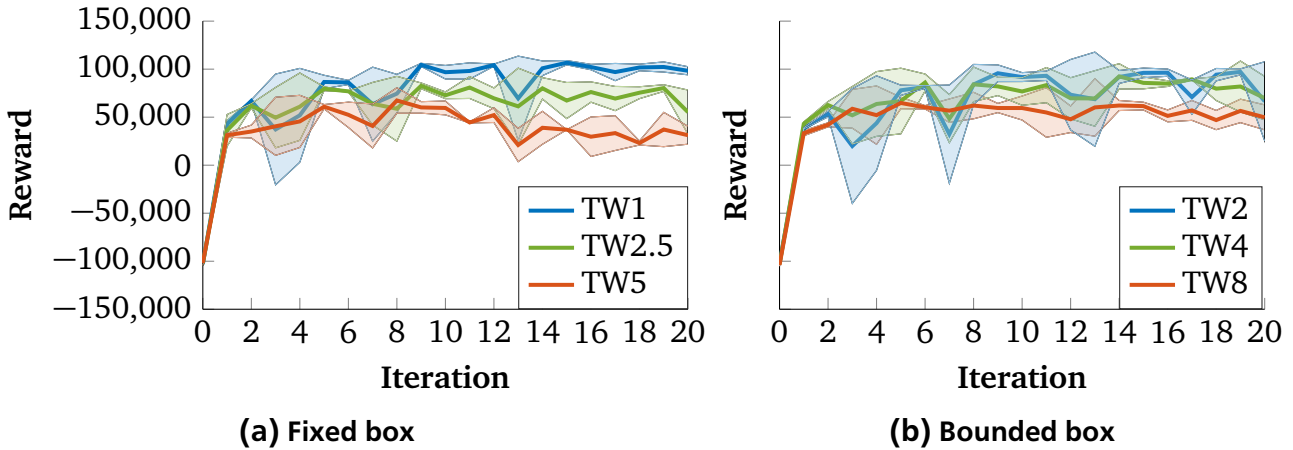


Figure 3.4.: The test width evaluated in the actual fitted Q-iteration. Really small tests perform worse in the beginning but converge to a higher reward collected. Too big tests perform as badly as expected. The final selection is a T_{width} of 1 for fixed and a T_{width} of 4 for bounded boxes.

able to outperform it in the end. The final choice will therefore be a test width of 1. For the bounded boxes, the highest reward was obtained with a test width of 2. However, since the learning is more stable using a TW of 4 and also the overall largest reward was gained using this value, we will use this as our final parameter.

3.1.5 Grid Size

The last parameter we evaluated is for the benchmark representation. This representation uses an $N \times N$ grid of features to represent the object distributions. Each bin in this grid contains the amount of objects of a certain type in it e.g. in the game we would have $N \times N \times N_{types}$ bins representing the state. We then use linear ridge regression to learn a simple model using this vector of bins as the state.

The grid size N controls how finely we represent the game state. The higher the grid size, the more features we have. However, the smaller the grid elements, the less the state generalizes. Since we are using linear regression, we also need more samples to fill the feature-space with examples when increasing the number of grids. In Figure 3.5 we can see the result with different grid sizes. Both 15×15 and 21×21 performed far worse than 10×10 .

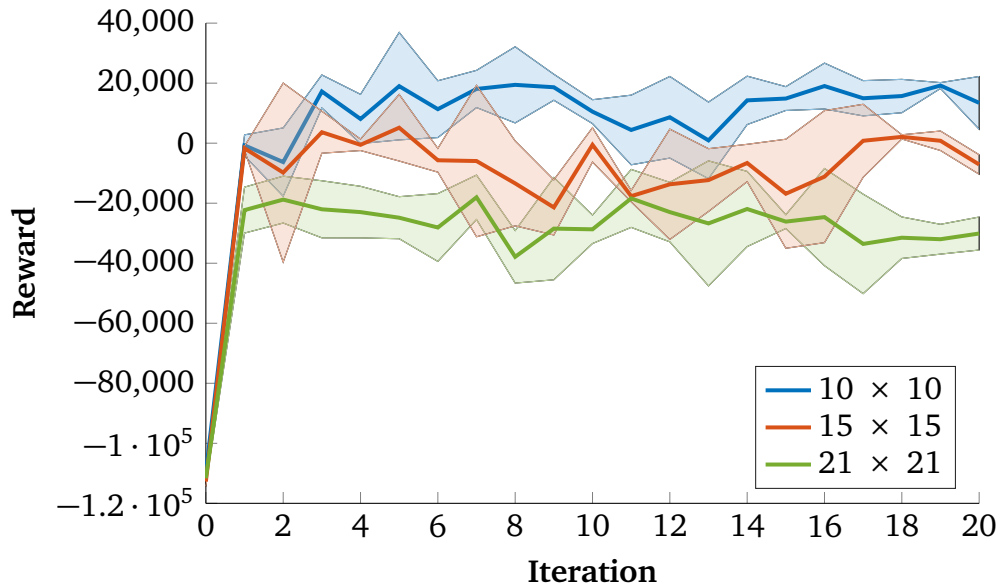


Figure 3.5.: Different Grid-sizes compared. Using a 10×10 grid worked the best out of all. Using a too fine grid of 21 performed the worst out of all configurations.

