ARL Final Report

Egger, Maximilian — Pelat, Guillaume — Pilger, Laura

July 8, 2021

Abstract

In the past decades, Reinforcement Learning (RL) techniques have been tested and improved with the help of video games. The goal of this project was to train an agent to play a simplified version of Haxball in order to develop our RL skills. Aiming not only at obtaining the best agent, but also to get the fastest training possible, we present in this work a comparison of two classic algorithms, Dynamic Programming and Q-Learning. In addition, we also focus on two improvement strategies of Q-learning while investigating different approaches for feature and reward function design.

1 Introduction

Haxball is a physics-based simulation of football released for the first time in December 2010. Since then, this team-based game has become one of the targets for reinforcement learning adepts, allowing them to hone their skills in a completely deterministic environment. In this project, we tackled a simple version of Haxball, with only one player in the field and the aim of scoring as much as possible in the defenseless opponent's goal. Ball and player are initialized with random positions, but at each score, the ball is repositioned in the center of the field.

Because the goal of this project is not to spend most of our time trying different hyperparameters, we had only one restriction set: not using any Deep-Learning based components. We instead focused on designing a set of features and a reward function, and then tried out different algorithms in order to get not only the best Haxball player, but also one that learns the fastest possible.

This task presents two main challenges. We deal with a continuous state space and need to find a good way to represent it. Furthermore, there are a lot of bad actions the agent can take at each step, hence the chances of discovering a good policy are very low without a carefully designed reward function.

During this work, we intended to compare the results obtained by using Dynamic Programming and Q-Learning. Our first steps were to design features and rewards that we could actually apply our algorithms on. This can be found in sections 2 and 3, respectively. Then, we decided to challenge the classical approaches and experimented with two ways to improve the efficiency of Q-Learning. We tried each of them for 50 to 2000 epochs, each one being comprised of 1.000 trajectories with 10.000 steps each. All results obtained with an algorithm are described in their respective sections, and then summarized and analysed in section 8.

2 Feature Design

In the Haxball environment, the states describing the dynamic changes of the environment consist of six continuous variables. The agent and the ball position are each represented by two variables (x, y) describing the respective positions on the field, where $x \in [-4, 4]$ and $y \in [-2, 2]$. The ball velocity is described by two orthogonal variables (v_x, v_y), each bounded by ± 0.225. Stationary environment descriptions contain the goal positions and the size of the player, the ball and the field. One of the key challenges in our project is now to find a suitable feature space that efficiently reduces the size of this problem, but still provides the agent with all the information it needs to succeed in its task. During our project, we therefore designed different feature sets, which are described in the following subsections.

2.1 Initial Feature Set

Purely discretizing this continuous state space requires a proper precision management and results in a huge amount of low-level feature combinations which in turn complicate the agent's learning and decision process. In order to condense the available information and thereby reduce the state space to a level being sufficient for the agent's decision making, we therefore firstly tried the following encoded high-level features:

- A boolean describing the ability to shoot, resulting from the player and the ball position combined with their sizes.
- Two two-dimensional ternary variables describing the direction from the player to the ball in terms of compass directions (horizontal: W, 0, E; vertical: S, 0, N), i.e., $x, y \in \{-1, 0, 1\}$ for both the current and an estimation of the future ball position, i.e. an estimation $p_{ball}(t+1)$ given the current velocity.
- One integer describing the discretized angle from the player towards the goal, in steps of $10°$.
- A boolean that indicates in which half of the field the player currently is.

This initial feature set is visualised in Figure 1a. These features transform the external view on the environment to a player's view while discretizing the information of interest. In the above case, this results in $2 \cdot 3^4 \cdot 18^2 = 52488$ possible combinations. However, this feature set has shown to be insufficient, as the agent does not have enough awareness where it is on the field. This lack of information makes this approach unscalable, such that, for example, introducing an opponent would not be possible. Furthermore, the discretized angle to the goal would restrict the agent to direct kicks towards the goals, which crosses out the vast amount of strategies that score indirectly via the walls.

2.2 Discretized Feature Sets

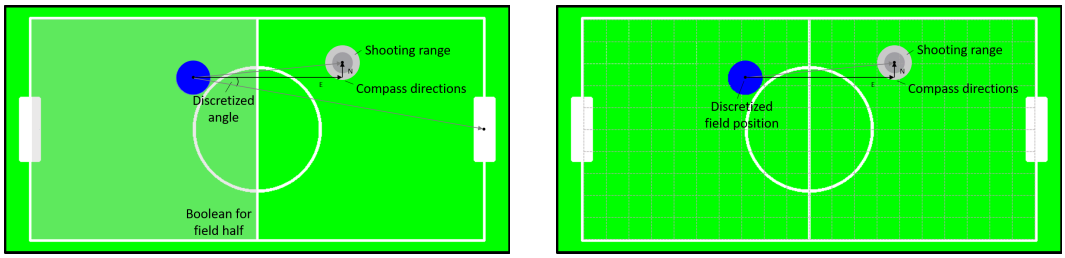
For the reasons written above, we decided that we need to discretize the field into a small, but sufficient grid. Therefore, we designed two additional feature sets, one that discretizes only the player position and one that additionally takes the discretized values for the ball positions.

