Fundamentals of Machine Learning

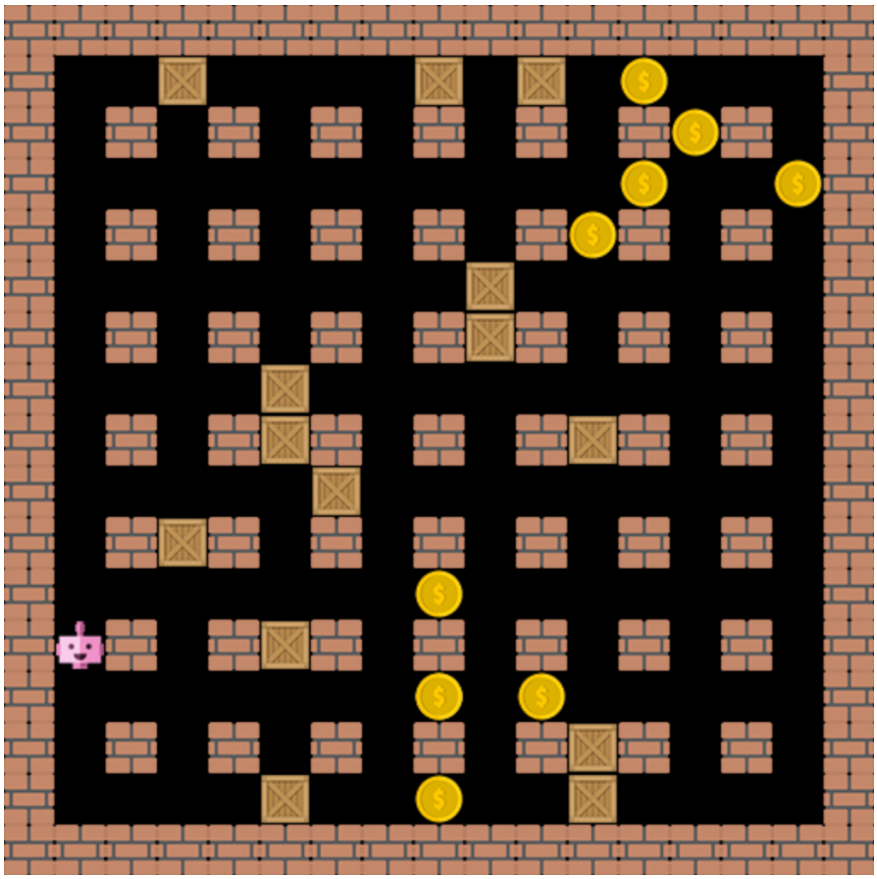
Final Project - Report

Reinforcement Learning for Bomberman

`https://github.com/hericks/FML`

Alessandro Motta, Matthias Hericks, Mika Rother

March 29, 2021



Contents

Chapter 1

Learning methods and design choices

1.1 Learning methods

by Matthias Hericks

1.1.1 Generalized policy iteration

In their most abstract form, all of the reinforcement learning methods we deployed during this project can be described as *generalized policy iteration*. The key components of generalized policy iteration are two interacting processes. The *policy iteration* process, makes a value function v or action-value function q increasingly consistent with the policy π currently deployed. This computation of v_π and q_π for an arbitrary but fixed policy π is also called the *prediction problem*. The second main part of generalized policy iteration is the *policy improvement* process. This process makes the policy π greedy with respect to the current value function v or action value function q, respectively. **?** introduced the term generalized policy iteration to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the exact details of these processes. Furthermore, Sutton describes that generalized policy iterations in many (theoretical) cases convergence to the optimal (action-) value function (q^*) v^* and optimal policy π^* is guaranteed as long as the process continues to update all states.

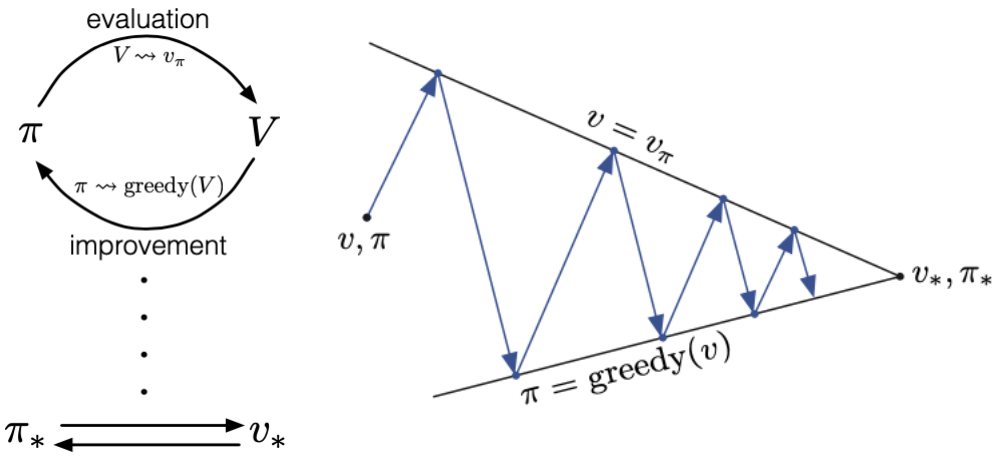


Figure 1.1: Generalized policy iteration as described in **?**.

This generalized policy iteration is one possible solution to the *control problem* (in contrast to the prediction problem), that is the relevant problem of finding a good policy.

The policy improvement step is typically done by making the policy π greedy with respect to the most recent version of the value function v or action value function q. During the project, we relied on an action-value function q, since in this case no model of the environment's dynamics is required to construct the greedy policy

$$\pi(s) := \arg\max_{a \in \mathcal{A}} q(s, a). \tag{1.1}$$

For the evaluation or prediction part of the generalized policy iteration, we relied on on-policy *Temporal Difference* learning. Just like Monte Carlo methods, Temporal Difference methods are model free and allow for learning directly from raw experience without a model of the world. In contrast to Monte Carlo methods, Temporal Difference methods allow for *bootstrapping*, that is, they can update estimates based in part on other learned estimates, without waiting for a final outcome. This was important for the Bomberman example, since the agent should not be forced to wait till the end of a round for an update.