3.2 Comparison of State Representations

As previously mentioned, in this game mode we have four actions

$$A = \{up, right, down, left\}.$$

Therefore, the optimal behaviour is running away from the enemies while collecting as much money as possible and not colliding with walls. We start by collecting a dataset of 20,000 samples from a policy $\pi_0(s)$ returning a random action. Every 50 samples we restart the game with a new random level. For the rest of the iterations, the agent uses an epsilon greedy policy to balance exploration and exploitation. The amount of exploration is controlled by the parameter ϵ , that determines the probability of taking a random action instead of the optimal one. We set ϵ to 0.2 for the experiments. The evaluation is done on a separate validation world, where there is no random actions and the agent always chooses the action according to the highest Q returned from one of the forests.

The Figure 3.6 shows a comparison between the different state representations performing 20 iterations. All runs collect a reward of approximately -100,000 in iteration 0. We run the algorithm 5 times with the same parameters and plot the mean and the standard deviation of the reward collected in 20,000 samples at each of those 5 runs. The classical formulation for random forest tests

performs the worst out of all. It is a very unstable learning process and does not manage to gather a positive reward in the end. The grid representation slowly converges to a locally optimal policy collecting about 20,000 reward. It is interesting to note that while the obtained policy collects a positive reward it looks very unstable because the area of effect each object has on the Q-value is big. The agent often goes from left to right and back again because the value changes drastically when moving in the environment. This could also be due to the discretization error made.

The representation using completely random boxes on average collects a reward of 70,000. Finally, the representation using equal-sized boxes shows a slightly less stable learning process but converges around 100,000. The bounded box approach performs similar to the fixed box. The highest the agent collected overall was 120,000 reward with a fixed box policy. Considering the fact that a coin gives 100 points while colliding with an enemy gives a penalty of -300, this is a very high score. When the game is played by a human, they are able to collect approximately 150,000 with the same amount of samples.

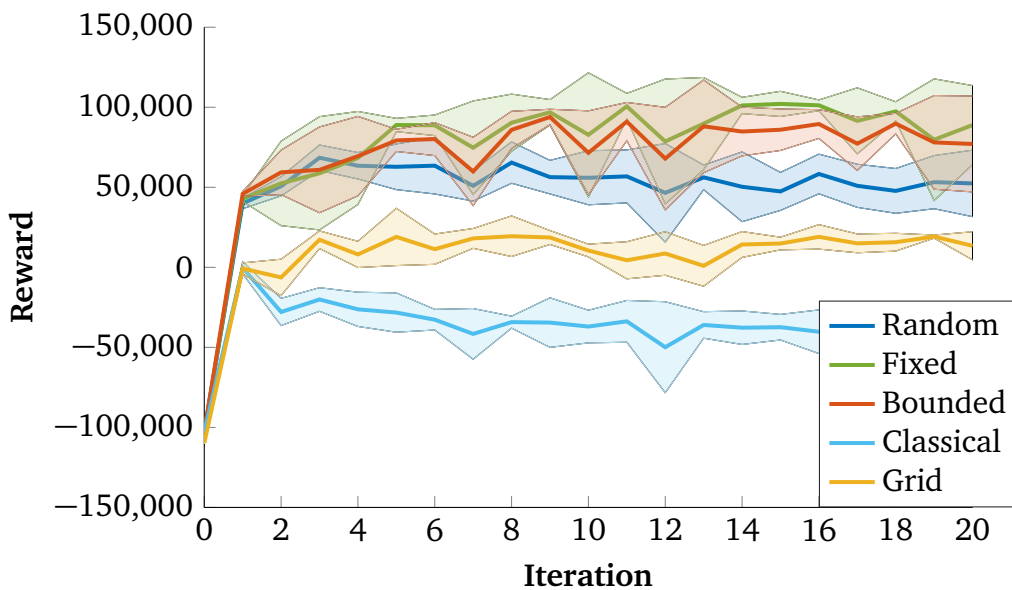


Figure 3.6.: Execution of fitted Q-iteration with different state representations. The fixed test width random forests reaches the highest overall reward. The classical formulation of random forests fails to really improve much. The grid representation provides a very stable learning progress, but fails to reach the same levels as random forests with box tests.

There are multiple reasons why the random box does not reach as high returns as the two approaches where we control the size of the tests. The problem with the random box approach is that the tests are often large and general, or too

small and specific. For example, a test may cover the majority of the observed area, or it may be only a couple of pixels wide. These tests often do not even separate the training data. The more regularly shaped bounded and fixed box approach do not exhibit this issue as much and generalize to medium-sized regions.

Another reason we do not get higher returns is that the method is only a reactive policy. The agent can only behave optimally in the local view of the game, since it does not know anything about the surroundings. In situations where there are no more coins or enemies left, the behaviour has no guidance. This leads to the agent being stuck at those local positions, without exploring further.