2.2.1 Discretized Player Position, Relative Ball Position

In the first one, we only discretized the x and y position of the player. In addition, we removed the boolean describing the side of the field in which the agent is located in and the angle relative to the right goal. Our newly crafted feature set therefore sums up to:

- A boolean describing the ability to shoot, resulting from the player and the ball position combined with their sizes.
- Two two-dimensional ternary variables describing the direction from the player to the ball in terms of compass directions (horizontal: W, 0, E; vertical: S, 0, N), i.e., $x, y \in \{-1, 0, 1\}$ for both the current and an estimation of the future ball position. This requires both position vectors and the ball velocity.
- Two features describing the discretized agent's position, in both horizontal and vertical direction. Therefore, we divided the field into a grid of 20×10 subfields which corresponds to a resolution of 0.4.

In Figure 1b, the features of this set are depicted. The set eventually results in $2 \cdot 3^4 \cdot 20 \cdot 10 = 32400$ possible states which is an even smaller state space than the one described above.



(a) Initial feature set

(b) Discretized feature set 1

Figure 1: A visualisation of the two feature sets we mainly used.

Because we wanted to see if moving straight towards the ball was not causing issues, we also created a "target" point which can potentially be used instead of the ball position, defined by:

$$p_{target} = p_{ball} + 0.5 * \frac{p_{ball} - p_{goal}}{||p_{ball} - p_{goal}||}$$

In most cases we used the ball position directly, unless specified otherwise.

2.2.2 Discretized Player and Ball Position

Additionally, we tried to discretize the current and future ball position with the same resolution than described before. This second discretized feature set hence consists of:

- A boolean describing the ability to shoot, resulting from the player and the ball position combined with their sizes.

- Six features describing the discretized agent's, ball's and future ball's position, in both horizontal and vertical direction. Therefore, we divided the field into a grid of 20×10 subfields which corresponds to a resolution of 0.4.

This resulted in a huge state space, namely $(20 \cdot 10)^3 = 8.000.000$ states, but did not improve the performance particularly and was therefore not of main interest.

3 Reward Design

Just as important as the features are the rewards that the agent gets for executing a desirable behavior. Since the development objective is to enable an agent to score goals, it seems quite straightforward to design a reward function that rewards the agent every time it succeeds in this task. However, given the continuous state space and the set of different actions, it is unlikely that the agent actually achieves this by chance, as it has no idea or guidance on how to perform the preliminary steps for scoring a goal. Therefore, our reward function must also give incentives that will drive the agent to move closer to the ball and shoot in the right direction in order to enable learning. At first, we chose to use a reward with fixed values, each associated with a component:

1. $R_{ball} = 0.1$ for getting closer to the ball:

$$||p_{ball}(t+1) - p_{player}(t+1)|| < ||p_{ball}(t+1) - p_{player}(t)||$$

2. $R_{shoot} = 50$ for shooting in the direction of the goal
3. $R_{rgoal} = 100$ for scoring in the right goal

While designing this multi-part reward function, we had two options that we intended to compare:

1. The sequential approach that uses a reward function dependent on the training stage. At first, we would reward the agent for moving towards the ball, then change it to give a reward for each shot in the right direction, and later on for a goal.
2. The scaling approach that calculates the unweighted sum of the reward components and rewards all three actions right from the beginning of the training process.

The challenge for the first approach was to estimate the length needed for training before proceeding to the next stage. For the second one, the difficulty was that if the scaling was off, the agent would not learn the intended policy. During testing, the scaling approach proved to be efficient enough, so we gave up using the sequential approach at all. However, we found out that our approach with fixed values had three problems: firstly, if the incentive to aim for the right goal was not large enough, our agent would end up scoring equally in both goals. Adding a big punishment for this undesired action only seemed to deter scoring at all. Secondly, there was no incentive to use diagonal movements compared to moving along one axis, and thus the agent's movements could have been more efficient. Thirdly, the agent tended to wedge the ball into one corner.

Hence, we decided to use the sum of the following components as our new reward function:

1. Getting closer to the ball:

$$r_1 = R_{ball} * 2000 * ||p_{ball}(t+1) - p_{player}(t)|| - ||p_{ball}(t+1) - p_{player}(t+1)||$$

2. Getting the ball closer to the right goal:

$$r_2 = R_{ball} * 100 * ||p_{ball}(t) - p_{rgoal}|| - ||p_{ball}(t+1) - p_{rgoal}||$$

3. Staying at the left side of the ball:

$$r_3 = \begin{cases} R_{ball} * 0.3 * (p_{ball}(t+1)|_x - p_{player}(t+1)|_x) & \text{if } p_{ball}(t)|_x < p_{player}(t)|_x \\ 0 & \text{else.} \end{cases}$$

   Additionally, this component is reduced by 1 if abs $(p_{player}(t+1)|_y - p_{ball}(t+1)|_y) < 0.5$ to help the agent circumventing the ball. We will refer to this in the following as the 'corridor punishment', as it motivates the agent to stay out of the corridor on the right side of the ball.

4. The ball is wedged into the corner:

$$r_4 = -0.0001$$

5. Scoring in the right goal:

$$r_5 = R_{goal} = 100$$

6. Scoring in the wrong goal:

$$r_6 = -R_{goal} * 0.5 = 50$$

7. Neither the ball nor the agent is moving:

$$r_7 = -10$$

The scaling of the different reward components was done by visually observing the agent's performance and calculating the reward boundaries described in subsection 3.1. Although the agent still has trouble scoring, it is visually much better at moving and shooting roughly in the right direction. In particular, the addition of r_3 was probably the most efficient change to avoid shooting in the wrong goal, although not entirely removing this problem. Since the scaling approach is very sensitive to the interplay of the different rewards, the next subsection will give an impression about their value boundaries.