1.1.2 Sarsa algorithm

The first Temporal Difference learning method, we used was the *Sarsa algorithm*, which updates after every transition from a nonterminal state.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big], \tag{1.2}$$

where

- S_t denotes the initial state of the state transition,
- A_t denotes the action taken in the transition,
- R_{t+1} denotes the reward the agent received for the transition.

$Q(S_t, A_t)$ denotes the value estimate of the state-action-tuple (S_t, A_t) and is defined to be zero for all tuples (S_t, A_t), where S_t is terminal.

- S_{t+1} denotes the state of the agent after the state transition,
- A_{t+1} denotes the action the agent will take after the state transition. Note that the value estimation $Q(S_{t+1}, A_{t+1})$ is zero for S_{t+1} terminal, independently of the action A_{t+1}. Thus, no action is required for the evaluation of terminal states S_{t+1}.

Moreover, this first algorithm introduces two important hyperparameters,

- $\alpha > 0$ denotes the *learning rate* and controls the effect of a single
- $\gamma \in [0, 1]$ denotes the *discount factor*, which is a measure for the weight of future rewards.

The update rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. In this way, the quintuple gives rise to the name Sarsa for the algorithm. Sarsa was our initial choice for a learning algorithm since it has excellent convergence properties as proven in **?** and is easy to implement.

From the Sarsa prediction algorithm it is straightforward to construct an on-policy control method using the generalized policy iteration. To do so, we gradually estimate the action-value function q_π with the behaviour policy π and simultaneously let π greedily adapt to q_π as in (1.1). One of the frequently encountered conditions to guarantee the convergence of this iteration, is that all state-action pairs are in theory visited infinitely often. Without doubt, this is not possible in the large state space of the bomberman game. Still, exploration was necessary as easily seen in the classical example of the *exploration-exploitation-tradeoff*.

1.1.3 Exploration-exploitation-tradeoff

In the beginning of the project, we used ε-greedy policies to ensure exploration of unknown action-state combinations. For every policy action-value function q and $\varepsilon \in [0, 1]$ the corresponding ε-greedy policy selects a random action with probability ε and the action greedily proposed by q with probability $1 - \varepsilon$. Thus,

$$\varepsilon\text{-greedy}(q)(s) \sim \begin{cases} U(\mathcal{A}_s) & X = 1 \\ \arg\max_a q(s, a) & X = 0 \end{cases}, \tag{1.3}$$

where $U(\mathcal{A}_s)$ denotes the uniform distribution on the set of possible actions in the state s and $X \sim \text{Bin}(1, p)$ denotes a binomially-distributed random variable with success probability p.

In later stages of the project in which it was necessary for the agent to cautiously drop bombs to complete tasks, exploration with blind tentative ε-greedy policies was not practicable anymore. Therefore, we switched to softmax-policies to still guarantee exploration. For every action-value function q and *temperature* $\rho \in (0, \infty)$, the ρ-softmax policy selects actions in state s with varying probability according to the corresponding estimated value $q(s, a)$ of the state-action tuple (s, a). To be more precise,

$$\mathbb{P}(\rho\text{-softmax}(q)(s) = a) = \frac{\exp\left(\frac{q(s,a)}{\rho}\right)}{\sum_{a_i \in \mathcal{A}_s} \exp\left(\frac{q(s,a_i)}{\rho}\right)}.$$

Similar to ε for ε-greedy policies, the temperature ρ determines the extent of the probabilistic nature of the corresponding ρ-softmax-policy. For $\rho \to \infty$ the corresponding ρ-softmax policy converges to the uniform distribution $U(\mathcal{A}_s)$ on the set of possible actions $\mathcal{A}_s$. For $\rho \to 0$ the ρ-softmax policy selects the action with maximal estimated

corresponding value $q(s, a)$ with arbitrarily high probability.

A more detailed description of our decision-making process regarding the different exploration possibilities can be found in section **??**.

1.1.4 The tyranny of the single time step

The Sarsa algorithm performed really well to train some initial agents on the first task. Still, one major drawback of single time step methods are described by Sutton as the *tyranny of the single time step*. The tyranny of the single time step denotes the disadvantage of one-step method, such as Sarsa, that the update of an action and the amount of bootstrapping is linked. It is reasonable to update the action value function frequently to take into account anything that has changed. Unfortunately, bootstrapping works best over longer time periods in which a significant change in states has occurred. For one step methods the update intervals and bootstrapping time periods are equal and a compromise must be made. To defy the tyranny of the single time step and to enable fast training, we therefore switched to an n-step method with bootstrapping over multiple steps.

1.1.5 n-Step Sarsa

The simple idea behind n-step Sarsa is to modify the update rule (1.2) to update an earlier estimate based on how it differs from an estimate *after n-step*, instead of based on how it differs from an estimate after a single time step. This gives rise to the new update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[\sum_{i=1}^{n} \gamma^{i-1} R_{t+i} + \gamma^n Q(S_{t+n}, A_{t+n}) - Q(S_t, A_t)\big]. \tag{1.4}$$

The n-step Sarsa methods form a family of learning methods. Since Monte Carlo methods perform their update only after every game, they can be considered as an extreme case of n-step Sarsa methods with $n = T$, where T denotes the number of steps per game.

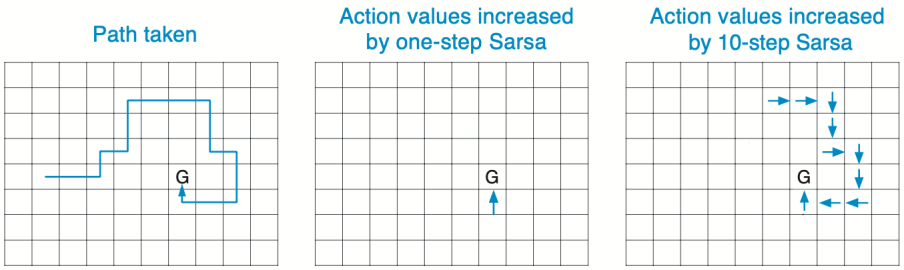


Figure 1.2: Visualisation of n-step Sarsa by **?**.