To improve on this problem, we have to additionally add information about the global environment - the whole game world - to the learning process. While we could make the agent follow a manually defined trajectory in the world and only learn the behaviour in local parts of the game, we preferably would inherently learn the global behaviour of the agent with a modification of our method. In the next subsection, we will do experiments regarding solutions for this problem.

Another problem that has to be analyzed is the stability of the methods after each iteration. Often the return crashes massively after a good iteration i.e. it falls from a really high value to a low value. This happens because we re-sample completely from the game after each iteration and forget any information previously obtained. If for example we get a good policy in one iteration, the next one often fails to perform well because we don't get enough negative examples when collecting samples with the good policy. One would have to keep the information i.e. with importance sampling or similar approaches to reach a more stable progress in this setup. Another way to look at this problem is adjusting the way we sample the training data.

3.2.1 Exploration Problem

In this section, we investigate multiple methods for incorporating the global state of the game into the learning. As shown in Figure 3.7, the agent now has information of the whole world instead of only its local sub-view. We want to use this information together with the proposed algorithm in order to learn the behaviour of the agent even if the Q-function is flat i.e. there are no enemies or money in the area. In order to test if the agent is able to learn a global

behaviour we added a goal to the world. When generating the game we now create 20 coins very close to each other in an area near the border of the world. If the agent can constantly find this stack of coins he will gain a much higher reward than previously.

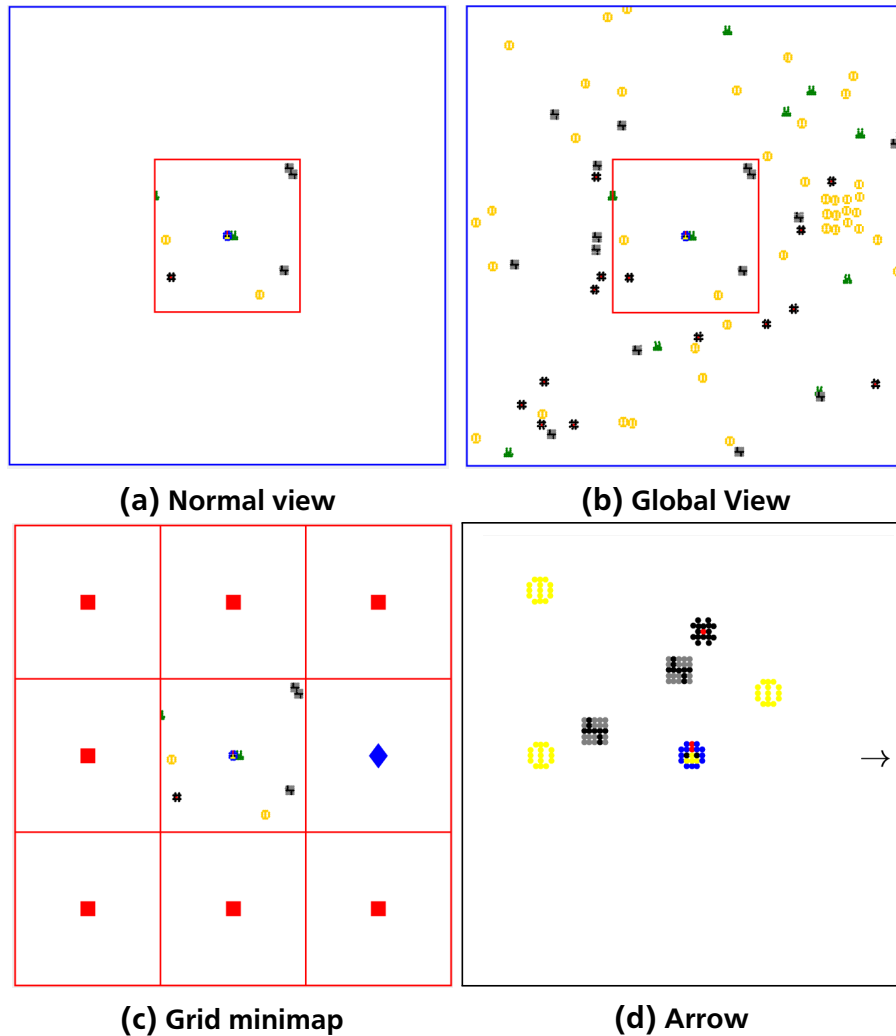


Figure 3.7.: Instead of using only the information from the local view in (a), we now also use information of the whole world (b). In (c) we visualize the minimap approach and the different objects spawned. The red circle means there is money in this area and the blue diamond means that the stack of coins is in the area. In (d) the arrow pointing towards the goal is shown. It is always close to the border of the local view of the agent.