3.1 Reward boundaries

As the magnitudes of the rewards r_1, r_2 and r_3 are difficult to assess, it is useful to evaluate their possible value range. Therefore, one needs to think about the possible values for $p_{player}(t)$, $p_{player}(t+1)$, $p_{ball}(t)$ and $p_{ball}(t+1)$, as they are not fixed but can vary in its x-coordinate between $[-4, 4]$ and in its y-coordinate between $[-2, 2]$. Since there is only one time-step between those values, respectively, they are also limited in its difference to a maximum step size. This step size is set by the environment and can be calculated to:

$$\Delta_{max,player} = dt * v_{max,player} = \frac{0.075}{60} = 0.00125$$

$$\Delta_{max,ball} = dt * v_{max,ball} = \frac{3 * 0.075}{60} = 0.00375$$

With this in mind, the reward boundaries can be calculated to:

$$-0.335 \leq r_1 \leq 0.353$$

$$-0.046 \leq r_2 \leq 0.053$$

$$-0.24 \leq r_3 < 0 \text{ generally, and } -1.24 \leq r_3 < -1 \text{ in the corridor}$$

The value of r_1 depends on the angle between the agent's movement and the vector $p_{ball}(t+1) - p_{agent}(t)$. For r_2, the biggest value is given if $p_{ball}(t+1)$ is equal to the right goal position and the lowest if the ball is shot in either one of the left field corners. Since r_3 only functions as a punishment if $p_{player}(t)$ is bigger than $p_{ball}(t)$, this reward is bounded by zero and has its maximum when the ball is near the left wall and the player on the opposite side, touching the right wall.

Hence, the reward for scoring in the right goal is by far the biggest one, followed by the punishment for the wrong goal. Since scoring is the main objective of our implementation, this scaling is appropriate as the agent only gets this reward few times in a game and should not be motivated to learn a strategy that avoids kicking. In order to keep the game running, the punishment for being on the right side of the ball should be a bit smaller than the reward for going towards the ball, but in the same magnitude to still have an observable effect. The only exception here is when the ball is in the 'corridor' – as this induces high risk of an own goal and should therefore be discouraged.

4 Metrics

To investigate the performance of the model during the training phase, we decided to employ two metrics:

- **Mean of value function**
  This metric uses the mean of the value function to provide insights into the performance of the agent. Although influenced by values related to states with rare occurrences (which are often inaccurate during most of the training), this metric clearly highlights the convergence behavior of the algorithm.
  For Q-learning, this metric is calculated as

$$\mu_Q = \frac{1}{N_F} \sum_{i=0}^{N_F-1} max_a Q(i,a)$$

  and for dynamic programming as

$$\mu_{DP} = \frac{1}{N_F} \sum_{i=0}^{N_F-1} V(i),$$

  with N_F being the size of the feature space. Note that although this metric reflects how fast the agent adapts to the chosen rewards, it does not provide insights into how the agent actually behaves on the field.

- **Normalized cumulative reward**
  In theory, this is the most representative metric to use since it is directly calculated while

following a trajectory and thus reflecting the actual interaction between the environment and the agent's decisions. For every epoch e and trajectory t, the earned rewards $r(e,t,s)$ for every step s get summed up:

$$r_{\text{cumulative, norm}}(e) = \frac{\sum_{t=0}^{N_T-1}\sum_{s=0}^{N_S-1} r(e,t,s)}{N_T \cdot N_S},$$

with N_T being the amount of trajectories and N_S the number of steps for each trajectory. Since we did not achieve to make our agent prefer one goal over the other, this metric is always relatively stable and does not change significantly over time. Thus we were not able to use it as a measure for comparing the different algorithms.

5 Proposed Method 1: Dynamic Programming

Since we deal with a continuous state space in our task, it was crucial to discretize the initial state space to a much smaller feature space. However, this step adds a lot of uncertainty about whether the RL algorithm can still converge to an optimal solution. Hence, the initial plan was to use Dynamic Programming (DP) as a baseline in order to validate our choices for the feature space and the reward function. Since the loss in precision compared to the original state space is big with every chosen feature design, we did not expect to obtain an optimal policy that way. However, we hoped to gain a policy that is at least "not bad", especially compared to other techniques relying too heavily on a good exploration in order to converge.

Even though this plan holds in theory, we soon encountered some issue during the implementation. In each iteration, this algorithm should run once through the whole state space, or in our case, the feature space. With our setting, this is rather difficult to realize, since we would need to convert our feature space back to the state space in order to set the environment state appropriately. Due to the previously performed transformation in the state-feature mapping, the inverse operation, in turn, would add inaccuracy.

Therefore, we tried to solve the problem by implementing three different approaches.

At first, we tried applying a conventional recursive algorithm that tries to maximize the sum of all rewards given a certain number of steps. For every state, which is randomly chosen by resetting the environment, it iterates through every possible action and computes the maximizing action and value by recursively applying the algorithm for every subsequent state. Since the environment is deterministic, each state's value function only needs to be calculated once, as repeated calculations would always end up with similar results. Therefore, in order to enable a faster run time, we restricted the algorithm to only calculate a value for a state a single time. This resulted in an agent that was able to find the ball, shoot and even score goals in a very short amount of training time. Even though the agent behaved very well on certain positions, it also scored in its own goals regardless of the size of its punishment. A variation of the recursion length did not yield any improvements on that matter, however, our experiments were also bounded to a maximal recursion length of 20000 steps due to hardware (or more specific memory) restrictions. Nevertheless, due to its short training time, this approach turned out to be very useful to test our different features and reward sets in a reasonable amount of time. Figure 2 shows an exemplary result of this algorithm executed for 2000 epochs with the feature space described in subsubsection 2.2.1. It can be seen that the mean of the value function converged after this number of epochs. Please also note that carrying out the entire experiment did not even take one minute to be executed, which is extremely fast.