Controlling from which states bootstrapping occurs is important, because it affects the fundamental trade-off between bias and variance of updates. Special motivation for the upgrade of the Sarsa algorithm to the n-step Sarsa algorithm was provided **?**. We achieved the best training results with an intermediate value of n ($n \approx 4$), which illustrates how the abstraction of one step Temporal Difference and Monte Carlo methods to n-step learning methods can increase the performance of the two extreme methods. The results of our associated simulation study are in line with previous research by Sutton **??** and they can be found in section **??**.

1.1.6 Learning with eligibility traces

The last and final update to the learning algorithm was to upgrade the n-step Sarsa method to a generalized Sarsa(λ) algorithm. Just like n-step Sarsa, this new family of methods unifies Temporal Difference learning and Monte Carlo methods. Sutton describes the n-step approach that we have been taking so far as a forward view of a learning algorithm. For every state visited, the update algorithm looks n-steps forward in time to all the future rewards and punishments to determine the next update. The Sarsa(λ) methods reverse this view by using a short term memory, which stores information about the recently visited states and updates them accordingly. The results are similar to an n-step method, but moreover eligibility traces offer an elegant algorithmic mechanism with significant computational advantages. Instead of storing the last n state transition tuples $(S_t, A_t, S_{t+1}, R_{t+1})$, the Sarsa($\lambda$) algorithm stores an eligibility trace z of dimension d. When the action-value function is not approximated by a parametric function $\hat{q}_w$, d is equal to the product of the cardinality of the state space and the cardinality of the action space. This is infeasible for the Bomberman game with large state space. Still, we restrict ourselves to this theoretical case here to provide a clear explanation. In this setting we may index z by a state and an action $z_{s,a}$. $z_{s,a}$ serves as a short-term memory, which measures the weight of the state-action tuple (s, a) on the current state. Computationally, this is done by bumping up the component $z_{s,a}$, whenever the state-action pair (s, a) participates in producing an estimated value, e.g. action a was performed in state s. Then, the corresponding component $z_{s,a}$ fades away exponentially in time. When the agent receives a reward, all state action values $q(s, a)$ are updated, but the update is weighted by their corresponding component in z. Thus, larger updates will be performed for the tuples (s, a), which were visited just shortly before the reward occurred.

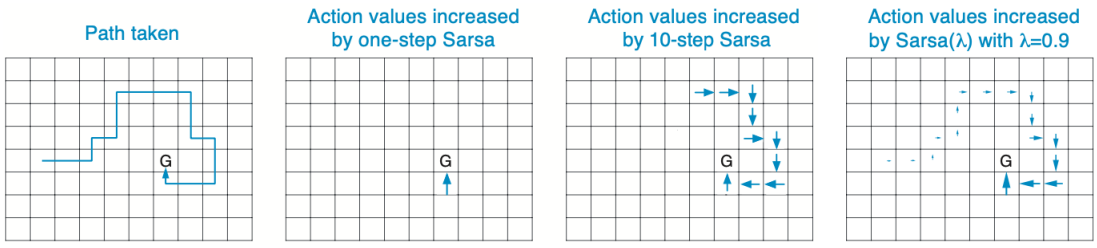


Figure 1.3: Visual comparison of Sarsa, n-step Sarsa and Sarsa(λ) by **?**.

The hyperparameter $\lambda \in [0, 1]$ determines how fast the components of the eligibility trace z fade away in time. The fading of the components is implemented as a multiplication of z with λ in each time step. In the first extreme case $\lambda = 0$, a new reward won't effect the action-value estimation of a state-action pair, which occurred more than a single time step ago. In particular, we have Sarsa(0) = 1-step Sarsa = Sarsa. The second extreme case $\lambda = 1$, yields the well-known Monte Carlo method, again. Thus, the family of learning methods Sarsa(λ) unifies single step Temporal Difference learning and Monte Carlo methods in a continuous way, in contrast to the discrete unification of the two families by n-step methods. Moreover, Sarsa(1) is applicable even in infinite non-episodic task, where the standard Monte Carlo method fails, since there is no point in time to perform the update.

In our case, the switch from n-step Sarsa to the Sarsa(λ) method did neither come with an increased training performance nor with a decreased training performance. Nevertheless, this was not clear beforehand. **?** provides studies, where Sarsa(λ) easily outperforms multi-step methods. In hindsight, we learned a lot while studying the theoretical basis for Sarsa(λ) and our final implementation is much more elegant and straight-forward than the implementation of the n-step variant.

1.2 Action-value function approximation *by Matthias Hericks*

In the previous section, no assumptions were made about the structure of the action-value function q and we implicitly considered tabular solution methods in their most general form. Because of the large feature space of the Bomberman environment, this is infeasible in practice. For the actual implementation, we had to rely on a parametric action-value function $\hat{q}_w = \hat{q}(\cdot, w)$ depending on a *weight vector* w of dimension $d \ll |\mathcal{S}|$. Therefore, a change in the weight vector potentially effects the estimation of the value of many different action-state tuples.

Fortunately, all of the learning methods presented in the last section can easily be modified to update the weights of a parametric action-value function. Generally, the weight update can be written as

$$w_{t+1} \leftarrow w_t + \alpha\big[U_t - \hat{q}(S_t, A_t, w_t)\big]\nabla\hat{q}(S_t, A_t, w_t), \tag{1.5}$$

where $\nabla\hat{q}$ denotes the gradient of $\hat{q}$ with respect to w and U_t denotes the *update target* that s's estimated value is shifted toward. In the Sarsa algorithm, the update target U_t is defined as $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t)$. Using this definition of the update target 1.5 becomes

$$w_{t+1} \leftarrow w_t + \alpha\big[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)\big]\nabla\hat{q}(S_t, A_t, w_t),$$

which nearly resembles the Sarsa update in (1.2). Moreover, we modified the n-step Sarsa update as well as the Sarsa(λ) methods accordingly.