In the first approach we give the whole world as state for the training data. In this manner, the tests are generated across the entire game world and the state is fully observable. The second approach transforms the world into a compact minimap as shown in Figure 3.7c. We split the world into a grid where each cell is as big as a local view of the agent. For each cell we add an object to the area if there is gold in the area and the agent hasn't previously visited it. Addi-

tionally, we add a different object to the cell which contains the stack of coins. This global overview in the form of objects is then added to the local view of the agent. Different distributions of those objects should provide the agent with the necessary information to find the goal and more gold. Once the agent has entered a cell, the object is removed. This adds two more objects to the learning.

For the last approach shown in Figure 3.7d we added an arrow to the local view of the agent which always points into the direction of the goal. This method adds one object to the learning process and provides information of the global location of the stack of coins.

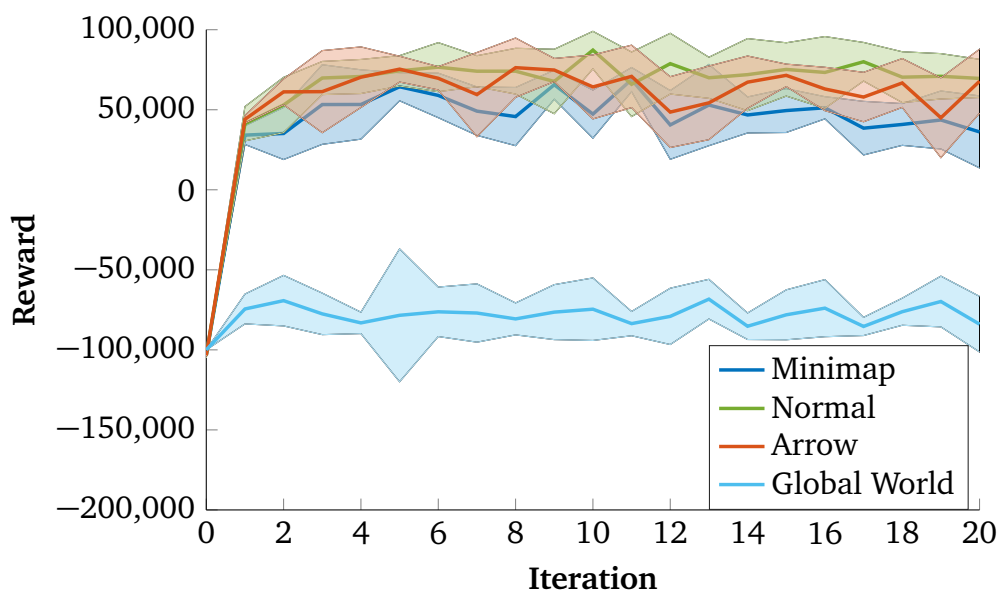


Figure 3.8.: Execution of the three exploration methods in comparison to the standard approach without any information about the whole game world. All methods perform worse than the normal version.

For all runs in this section, we will use the random box sampling with $N = 30$, $K = 500$, $M = 30$. We collect 20,000 samples at each iteration and average all results over 5 runs, while also depicting the standard deviation. In iteration 0 the agent collects a reward of -100,000.

The results in Figure 3.8 show that none of the methods massively increased the reward collected. The normal method without any global information outperforms all others. The arrow and minimap version perform only slightly worse, which is caused by the additional objects in the state representation that make the game more complex. The game does not discover the treasure often enough to make use of those cues. Finally, the global world view performed poorly staying very close to a random policy in behaviour and return.

These results show that it is not straight-forward adjusting the method in order to include global information. The global view suffers from too many objects and too much space. The tests created are very big no matter where they are created. Finding a good way to sample the tests in such an environment would make this work better e.g. prioritizing local features close to the agent while still using global tests.

The objects depicting the global state in the two other approaches have no effect on the return. This could be caused by the fact that we forget information from previous iterations and completely re-sample. The additional context may also require more samples in order to capture the information provided by the new objects. The structure of the actual rollout might also need adjustment in order to find the treasure more often while exploring.

3.2.2 Sampling

For this experiment we will look at different ways of handling the data between each iteration. In the previous experiments we re-sampled a completely new training set from the current policy and learned a completely new random forest from this data. The original version of fitted Q-iteration described in [2] is inherently an offline method - given the initial data set, the algorithm estimates the optimal policy by only updating the output of the samples after each iteration. When we ran the algorithm this way, the returns increased very slowly and converged to a way too low value.

Therefore, we mix the two versions together and after every 5 iterations we completely abandon the model and re-sample the data using the currently best policy. This turns the process into an online learning approach. Each iteration shown in the experiment is obtained by sampling from the policy learned directly after the re-sampling of the data. The parametrization for the trees is the same as in the previous experiments.

As Figure 3.9 shows the mixed approach led to a very stable convergence of the algorithm and the mean was much higher than in the version where we completely abandon the dataset/model after each iteration.