Next, we wanted to try out two slightly modified approaches of Dynamic Programming for which we exploited two properties of our environment: firstly, resetting the environment leads to a random state and secondly, the resetting to a certain state is always possible. The idea was to cover all states often enough randomly by resetting the environment to ensure convergence. Therefore, for each randomly chosen state, we iterated through every action, calculated the Bellman operator and saved the maximizing action and value. In practice, however, this algorithm took a very long time to visit all states and was therefore not able to converge in a reasonable amount of time.

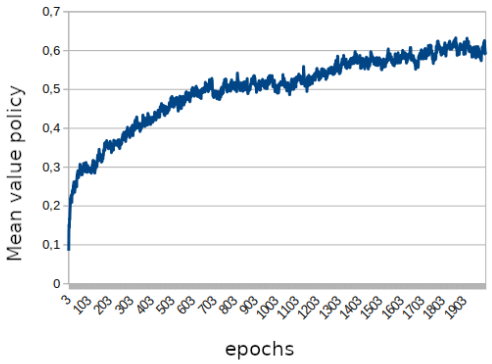


Figure 2: Mean value function per epoch when training with player position discretized and relative target direction for the approaching agent.

To speed up the convergence behavior, we implemented an algorithm that saves a value for each state-action pair and uses a huge amount of short trajectories for learning. This set-up would allow us to initialize with a random policy and a randomly chosen starting state, calculate the Bellman operator for every state in the trajectory based on the current policy and change the values accordingly. Since this modification of DP became rather close to Q-learning, we then decided that we will give up on DP for this task and focus our efforts on Q-learning.

6 Proposed Method 2: Naive Q-Learning

In parallel of the Dynamic Programming approach, we trained a second agent with Q-Learning. The first version used the basic implementation of what was seen in class, but a notable improvement was the implementation of an eligibility trace: while each epoch is slightly longer, the overall training time was greatly reduced.

We decided to keep the eligibility weights only if they are greater than $1e^{-2}$, which with an intermediate λ (according to the examples in Sutton & Barto's book [1], $\lambda \approx 0.7$ seems to work best) leads to $\sim$ 20 coefficient updates in the whole Q-matrix per iteration. However, as the reward for scoring is usually obtained multiple steps after shooting, we decided to keep two separate eligibility traces: one is updated continuously, and the second one is a copy of the state of the eligibility matrix when the agent last touched the ball. This way, instead of rewarding the agent for whatever he is doing while the ball travels towards the goal, and wait for the algorithm to back propagate it, we directly reward the steps that lead to the action causing the goal.

Another aspect we spent time working on was the exploration-exploitation dilemma, a well-known problem in Reinforcement Learning for which multiple strategies exist. We intended to try both Boltzmann Selection and ϵ-greedy strategies. But as the first one is much slower, we focused on the second one.

With an ϵ-greedy action selection, the agent chooses the action with the (state, action) pair yielding the highest return, unless a random value is smaller than ϵ in which case it chooses a random action. Since after more training the agent should have a better idea of what the optimal strategy looks like, the value of ϵ should decrease with time. In our experiments, we used $\epsilon = \frac{1}{2i}$, where i is the current epoch. During some experiments, in order to avoid being stuck in an

effective but suboptimal path, we tried to raise this value artificially to 0.2 every five epochs without notable effect on the final policy.

A common problem we faced during our experiments was "oscillations", a phenomenon happening when two neighboring states had opposite actions, resulting in the agent being stuck at the boundary. This was especially true when using a discretization of the field as our features, but occurred with various degrees of intensity in all of our experiments. At first, we believed that this would be solved with more training, but Figure 3 proved us wrong. We then tried to solve this using a new reward component $r = ||\pi(p_{agent}(t) - \pi(p_{ball}(t+1))|| - ||\pi(p_{agent}(t+1) - \pi(p_{ball}(t+1))||$, where $\pi : \mathbb{R}^2 \to \mathbb{N}^2$ is the projection from the real coordinates to the discretized features, but without success.

Finally, what solved the problem was a modification on the way we applied greedy actions: instead of applying a random action once in a while, which most likely did not really change the state of the agent, we applied the same action for 30 steps. This modification, however, was only found with the improvements we applied to the naive Q-learning described in the next section.

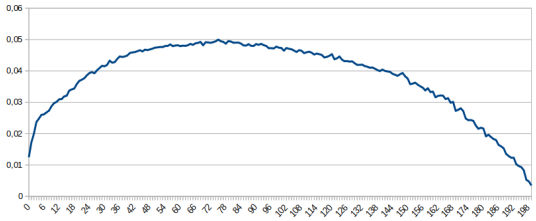


Figure 3: Mean policy value per epoch for an agent using a fully discretized state space. The increase in oscillation cases is clearly visible starting from epoch 80.

With this algorithm, we achieved quite similar results with our different sets of features. Most of our agents did not learn to differentiate the two goals before being plagued by the oscillation curse, although the discretized ones seemed better at scoring quickly that with our initial features. Our best result was achieved with a 10×5 full discretization, which seemed to be the only one actually preferring the right goal; however, due to the low precision of the discretization, it often missed the goal by a small margin and overall did not score as often as its predecessors.