1.2.1 Linear function approximation

More concretely, we began with a simple linear function approximation, that is we write

$$\hat{q}(s, a, w) = w(a)^T \cdot x(s) = \sum_{i=1}^{d} x(s)_i w(a)_i,$$

where

- $x(s)$ denotes a d-dimensional feature vector extracted from the game state s (see section 2.2),
- $w(a)$ is a weight vector for every possible action a.

In this section, we have to ensure that (1.5) only updates the weights $w(A_t)$ for the specific action taken.

The advantage of using a linear function approximator is its simple form as well as the excellent convergence properties proven in **?**. This linear function approximation was sufficient for many different tasks. The disadvantage is the need for *good* features $x(s)$. In the end, we sticked with this linear model and tried to increase the quality of the model by crafting high-quality features by hand as well as with the help of a genetic algorithm for feature extraction.

Chapter 2

Training process

2.1 Exploitation of spatial symmetries

by Matthias Hericks

In an early stage of the project, we noticed that the environment exhibits multiple spatial symmetries and though about how this can be exploited, when training an agent. Loosely speaking, if an agent knows how to act in a particular state s, this information can be used by the agent to act in the seven different states, which arise as symmetric transformations - flips / rotations / transposition - of the state s, given that its actions are transformed accordingly.

A search in relevant literature revealed that **?** identifies these symmetries as symmetries with *adherence to an equivalence relation*. Moreover, **?** describes two possibilities to exploit these symmetries in the training process of an agent.

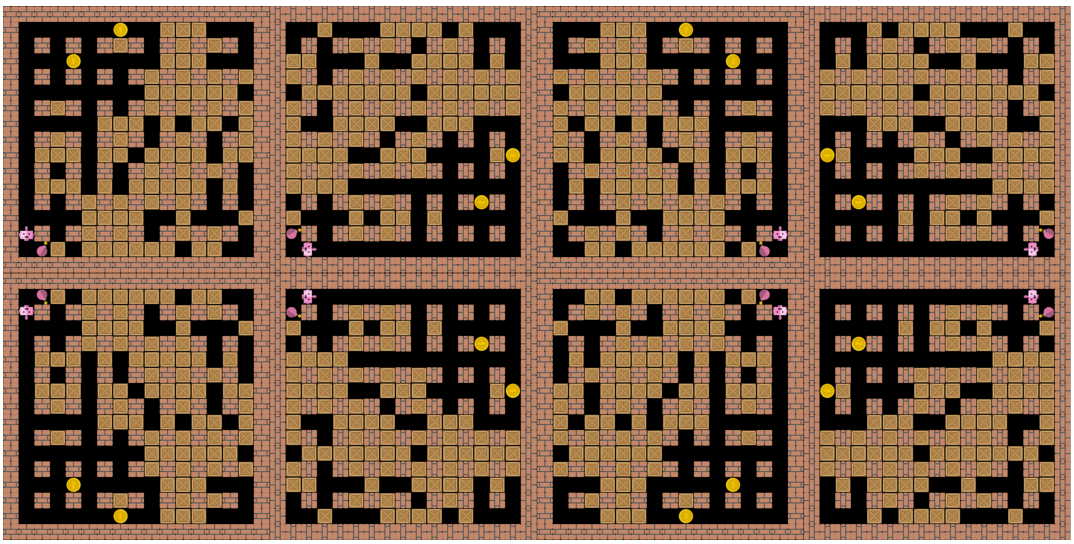


Figure 2.1: Visualisation of 8 states in the same equivalence class w.r.t. $\sim_{\text{sym}}$.

The key idea is to define an equivalence relation $\sim_{\text{sym}}$ on the set of all states, such that $s \sim_{\text{sym}} s'$ if and only if there exists a sequence of symmetric transformations, which maps s to s'. For every equivalence class under this relation, we fix one particular state in this

class as the *candidate* state. Therefore, for each state s there exists a candidate state s_{can}, such that symmetric transformations can be used to transform s to s_{can}. Instead of training on all possible states, the agent now merely trains on the candidate states. After each state transition the new state is transformed to the corresponding candidate step in a preprocessing step before presenting it to the agents learning method. Moreover, a special map is computed, which maps the action of the agent in the candidate state to the associated action in the original state of the environment in a postprocessing step. By doing so, we effectively divide the number of states which the agent encounters by 8, as seen in the graphic on the previous page.

We conducted a simulation study, comparing the convergence of a simple coin-collecting agent with varying exploitation of symmetries and obtained similar result as the ones by **?** (compare **??**)

2.2 Feature design

by Mika Rother

2.2.1 Simple features

First of all, a linear model requires good feature design. Since many features cannot be represented by such a model, the return values of a feature had to be either 1 or 0. Accordingly, it is not possible to create a *numpy array* that indicates for all bombs on the field how long a bomb needs to explode. In such a case, one would logically need an array for each time step of a bomb in a linear model that indicates whether a bomb will explode after as many time steps.

Accordingly, the first step was to design simple features that could be used to analyse the agent's behaviour and achieve initial training results. At the same time, it was also important that the features did not define a desired behaviour too precisely, so that the agent still had to learn to behave well itself.

So the work started with designing features that would teach the agent on a field without crates and only with coins to collect these coins as efficiently as possible.

Coins per quarter

The first feature that was developed was very simple: coins per quarter. The idea of this feature was based on giving the agent four features, where each of the features was for one of the possible moves, (where `'WAIT'` and `'BOMB'` were taken out as possible actions for this part, because it wouldn't make sense to place bombs or wait on a board without crates). The playing field was accordingly divided into top left, top right, bottom left and bottom right. Then the sum of the coins in each of these quarters was added up and given to the agent, so that he was always informed in which part of the field the most coins could be collected.