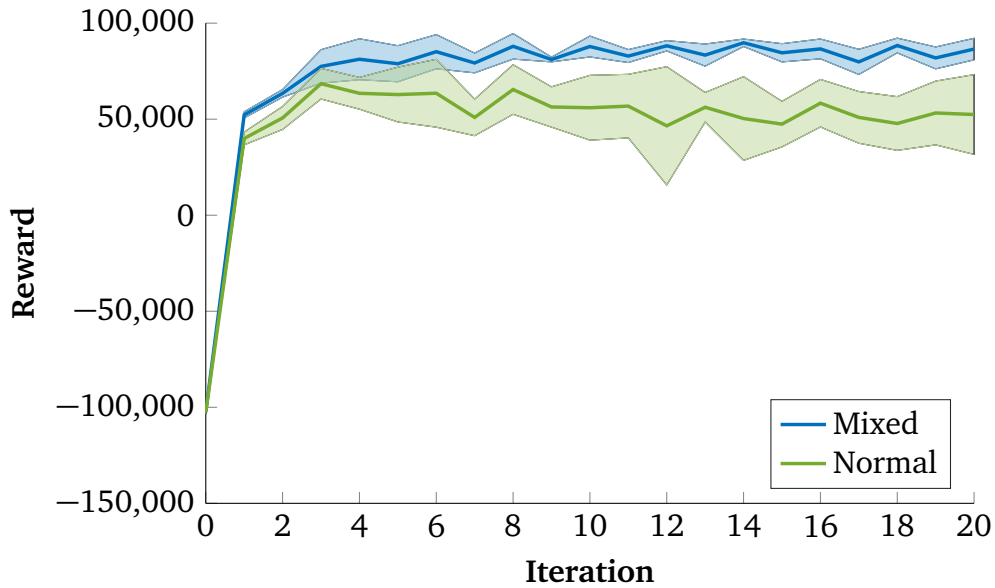


Figure 3.9.: Comparison of different data set sampling methods. In the normal version we abandon the training data and model after each iteration. In the mixed approach we do 5 iterations where we only update the output of the random forest between each re-sampling of data. The latter approach proved to be more stable and reaches a higher reward.

Although we achieved a more stable learning progress with additionally higher average reward, it has to be noted that the policy which obtained the highest reward overall still was obtained from the normal method. This is again caused by sampling from a too good policy, where the bad situations aren't encountered often enough and therefore the next iteration gets a low return. After such a low reward, the agent has enough information to avoid bad situations and the reward becomes very high in the next step. This is suboptimal and we hope to find a method that keeps information from previous iterations and combines it with the current model in future work. This way we would reach higher returns while also being stable because we do not forget important previous information.

3.3 Game with Additional Sword Actions

In the section we present the experiments on the game with eight actions

$$A_{ext} = \{up, right, down, left, upswipe, rightswipe, downswipe, leftswipe\}.$$

The agent now additionally has the ability to do an attack into one of four directions with his sword. Hitting an enemy returns a high positive reward of 300, but he can not win against more than one enemy. Fighting multiple enemies at

the same time gives a negative reward of -900. Swinging his sword against a stone or in the air gives a negative reward of -750.

This way he can collect a high positive reward by collecting coins and hunting single enemies, but has to avoid swinging his sword wildly. For this experiment, we use 40,000 samples in each iteration while training. However, the evaluation is still done with 20,000 samples. The parametrization used here is the same as in the 4-action case. Iteration zero is not shown in the plot because the agent collects a reward of about -6,000,000 with a random policy.

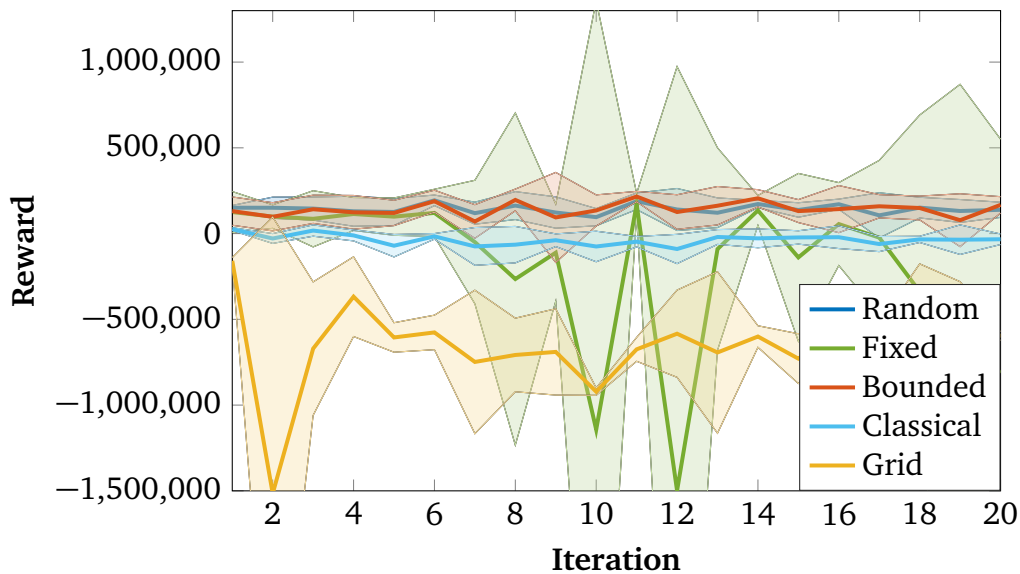


Figure 3.10.: Execution of fitted Q-iteration with different state representations in the 8-action environment. The grid representation fails to perform well in this setup. The classic version of regression trees is very stable and manages to collect a positive reward. Both the bounded and random box perform the best on this setup and the fixed box setup is extremely unstable.