7 Proposed Method 3: Improved Q-Learning

In order to improve our policy further, we decided to advance our Q-learning algorithm by implementing a situational-based agent that can change between policies depending on the field situation and by performing a non-random initialization of our Q-matrix. These two approaches are described in the following subsections.

7.1 Situation-Based Agent

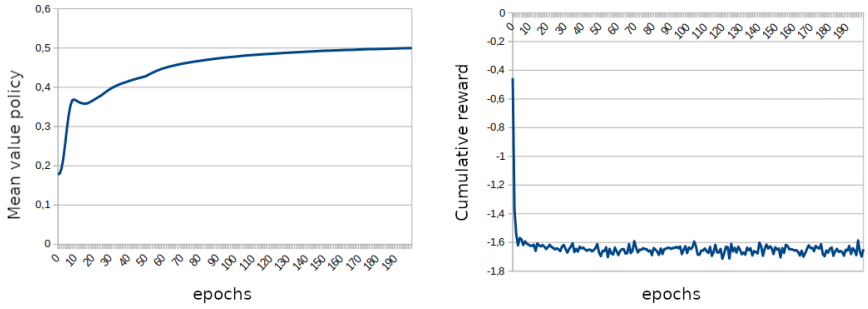
The idea behind the situational-based Q-learning agent inspired by [2] is that we train two agents with two separate Q-matrices, which take turns depending on the field situation. The first agent has the task of approaching the ball, while the second agent is responsible for shooting. A half circle around the left side of the ball position marked the transition line for these two agents.

In theory, this newly introduced separation should have some advantages. Firstly, it leads to a reduction of both necessary features and actions for the approaching agent. Secondly, it allows us to design the rewards for the approaching agent in a way that it does not take the future ball position as its reference for rewarding, but a point to its left in order to minimize the chance of

an own goal. Also, rewards for shooting towards the goal and into the goal are omitted for the approaching agent. These rewards should motivate the behavior of prioritizing the adjustment for a better kick starting position over kicking as soon as the agent comes in shooting range. Additionally, designing the weights of the reward components for the approaching agent is easier since we here use less components as for the naive TD agent.

In the beginning, we tried to implement the approaching agent's feature space by only giving the compass direction of where it should head, namely North/South and East/West relating to the current and the future ball position. As described above, the reference point for rewarding thereby was the target position p_{target} introduced in subsubsection 2.2.2. This led to a tremendous reduction of the state space, going from 32400 or even 8000000 states (depending on the discretization) to 81, but was failing due to the unawareness of the agent where it is on the field and thus preventing it from learning more advanced strategies.

In order to fix this issue, we then decided to also use the discretized x and y positions of the agent as a feature with the above mentioned accuracy of 0.4 together with the relative directions towards the target position resulting in a total of 1800 states. Compared to the feature set described in section 2.2.1, we basically removed the boolean describing the ability to shoot and the relative directions towards the current ball position for the approaching agent to simplify the feature space. The feature space of the shooting agent remained unchanged and therefore summed up to 32400 in total. This enabled the agent to learn advanced strategies to approach the ball by being aware of its own approximate position on the field. The results are shown in Figure 4. The mean of the value function in Figure 4a shows that the algorithm converges at around epoch 30 to a value of 0.5. The cumulative reward in Figure 4b stabilizes very fast to a value around -1.6 and remains unchanged. This shows that even if the policy is improving with respect to the chosen rewards and thereby the agent is learning to more efficiently move towards the ball and scoring goals, it does not learn to aim for the right goal only. Therefore, the cumulative reward does not improve over time.

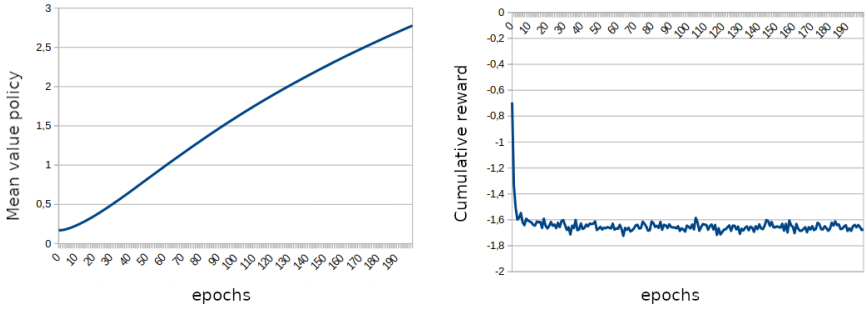


(a) Mean value function (b) Cumulative reward

Figure 4: Metrics per epoch when training with player position discretized and relative target direction for the approaching agent.

Additionally, we tried out employing a fully discretized feature set consisting of the player position and the target position derived from the future ball position. Basically, this is the feature set described in section 2.2.2 without the boolean and the discretized current ball position using the target position instead of the future ball position. This results in a feature space size of 40000. The outcome of one of the experiments can be found in Figure 5. By comparing the values in Figure 4a with Figure 5a, it gets clear that the fully discretized feature space results in a more

improved policy. With the same settings, the values improve much faster and do not converge after already 200 epochs as seen in Figure 5a. This means that there is still space left for further improvement, while the partly discretized state space reached its full potential after 200 epochs already. The plot for the cumulative reward in Figure 5b looks exactly the same as for the above feature space, stressing even more the fact that the agents do learn how to efficiently move towards the ball, but not to prefer only the opponent's goal.