Training with these features alone led to some problems, of course. On the one hand, it could happen that a change of position led to a situation where suddenly the most coins were in a different quarter than in the step before. This often resulted in a back and forth movement, which is of course very inefficient. On the other hand, the agent often passed by nearby coins on the way to the quarter with the most coins. To fix this problem, we gave the agent more features.

Relative environment of the agent

Since the agent had not been aware of its immediate surroundings so far, *relative maps* were created to tell it which objects are in its immediate neighbourhood. Since only *walls* and *coins* were on the field at this point, maps were first created for these two objects. These then had the following format:

```
wall_map = np.zeros((31, 31))
coin_map = np.zeros((31, 31))
```

So both were maps of size 31×31. The idea of these maps was that the agent is always in the centre (`pos = [15,15]`) and sees the environment from his point of view. This had to include a total of 31 fields on both axes so that the agent also had a complete map available in the corners of the field.

It is easy to calculate that 31×31 entries for two maps lead to an extremely large number of features, since each field has either the value 0 or 1, depending on whether the object to be examined is located on this field or not. Therefore, and because primarily the immediate environment of the agent is interesting, another variable was introduced, the `NUM_LOOK_AROUND`. If, for example, the value 3 was chosen for this variable, the agent only got informations about his environment with radius 3, i.e. three fields in each direction from the agent and thus a 7×7 array for the relative maps. With the help of these features, considerable progress could already be made, even if the road to perfection was still a long one.

Escape death features

So, since the agent was able to collect coins in an acceptable time, the next step on the list was to move on to crates. Now the agent first had to learn to plant bombs to destroy crates without blowing himself up, and then collect the coins that appeared in the explosion. This task proved to be much more difficult, as the agent still had to earn as many coins as possible without blowing himself up.

Get all safe tiles

The first step was to find all the fields that could turn out to be unsafe. For this the function `get_unsafe_tiles()` was written, with the help of which one could check which fields in the periphery of a bomb were dangerous, i.e. on which fields the agent would not be allowed to stay without diying if the bomb exploded. This function was then

passed the `field` and the `bombs` on each turn, which could be imported from the current `game_state`.

Furthermore the function `get_reachable_tiles()` was created. This function was passed the current position of the agent (`pos`), the number of steps the agent can still walk until the bomb explodes (`num_steps`) and again the current playing field (`field`). Using this data, the function could then determine which fields could still be entered in a given number of steps.

These two functions were now used to find all fields that can still be entered in a given number of time steps without dying. This resulted in the following function:

```
def get_reachable_safe_tiles(pos, field, bombs, look_ahead=True):
    if len(bombs) == 0:
        raise ValueError("No bombs placed.")

    timer = bombs[0][1] if look_ahead else bombs[0][1] + 1
    reachable_tiles = set(get_reachable_tiles(pos, timer, field))
    unsafe_tiles = set(get_unsafe_tiles(field, bombs))

    return [pos for pos in reachable_tiles if pos not in unsafe_tiles]
```

With the help of this function it was now possible to teach the agent not to enter fields that led to a certain death. Because the function `get_reachable_safe_tiles` returns all positions, which an agent can still enter without dying.

Get safe death features

However, in order to be able to represent the whole thing as a feature in a linear model, of course only binary values (0 and 1) may be returned, so that training can be done in the correct way. Accordingly, a new function was created, which creates a numpy array with five entries (for the actions: `'UP'`, `'DOWN'`, `'LEFT'`, `'RIGHT'` and `'WAIT'` and depending on which action is feasible without dying, has a 0 or 1 in the respective entry.

For example, if the agent can only go up to avoid dying, the new function would return the following:

```
  ret = ([0,1,1,1,1])    # (['UP','DOWN','LEFT','RIGHT','WAIT'])
```

The function that creates this output is called `get_safe_death_features`. Thus, in addition to the possibility of collecting coins, a list of features was now given to help the agent understand which fields it would be better to stay away from in order to avoid death.

Up to this point, the agent was simply trained to avoid fields that could end in his death. However, he was not explicitly told not to commit suicide. For a behavior that

should be avoided in any case that the agent blows himself up by his own bombs. For example, it should be made clear to him at the beginning of the game that he must not die by planting his own bombs. This idea served mainly to ensure that the first action is not `'BOMB'`, since that usually ends in certain death. Therefore, another feature `is_bomb_suicide()` was developed, which returns whether a bomb planted by the agent led to his death.

Relative maps for escape death features

To round off the whole thing, relative maps for crates were generated, which were used to pass the crates in the immediate vicinity to the agent for training. The aim here was, of course, that the agent finds the way to neighboring crates as quickly as possible and does not have to search for so long.

As mentioned at the beginning of this chapter, in order to have all the time steps of all bombs and where they do damage, it would be necessary to create a relative map for each time step of a bomb. Since it would then be necessary to create four different maps, corresponding features were written, but rarely used for training, since it would then have been much more difficult to find good weights. And one of the goals of reinforcement learning is to train the agent with well-chosen rewards and not with a large selection of features.

2.2.2 Shortest paths

Now the point had come where our agent was performing his tasks to some extent, but not yet very efficiently. Therefore it was necessary to give him not only his environment, but actually the way he has to go, if he wants to find the next coin, for example. Since it is often important to know not only the way to the next coin, but also to the next crate or opponent, the function that calculates this shortest path had to be valid for several objects.

The first approch that was taken was a simple *breadth-first-search* algorithm that treats the game board like a maze.

BFS in a maze

In order to be able to represent the playing field as a labyrinth, a distinction had to be made in the first step between accessible fields and walls. Logically, fields with a wall or a crate were displayed as walls, the rest were displyed as free fields. This was then stored in an array (`free_tiles`) that has the same shape as the game board.