As shown in Figure 3.10 we manage to obtain a high positive reward of up to 270,000 in this setup. The highest return was obtained using the bounded box setup, while the random box setup also achieved good results of up to 250,000 points. The fixed box representation is extremely unstable in this setup and the reward sometimes drops to values worse than random policies obtain. This result is related to the previously described problem that we don't use information of previous iterations. The grid representation improves in the first iterations, but fails to get a positive return. The classical formulation of random forests perform fairly well and converge to a reward of approximately 50,000. In summary, we were able to learn a good behaviour for the agent, but the extension to eight actions has again shown the importance of the trade-off between generalization

and specialization. The fixed box approach with this parametrization provides a too fine representation of the state. The area of effect the sword swipe has is too big. Therefore, one test can not represent that it is bad to swing the sword when there are multiple enemies approaching. The more general bounded box performs much better here. The grid representation has another problem besides the relatively fine representation. For each bin, we obtain one weight after learning. This weight is either positive or negative. It can not capture the condition that when there are two enemies in the area it is bad to swing and when there is one enemy, it is good. The tree based approaches are capable of dealing with this condition.

4 Conclusion and Future Work

In this thesis, we have learned the behaviour of an game agent based on object distributions in his surrounding. We proposed a variant of random forests as an abstract state representation for the agent and applied the fitted Q-iteration to learn an optimal policy for the agents behaviour.

The tree based fitted Q-iteration with the classical way of sampling tests by choosing a separating hyperplane in the feature space performed worse than our benchmark grid representation. Using the proposed changes to the way we sample the tests we were able to outperform the benchmark. We also analyzed different ways to sample the size of the boxes that are created in the tree structure. Both the fixed size and bounded size tests were able to outperform the random sampled tests in the 4-action case. This result showed the importance of the trade-off between generalization and specialization of the tests. The random box creates a lot of very general and very specific (i.e. small) tests which are not even splitting the data set. Limiting or fixing the size of the tests therefore improved the performance of the algorithm.

When further increasing the number of actions by adding sword attacks the fixed boxes performed poorly, while the bounded boxes were still able to perform well. This problem is caused by the big area of effect the sword swipe has and the fact that it is bad to attack more than one enemy. The multiple small tests are not big enough to cover the entire swipe area and therefore it requires multiple tests to describe the optimal behaviour. More general tests like the bounded or random box approach are able to handle such situations. In conclusion, the optimal way of sampling the tests depends on the problem setting.

Next, we also evaluated different methods of sampling the training data and updating the policy between each iteration. Instead of re-sampling the training data after each iteration, we were able to improve the learning progress by re-sampling the data from the currently best policy after each 5th iteration. For all the other iterations, we only update the output value of the model. This approach still abandons previously learned information which could be used to improve the model. However, the old information might still be important

and should be re-used by for example using importance sampling or similar approaches.

Although we were able to collect a very high reward, we are still 30,000 points below a human playing the game. We discovered that the biggest difference is the behaviour when there are no observable attractors or detractors, e.g. coins and enemies. The policy learned relies on what it sees in its local area and providing the agent with the state of all objects in the game world did not lead to a good performance. The method works well using only local information, and can be used with more global information. However, the agent will need a policy with more structured exploration to explore more of the game world and thoroughly exploit the global information.



Bibliography

- [1] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.
- [2] D. Ernst, P. Geurts, and L. Wehenkel, “Tree-based batch mode reinforcement learning,” *J. Mach. Learn. Res.*, vol. 6, pp. 503–556, Dec. 2005.
- [3] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [4] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Mach. Learn.*, vol. 63, pp. 3–42, Apr. 2006.
- [5] S. C. Amend, “Feature extraction for policy search,” bachelor thesis, TU Darmstadt, Mai 2014.



A Appendix

A.1 Parameter Evaluation: RMSE

In this section, we evaluate the performance of the parameters regarding the regression. This gives us a selection of parameters to choose from for the actual execution of the fitted Q-iteration. For this purpose, we use the Root Mean Squared Error (RMSE) defined as

$$RMSE = \sqrt{\frac{\sum_{t=1}^n (\hat{y} - y)^2}{n}}$$

a measure for the quality of the approximation in the first step of fitted-Q iteration. To evaluate the performance of the regression, we collected a data set \mathbf{TD}_1 which represents the Q-values in the first iteration of the algorithm i.e. there is no discounted future reward included and we only look at the immediate return received in this step. The training set consists of 20.000 samples collected by executing random actions in the game. The whole evaluation is done as a 10-fold Cross-validation in order to check the accuracy of the model on unknown states. Since the original data set is collected by executing a row of actions, we shuffle the dataset randomly so each bin for the Cross-validation has a wide range of different situations. Because for the learning part we split the dataset according to the action taken, here we evaluate the accuracy of each forest representing an action separately. We then average those four RMSE values and evaluate the combination. This allows us to choose the lowest RMSE for all actions. We compare different parameters for the Random Forests to each other. The same process is done for the grid representation for the state in order to select the number of bins N and for the test width in the alternative formulation of the tests.