(a) Mean value function (b) Cumulative reward

Figure 5: Metrics per epoch when training with player and future ball position discretized.

As a next step, we mainly focused on when to switch between the situations. One major issue thereby was to ensure that each agent has the chance to observe the effects of its chosen actions. The goal reward is a good example for this problem: It is quite obvious that this is one of the main rewards that the shooting agent need to adjust to the desired behavior. However, even though the shooting agent is in charge while kicking, the reward for this action, the goal, will only be given several steps later – probably to a time where the approaching agent took its turn again as the ball is not in shooting range anymore. In order to tackle this issue, we made sure that the shooting agent stays in charge for a fixed amount of time steps after it had shot the ball. After testing different number of steps and approximating the steps the ball need to travel over the field, we decided to use 300 time steps until switching back to the approaching agent. As we did not want to stop the shooting agent from performing desirable actions during this time, we gave him the same rewards than we would have given the approaching agent. However, this raises the problem that depending on if and when a goal was scored after shooting, the shooting agent is probably still in charge of the actions and therefore the advantages of using a simplified version of the features and rewards for approaching the ball get relativized. During testing, it turned out that this problem significantly complicates the learning process and reduces the advantages of this agent.
To account for this problem, one could for example also switch from the approaching agent to the shooting agent when reaching shooting range and only back to the approaching agent when a goal was shot. This, however, raises the problem that the shooting agent might be active forever if it did not shoot into any goal. Consequently, it is reasonable to use a combination of both approaches: Switching back from shooting agent to approaching agent if either a goal was shot or a predefined number of steps was reached after getting in shooting range.

Our next goal was to motivate the agent to circle the ball until it is in a suitable position to shoot towards the right goal. Therefore, the reward for going towards the ball, which is necessary to keep the agent moving, must be smaller than the punishment for approaching the ball from the right side. Thereby, especially an approach from the 'corridor', which was further described in

section 3, must be punished as it increases the probability of an own goal. This trade-off required some scaling, but we eventually found values that led to the desired behavior. Nevertheless, with this reward setting, the agent tend to push the ball into the right corners. We successfully counteracted this behavior by giving a small punishment if the ball is located in either ones of the corners.

7.2 Q-Matrix Initialization

During all of our previously presented work, we mostly focused on the algorithms and features used, but there is also one element which can greatly improve the training speed: a good initialization of the Q-matrix. During our first experiments, filling the Q-matrix with 0 was better than a random initialization when using Boltzmann selection. Hence, when we went back to an ϵ-greedy policy, we kept it that way, only penalizing actions 0 and 1 (respectively "no move" and "no move + shoot") because our reward function was working better when correcting a wrong move. Initial weights are very important, because with the right initialization, we are already close to the optimal policy.

However, this usually requires some kind of expert knowledge, and with a big number of states this is not really feasible to tune every weight by hand. Instead, we decided to try guiding our agent during a few rounds at the beginning of the training, before letting him learn by himself like in our previous tries. Previous work has shown that even a low number of human demonstrations is sufficient to impact the training efficiency [3], so we tried modifying the Haxball environment provided to modify the Q-matrix of the agent using actions from the keyboard input instead of those dictated by our policy, but the process was tedious and needed a relatively long human intervention at the beginning of each training.

Instead, we developed a simple deterministic "master" agent with a few guidelines.

- If we are directly at the right of the ball, move up or down.
- If we are at the right of the ball, move left.
- If we are at the left of the ball, choose the action bringing us the closest to the target point.
- If we are close to the target, shoot.

It is worth noting that this player, although completely deterministic, already outperformed most, if not all of our previous experiments. However, it is not able to handle the case were the ball is close to a wall, and will often get the ball stuck in a corner if given the opportunity. This flaw was intentionally not corrected, as it would show that the agent trained with the master was able to surpass it.

From the design of this master, we could derive a new list of features:

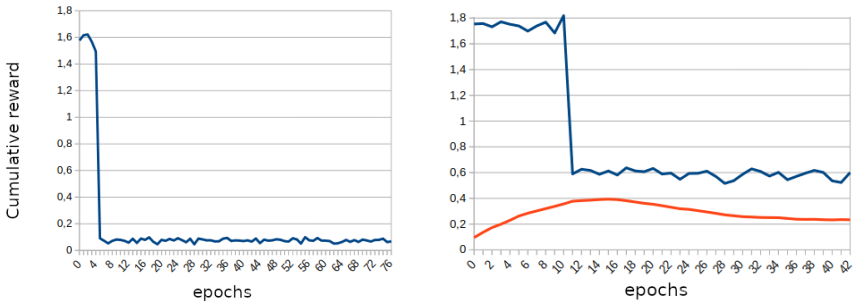
- A boolean indicating whether we are left or right of the ball,
- an integer describing our relative vertical position to the ball (from -2 to +2),
- compass direction leading to the target,
- a boolean indicating whether we are close enough to the target to shoot,
- two booleans indicating if the ball is close to a wall on the x/y axis,
- two booleans indicating if the agent is close to a wall on the x/y axis.

It is worth noting that if we restrict the agent to its first four features, it is enough to completely replicate the behaviour of the master after only two epochs of training.

Then the question of how to use this master during training arises. We considered using a new reward function:

$$r(a_{agent}, a_{master}) = \begin{cases} 1 & \text{if } a_{agent} = a_{master} \\ -1 & \text{else.} \end{cases}$$

However, we wanted to keep the same reward function during the whole training. We then thought of using the policy of the master during the first few epochs. This time, because we have an agent that is already able to move and score, we can give a reward only for scoring, which removes any bias due to an improper scaling of our reward components. However, this was seemingly not enough, as the agent soon forgets everything he learned. Figure 6a shows this for five epochs, but we tried up to 30 with similar results.