Based on this array, it was then checked if the agent has the possibility to reach its target. The parameters `free_tiles`, `start` and `target` were passed to the function `shortest_path()`. Here, the positions `start` and `target` represent the position of the agent and the position of its target. A maze was then created, looking in each iteration to

see which points could be reached with how many steps. The search algorithm that was used for this process was *breadth-first-search* (BFS). If the target position was reached, the search ended and the path was reconstructed. If no path could be found, the position of the agent was returned.

Although the shortest path could be calculated using this function, there was a major problem: the maze was recalculated for each step, which took about two seconds. That doesn't sound like much, but if one wants to train several workouts with at least 100 laps, then that was definitely too much.

A* search

An algorithm for finding the shortest path is the A* search algorithm. This is a generalization and extension of Dijkstra's algorithm, although Dijkstra's algorithm can also be reduced to the A* search algorithm. Now a new class has to be introduced to perform A* search:

```
class Node:

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

    def __lt__(self, other):
        return self.g < other.g
```

A node is thus initialized with a parent, a position and values f, g and h. The A* search algorithm always examines the nodes first that are the most probable to lead to the goal. For this, the values for f, g and h are required. If x is a node, the following relation holds:

$$f(x) = g(x) + h(x) \tag{2.1}$$

In this context, $f(x)$ is the estimated distance from the starting node to node x, $g(x)$ denotes the previous cost from the starting node, and $h(x)$ is a heuristic that estimates how far it is from the current node to node x.

Let u and v be two nodes and let v be the target node. Then the heuristic that was used in our A* search algorithm is given by

$$h(u) = 2 \cdot (u[x] - v[x]) + 2 \cdot (u[y] - v[y])$$

where x and y describe the coordinates of the game board. The choice of the heuristic is only a small detail, which can be improved, but since later only the nearest coins are considered, this was not necessary here. Nevertheless, an alternative evaluation function is given here, which Xiang Liu et al. tested in [**??**] for different mazes:

$$f(x) = g(x) + h(x) + h(y)$$

where y is the parent node of x. The is equal to the definition of the evaluation funtion defined in Eq. (2.1).

The A* search algorithm then creates two lists with the open nodes and the closed nodes. Then, starting from the starting node, the neighbors are traversed and whenever a node is found that has not yet been added to the closed nodes, a check is made to see if it is already among the open nodes. If not, it will be added and its g value will be increased by one compared to the node from which it was reached. If the node is already in the list of open nodes, it is checked if the new g value is smaller than the previous one. If it is, the entry is renewed. This also ensures that the parent node is always initialized with the predecessor node from which it is quickest to get to the current node.

If a shortest path to a position is found using this algorithm, this position is returned. If not, the position of the agent is returned instead. It has been shown that this algorithm is much faster than the maze-based BFS algorithm. However, the structure of the nodes has not yet been fully exploited, so the f,g, and h values have not yet been well accounted for.

A* search with priority queue

To get the last bit of performance out of the algorithm, it was necessary to store the open nodes list as a priority queue, where the priority in this case is determined by the f value of a node. The Python module `heapq` was then used for this, which provides basics for efficient data structures. This finally allowed a satisfactory runtime to be achieved, especially since later only the objects nearest to the agent were considered. And accordingly not all objects had to be traversed.

Find shortest path to specific objects

Once an efficient algorithm was available that could calculate the shortest path, the next step was to calculate only the shortest path to the objects of a class that were really close. Given the list of positions of all crates, one wanted to include only the relatively close crates in the calculation, since for example the amount of crates with a `CRATE_DENSITY` of 0.7 would be very large and a calculation of the shortest paths to all would cost too much performance.

Therefore, a function was written that is passed the list of objects, let's take coins as an example, and then finds the shortest Euclidean distance between the agent and the nearest coin using the following nesting numpy function (`best_dist`):

```
best_dist = min(np.sum(np.abs(np.subtract(coins, pos)), axis=1))
```

But since the coin with the shortest Euclidean distance to the agent is not necessarily the one with the shortest path, one could still extend the `best_dist` with an `offset` so that the shortest path from the agent to all coins in its neighborhood with radius R:

$$R = \texttt{best\_dist} + \texttt{offset}$$

was calculated.

The shortest path functions were then used to find the shortest paths to the next coin and to the next crate. When the shortest path function was tested with the agent collecting only coins, the results were much better than with the previous features. The idea behind using this feature for the crates as well was based on the observation that when training the agent with crates and bombs, he often did not move to other crates to destroy them. By training the agent to take the shortest path to crates, much better results could be achieved.

2.2.3 All features combined

Last small feature

Since the agent had been taught during the training with all the features described so far that he had to plant bombs to destroy crates, i.e. received a positive reward for it, unfortunately the problem occurred that whenever he could plant a bomb, he also placed one, even if no crates were destroyed by it. Therefore, another feature was needed that simply indicates whether the agent is standing next to a crate or not, with the idea that this would teach the agent to place bombs more often if it destroyed crates.

Play and train with all features

Based on all the features described so far, the agent could be trained to be capable of destroying crates and collecting the coins that appeared without dying. However, this did not always succeed and even when it did, sometimes more sometimes less well (more on this below).

2.2.4 Genetic algorithm for feature extraction *by Matthias Hericks*

At some point it was evident, that we spend more and more time on the construction of increasingly complex features. Since this is a machine learning project, we looked for ways to overcome this problem and came up with two possible directions:

1. Choosing a different (nonlinear) approximation function, like deep neural networks, decision trees or regression ensembles with gradient boosting.

2. Finding a way to learn good features (automated feature extraction).

We knew from extensive communication with other groups, which opted for option 1, that a change in the approximation function would not be straight-forward. Many more complex function approximators rely on off-policy batch gradient-descent updates and on-policy n-step Sarsa and Sarsa(λ) learning algorithms are not applicable at scale. Our initial idea was to implement PCA with training data from the rule based agent, since it is reasonable that the same features resulting in a good performance of the rule based agents might help our agent. A detailed discussion with the group of Fabian Kneissl, revealed that PCA was not applicable here. To explore some new grounds, we read a lot about genetic algorithms for feature selection and extraction with **?**, **?** and **?** being our primary sources.