A.1.1 Number of Tests in Tree

Following the procedure described at the start of this section we get the results presented in Figure A.1. Choosing the number of tests as 300 leads to the lowest overall error on the regression. It has to be mentioned that in case the algorithm

doesn't find a split in those 300 trials, it retries a few times before giving up and turning the node into a leaf. This leads to a overall lower RMSE.

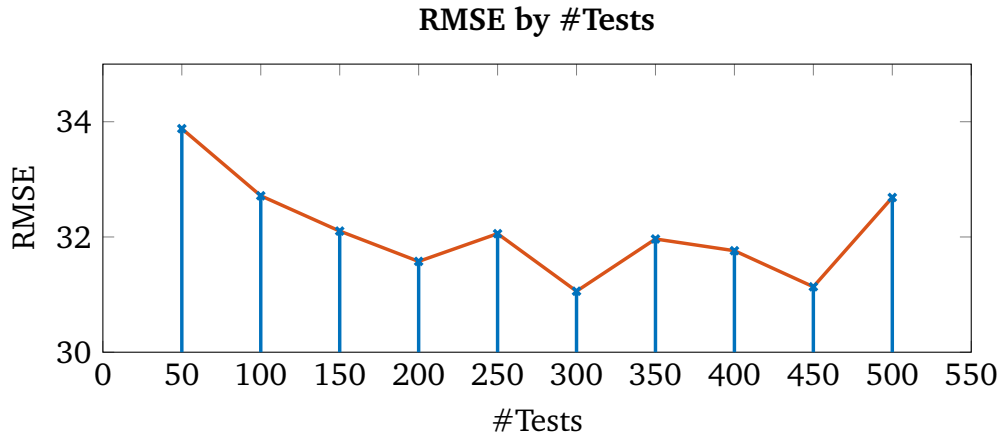


Figure A.1.: Sampling 300 different tests at each node results in the lowest RMSE.

A.1.2 Minimum number of examples in a leaf

The results of the evaluation are depicted in Figure A.2. Too low values performed as bad as too high values while an n_{min} between 15 and 25 has shown good results. We will further evaluate n_{min} of 10, 20 and 30 in the real execution of fitted Q-iteration.

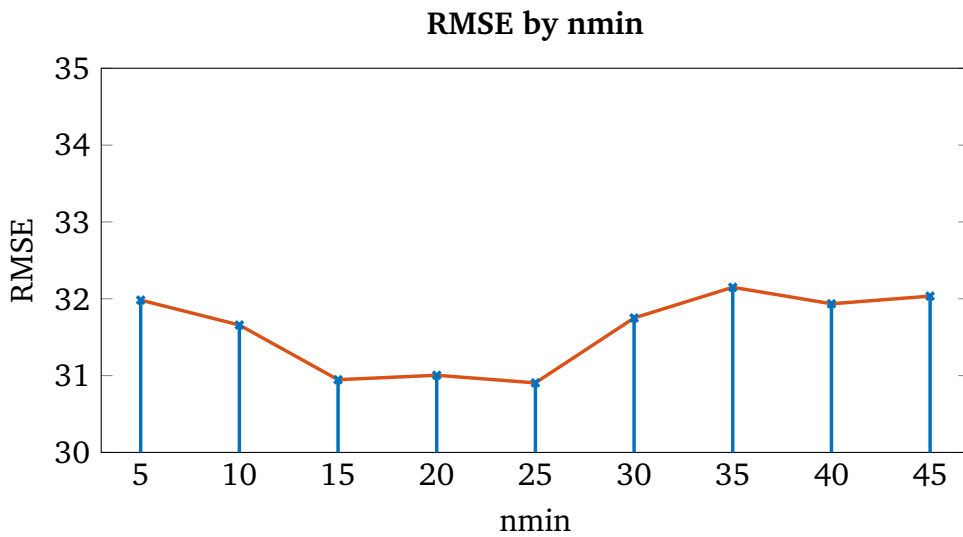


Figure A.2.: Turning a node into a leaf at around 20 samples remaining gives us the lowest overall RMSE.

A.1.3 Number of Trees

In Figure A.3 we can see the behaviour of the **RMSE** depending on the number of Trees. The evaluation shows a surprisingly low value for 10 Trees. We will

further compare the values from here in the fitted Q-iteration part to see how it influences the learning progress. Generally it can be said that the RMSE falls further with a higher number of trees.