(a) Averaged reward per epoch. After five epochs, the agent is left alone.

(b) Cumulative reward (blue) and mean value function per epoch (orange). After ten epochs the influence of the master decreases linearly.

Figure 6: Results of two training strategies. In both cases, only goals are rewarded. Training was stopped earlier than 100 epochs due to lack of improvements.

Another approach was using the greedy exploration: our agent would train normally, but instead of choosing random actions from time to time this action would be dictated by the master during the first 30 epochs. During this time, the exploration should be only performed thanks to the coefficients present in the randomized initialization of the Q-matrix, and once these 30 epochs are down, then truly random actions are chosen. Unfortunately, similar results as before were achieved.

During our last try, we went back to our first approach. The agent would use the actions dictated by the master to update its policy, but as the training progresses, it would take more and more decisions by himself with the proportion of actions chosen by the master decreasing linearly to reach 0 at the end of the training. The training curves for this method can be seen in Figure 6b.

8 Discussion

In the last sections, we presented our experiments and their results. In this section, we wanted to make a review of what we learned from the feature and reward design, and perform a comparison of the algorithms we used.

8.1 Feature Design

The big question we had while designing our features was how to transmit the information about the ball position to the agent. Should we give him the position of the ball and let him decide how to approach it, or should we give him a hint about where to place himself relative to the ball? This led to the design of the "target" point, which is the ideal place from where to shoot in order to score. This generally yielded better results than just giving the ball position, as the only difficulty was in how to approach the ball, and not what to do once the agent is in the vicinity of the ball. However, this point is much harder to define when there is an opponent, and this is the main reason why we did not generalize its use over all our experiments.

In section 2, we introduced different feature sets: one consisting of only relative positions, one in which the player position is discretized but the current and future ball positions are relative, and a fully discretized feature space. For every algorithm, we investigated the difference of these feature sets. While the first set has proven to be usable for learning basic strategies in a short amount of time, no advanced strategies were learned by the agents. This drawback was addressed by discretizing the player's position on the field. As the agent is aware of its position, the policies obtained were superior to the ones obtained by the more basic feature set. Discretizing not only the player position but also the current and future ball position generally enables even more advanced policies. However, this approach requires much more computational effort to train resulting from the increased feature space and is therefore only useful during the reward function design, which we were working on until the end, to some extend. Additionally, designing the discretization steps is a crucial step. We decided for a step size of 0.4, i.e., a grid consisting of 20×10 subfields as a trade of between loss of information and reduction of feature dimension. This has shown to be a reasonable approximation, as an agent trained with 10×5 often missed goals with a short margin, while higher number of subfields led to extremely slow improvements during training.

Finally, an interesting question is if it really makes a difference to give the agent access to both the ball position and its future position. Indeed, in most, if not all feature sets we considered, the ball speed is too low to induce a big change: even with a 0.4 space discretization, the ball and the future ball are usually both in the same cell. As our intuition tells us that both are important, we kept using the two positions; however, perhaps focusing on the ball's velocity direction instead of its future position would give different results.

8.2 Reward Design

A very frequent and undesirable situation happens when the ball is stuck in the corner with the agent blocking its path. To avoid such situations, we added a small punishment for the agent when the ball is near the corners. This modification almost completely eliminated this problem without unwanted side-effects.

Another component which took time designing was the punishment if the agent is to the right of the ball. The idea behind it is simple: in this situation, any interaction with the ball is much more likely to send it in the left goal than the right one. However, since the quickest way to get out of the wrong side of the field is a horizontal line, the agent ended up hitting the ball on its path and scored in its own goal more than ever. To avoid this, we decided to add the 'corridor punishment'. At first, we doubled the punishment if $\text{abs}\left(p_{player}(t+1)|_y - p_{ball}(t+1)|_y\right) < 0.5$, but this seemed to not be enough, as this component decreases as the player gets closer to the ball and gets overwritten by the "get closer to the ball" component when it needed the most to

avoid approaching the ball from the right. For this reason, this is one of the three cases where we went for a constant value punishment, which ensures being always the strongest one among all reward components.

However, the problem that the agent does not only aim for the right goal was not eliminated. Although shooting into the right goal is rewarded and hitting the left goal is punished, the agent does not only aim for the right goal in most of the training processes. This is most probably related to an improper scaling of the reward components. In particular, the interaction between the reward for a goal and the punishment of being right of the ball makes it difficult to balance. As soon as the agent shoots into the right goal while being on the right side of the field, the ball gets reset. Consequently, the agent gets punished for being right of the ball, which in turn prevents the agent from hitting the right goal. We tried to circumvent this problem by clipping the position dependent punishment for being on the right side of the ball to a relatively low absolute value and thereby making it more attractive to aim for the opponents goal. Although the agent then did not shoot into the left goal intentionally anymore, it still hits the ball in such a way that it ends in the left goal. Also, increasing the goal reward by different magnitudes did not result in an improved behavior. While the absolute values of the metrics we employed changed, the agent is still not clearly preferring one goal over the other. This is also the reason why the cumulative rewards stay at a relatively low level. The agent equally shoots into both goals while being punished for most of the actions on the field related to its movement.