The key idea, was to rewrite one of our ordinary linear agents, such that its training process depends on a a list of binary expression trees, which each represents a boolean expression. We will call such an agent a *genetic agent*. By default we provide the genetic agent with all of the complex features mentioned above as well as a high number of simple features, e.g. maps representing the agents relative environment up to four tiles in each direction. We will denote this high-dimensional feature vector of a state s by $x(s)$ and call $x(\cdot)$ the *raw features*. All of the boolean expressions represented by the trees, take fixed components of the raw feature vector as input and therefore represent composite features depending on one or multiple raw features. Instead of using all of the raw features for the linear function approximation of the action-value function q, the genetic agent now learns with its composite functions. Therefore, the effective number is of features used in the linear regression is no longer the (possibly large) dimension of $x(\cdot)$, but merely the length of the agents list of binary expression trees.

The problem with simple features.

To give a concrete example, lets assume that the raw feature vector $x(s)$ contains one-hot encoding of the agents surroundings. In a linear model this is only of limited benefit for the agent, since many important situations can not be described well by such a model. If a bomb is one tile above and two tiles to the right from the agent, it is quite possible, that the linear agent will learn to avoid the action `UP`, since in some situations such a bomb might have killed it. However, in a linear model the agent will shy away from the action `UP`, even if there is a wall one block above and one block to the right of him, which would block the bomb. If he still chooses the action `UP`, it is quite possible that the wall to the top right of him adds positive value to the estimation of the `UP` action. However, this is not desirable if the tile above the agent is blocked.

Composite features might fix this problem, by introducing features like `is tile above free AND (is no bomb in the row above OR will a wall block such a bomb)`, where most of the operands are itself composite expressions. Therefore, the aim is to find automatically construct binary expression trees, which represent good features.

For this complex task, we introduced a genetic algorithm following the following process for feature extraction.

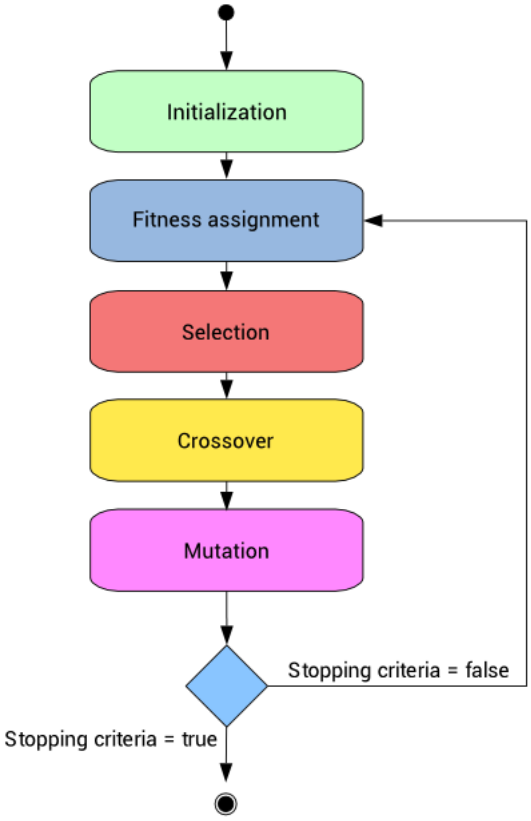


Figure 2.2: Modified feature extraction process from **?**.

An external script, initializes multiple lists of binary expression trees at random. We will call this set of lists the *population* and one list of binary expression trees an *individual*. Then for each of these individuals the fitness of the associated genetic agent is evaluated by training a linear agent with the corresponding composite features and measuring some kind of performance, e.g. average reward over the last 50 training episodes. After this is done for all the individuals, the best performing agents were selected as *the offspring* and cloned to create a new population. Moreover, on some of the clones mutation and crossover operations were performed. Mutation operations mutate some of the binary trees of an individual at random. Crossover operations take two individuals and exchange whole trees or subtrees. This resulting group of individuals is called the *next generation*. The process is now iterated with the next generation as starting population. Throughout this process, the average performance of the individuals in the population are tracked, which hopefully increases.

For the implementation, we used the `DEAP` *"Distributed evolutionary algorithms in*

python" library, which gave us a lot of flexibility and even implemented some standard mutation and crossover operations on binary expressions trees. Using this method, we were able to construct good features in toy examples. One example being the selection of valid game-related features in a raw feature set with many random features (a feature returning 0 or 1 at random). Moreover, the genetic algorithm was able to find solutions to the simple feature problem mentioned above.

Unfortunately, there are still multiple flaws with this feature extraction algorithm.

- For every new generation, the fitness of each new individual in the population has to be evaluated by training an agent. Since typically many generations are necessary to construct a good features, e.g. a good population, this easily results in long runtimes.
- The genetic algorithm itself introduces a variety of hyperparamters, the primary ones being the population size, the mutation probability, the crossover propbability and the number of generations. Moreover, the concrete choice of mutation and crossover algorithms determines the failure and success of the algorithm.

We tried to use as many heuristics and guidelines mentioned in the sources mentioned above, but in the end the deadline forced us to use our handcrafted features together with a few compound features motivated by the genetic algorithm. Still, the results in toy examples look promising and we are eager to learn more about genetic algorithms for feature extraction in the future.

2.3 Rewards and fine tuning

by Mika Rother

The first part of this section is about the different rewards we gave to the agent for different events. This part is further divided into a rewards available through regular events, i.e. for events that have already been defined from the beginning in `event.py`, and rewards based on custom events.