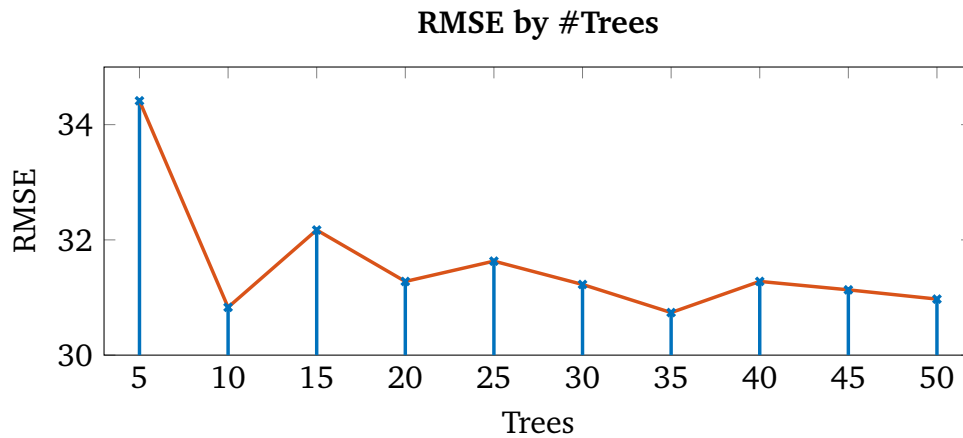


Figure A.3.: The RMSE falls further the higher the number of trees is.

A.1.4 Test width

Here we evaluate the RMSE for different Test widths used for representation where we sample the center of the test and create a box around it by going T_{width} into every direction. This creates a lot of equal sized tests and has proven to perform well on the actual learning task.

As Figure A.4 shows, we get the lowest RMSE at 2.5 T_{width} on the dataset collected in the first step. Considering that this test is 5×5 units big while the whole game is only 20×20 the low RMSE is surprising. We will further evaluate the behaviour of this test width in the actual application of the algorithm.

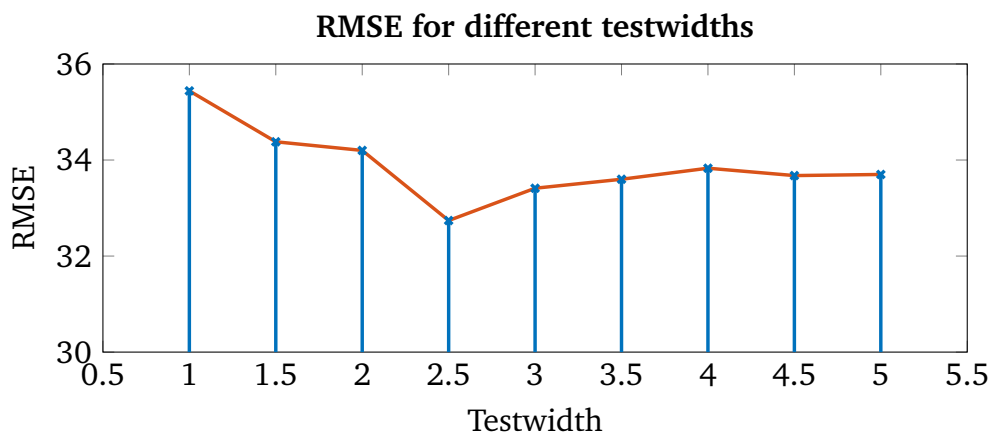


Figure A.4.: A test width of 2.5 results in the lowest RMSE.

A.1.5 Grid size

The evaluation shown in Figure A.5 suggests that using a grid size of 16 gives us the lowest RMSE overall.

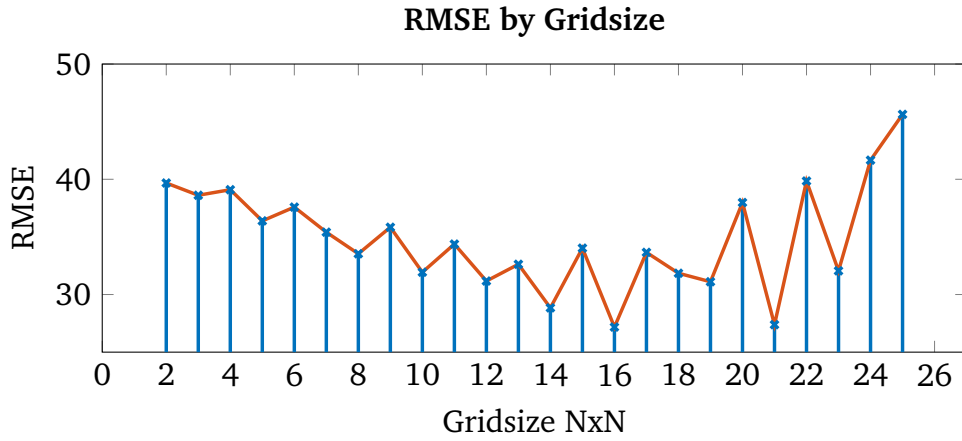


Figure A.5.: Using different Grid sizes for the linear ridge regression results in different Errors. A size of 16×16 has the lowest RMSE.