8.3 Algorithms

The main conceptual question we faced during this project was "how long do we wait before discarding an experiment?". During our experiments, we saw that our metrics could decrease for more than twenty epochs before going up again. On the opposite, Figure 3 shows that problems can occur much later. There is obviously no consensus on this among the RL community, as the optimal value is at once task-, feature- and reward-dependent. In this project, we settled with at least 50 epochs as our minimum standard training duration, which already took more than one hour on our devices.

The training time constraint we fixed is probably most visible in the training of the agent using a master. Indeed, as we wanted this agent to learn some things by himself, we decided to restrain the use of the master to at most 30% of the total training time. However, the moment we stop using the master is clearly visible on the training curves, and can have two origins: either the reward function we use is unadapted to the policy taught by the master, or these thirty epochs are simply not enough at all. To eliminate the first hypothesis, we decided to only reward the goals scored. This might have had a deleterious effect on the training time needed; however, since this is an idea that we got very late, we did not have the opportunity to train for longer times. However, we are still convinced that this approach is better than naive Q-Learning: when limiting the agent to the first four features described in subsection 7.2, the agent that trained with the master was scoring more goals after one epoch than the one trained without after ten epochs.

In sections 6 and 7.1, we introduced the results for naive Q-learning and situation-based Q-learning, respectively. Especially on a first glance, the results obtained by situation-based Q-learning were very promising. However, after several training and testing iterations it got clear that this approach is lacking some important information. One of the most challenging aspects is the missing reward propagation between the different agents. While the approaching agent is not aware of the impact of its decisions on the overall goal of shooting goals efficiently, the shooting agent does not have the possibility to learn advanced strategies since being constrained

on the positions obtained by the approaching agent. Additionally, both agents do not have the possibility to interplay in order to converge to well-suited and efficient policies.

In addition to that, this environment contains quite a number of situations in which the approaching agent reaches its limits. This is for example the case if the ball is on the left end of the field. In this case, it does not make any sense trying to get left of the ball. Consequently, also switching properly between the two agent fails, leading to undesired behavior which shows that the agent is not capable of interacting with the entire environment to reach its ultimate goal, but only sub-goals.

Overall, using a situation-based Q-learning approach can simplify a lot the design of features and rewards and therefore lead to good results in a reasonable amount of time. However, especially for this kind of environment, there exist to many edge cases which make this strategy fail completely while preventing the agent to learn advanced strategies for shooting goals. That is why naive Q-learning, although being more computationally expensive to train and more involved to design rewards for, turned out to be the preferred approach for this kind of problem.

As conventional dynamic programming is not easily implementable for this scenario since it would require inverting the discretization steps and therefore creating statistical uncertainty, we described a recursive algorithm to maximize the expected rewards over time by choosing the best possible actions in section 5. Compared to Q-learning, this algorithm converges relatively fast as once a value is calculated for one state using the predefined recursion depth, this value remains constant and is not recalculated. Compared to Q-learning, this approach does not raise the problem of oscillating between two actions and provides a relatively good baseline model to compare with. Especially, due to the fast convergence, it is particularly interesting for testing feature sets and reward components. The huge drawback of this approach is the exponential computational complexity when it comes to large feature spaces, action spaces or when the problem requires a huge recursion depth.

9 Conclusion

To sum up, with none of the approaches the agents were able to efficiently shoot into the right goal while avoiding own goals. Hypothetically, the feature and/or reward design should lead to this behavior. These were the parts that were investigated the most. Especially, the design of the different reward components and their respective weighting was of interest. In order to cope with the main problem of shooting own goals, we added new reward components. This, in turn, complicated the reward design itself, i.e., the scaling of the single components. In order to gain a detailed understanding of the reasons and to find a systematic way of determining feasible weightings, we investigated in detail the bounds of the reward components. While this provided better insights, we were still not able to design the rewards in a way such that the agents purely aim for the right goal. Using only a reward for scoring into the right goal without additional components led to an agent that was not even able to approach the ball, also after a significant time of training.

While not considering edge cases in which, e.g., the ball is in one of the left corners of the field, the situation-based Q-learning agent was the most promising. Depending on the actual feature space and the reward components used, the agents were able to approach the ball from the left and therefore developing a relatively good policy. On the other hand, this agent was not able to successfully handle borderline situations at all. Since this is a major problem, we decided to focus on naive Q-learning and would generally state this as being the most promising, although

the employed reward might not be optimal yet.

Of course, there is still room for improvement. However, as we tried most of what we though of in the time allocated to this project, in order to decide what to do next we need to diagnose what was exactly the problem. To do this, we would need to work on the reward scaling between our components. Until now, we tried to make sure that their boundaries were all at the same order of magnitude, except for the goal reward which were always kept much higher, but perhaps their distribution is such that weighting some more would have helped. In order to do this, we should try investigating with a much more precise discretization resolution, which could additionally enable more complicated strategies. As it would require a huge amount of time, we focused on other approaches, but depending on the results this would help diagnose which one, from the features or the rewards, needs to be changed. However, once this is done, we are confident that the improved Q-learning methods proposed in section 7 will yield better results.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 2018.

[2] N. K. Ure, M. U. Yavas, A. Alizadeh, and C. Kurtulus, "Enhancing situational awareness and performance of adaptive cruise control through model predictive control and deep reinforcement learning," in *2019 IEEE Intelligent Vehicles Symposium (IV)*, pp. 626–631, 2019.

[3] G. V. de la Cruz Jr, Y. Du, and M. E. Taylor, "Pre-training neural networks with human demonstrations for deep reinforcement learning," 2019.