2.3.1 Rewards based on regular events

At the beginning, when the agent was only supposed to collect coins, we only needed a reward that would reward him for collecting a coin. Afterwards, when we wanted to teach the agent to plant bombs to destroy crates, the awarding of rewards became much more complicated. In order to train successfully, we had to teach the agent several things to instill the correct behavior. The following points had to be checked off:

- The agent must be able to plant bombs to destroy crates
- The agent in any case is not allowed to blow himself up
- The agent must continue to collect the coins
- The agent should not perform too many invalid actions

- The agent should try to play fast and play efficient

To accomplish all these points, a certain amount of subtlety was required to set the rewards correctly. For example, if a parameter was changed too much, an undesirable behavior often emerged (more on this below). Here is an overview of the rewards we have used up to this point:

Rewards	Range of values
`CONSTANT_REWARD`	$[-1, -0.1]$
`e.COIN_COLLECTED`	$[5, 15]$
`e.KILLED_SELF`	$[-50, -15]$
`e.CRATE_DESTROYED`	$[2, 5]$
`e.INVALID_ACTION`	$[-0.5, -0.1]$
`e.BOMB_DROPPED`	$[-1, 2]$

Two things in particular are noticeable here:

1. The range of `e.KILLED_SELF` is quite large relative to the other rewards. This is because we have done a lot of testing on what happens when the agent is heavily punished for a suicide, and what happens when he is not punished so much for it. The latter might motivate him to plant more bombs, which is in principle a desired behavior.

2. We actually had training sessions where we gave the agent a negative reward for planting bombs, as well as rounds where we rewarded him for doing so. This is because we wanted to teach the agent to plant bombs only if they lead to the destruction of crates. In this case we increased the reward for `e.CRATE_DESTROYED`.

We also used a negative constant reward, so the agent doesn't wait the whole game.

2.3.2 Rewards based on custom events

The problem caused by the second point was the following: As soon as the agent was given a reward for `e.BOMB_DROPPED` < 0, it developed the behavior of only waiting, (or, if it was also given a negative reward for waiting, only making `LEFT-RIGHT` or `UP-DOWN` movements). So it was not an option to give him a negative reward directly for bombing.

However, since we wanted to continue to make him more likely to plant bombs that also destroy crates, we developed our own events: Now let's explain the sense of them:

- `CRATE_DESTROYING_BOMB_DROPPED`. As already mentioned, this is the event that should develop the behavior of destroying crates with dropping bombs.

- `BOMB_DROPPED_NO_CRATE_DESTROYED`. This reward, on the other hand, should have exactly the opposite effect. For this event the agent should get a negative reward, because here the planting of a bomb does not cause the destruction of crates.

For these two custom events we could now also give rewards without rewarding the agent just for planting bombs. However, we still gave a reward for the event `e.CRATE_DESTROYED` because we noticed that we can't set `CRATE_DESTROYING_BOMB_DROPPED` too high without making the agent lay too many bombs again, thus simplifying fine tuning.

Finally, here is the final overview of the rewards we used to train the agent:

Rewards	Range of values
`CONSTANT_REWARD`	$[-1, -0.1]$
`e.COIN_COLLECTED`	$[5, 15]$
`e.KILLED_SELF`	$[-35, -15]$
`e.CRATE_DESTROYED`	$[2, 4]$
`e.INVALID_ACTION`	$[-0.5, -0.1]$
`CRATE_DESTROYING_BOMB_DROPPED`	$[2, 7]$
`BOMB_DROPPED_NO_CRATE_DESTROYED`	$[-2, -0.5]$

With this set of events, training was further simplified. Unfortunately, it turned out that some behavior patterns were difficult to avoid and occurred again and again when testing the agent. For example, if the agent was standing right next to a crate and there were coins on the field at the same moment, the agent got into the habit of waiting if it had to increase the Euclidean distance to the next coin first, i.e. move away from the coin first to reach it.

2.3.3 Fine tuning of other parameters

The values that could be assigned to the various rewards were not the only changeable parameters that could be fine-tuned. We tested different values for various types of parameters from time to time in order to achieve the best possible training results. In the following, these various parameters are described and their variation is briefly explained. (The exact results for the important points can be read in Experimental results).

Variying hyper parameters

The hyper parameters of the training were the *learning rate* (α) and the *discount factor*. Based on the book of Sutton [**??**] and of our simulation studies we learned that the following realtion leads to good results:

$$\alpha \times \texttt{NUM\_FEATURES} \approx \text{const.}$$

So we tried to choose α so that this realtion is approximately kept. For the discount factor we achieved good results, when choosing

$$\texttt{DISCOUNT\_FACTOR} \sim 0.85 \pm 0.1$$

Variying policy type parameters

During the project we used two different types of policies as explained in **??**. In most cases the randomness was reduced with increasing number of training iterations. This yielded good results for both ε-greedy and the softmax policy.

Variying parameters of update algorithms

In total, we used three different update algorithms:

- Sarsa,
- n-step-Sarsa and
- Sarsa(λ),

where Sarsa is the same as n-step-Sarsa with $n = 1$. For n-step-Sarsa, we obtained good results by choosing n between 3 and 6. For Sarsa(λ), we could choose the parameter λ between 0.7 and 0.95 to get good results.

Chapter 3

Experimental results

3.1 Introduction

by Alessandro Motta

In our challenging journey of setting up our agent for the ring against other agents we always had to try different combinations of the functions we created. These are the reinforcement learning method, normalizing the state, the different features and the game settings, such how many coins, crates and steps there are in each game. While playing around with all the different functions we created we wanted a way to measure how good our agent was learning. To safe some samples that indicated how our agent was learning we created a folder named "Histories" that contained a `.pt` file with the different stats of the game. Such as the number of coins collected, the number of invalid moves etc. This helped us to keep track of what settings lead us to which results. The next step was to write code to help us monitor these results in a clear and structured way. This file is called `monitoring.py`. To help explain our thought process we will divide the section of experimental result in 3 different parts in which we will also explain what interesting observations we made and what changes we:

(1) Coin collector (collect coins in an open field)

(2) Crate agent (set bombs to collect coins)

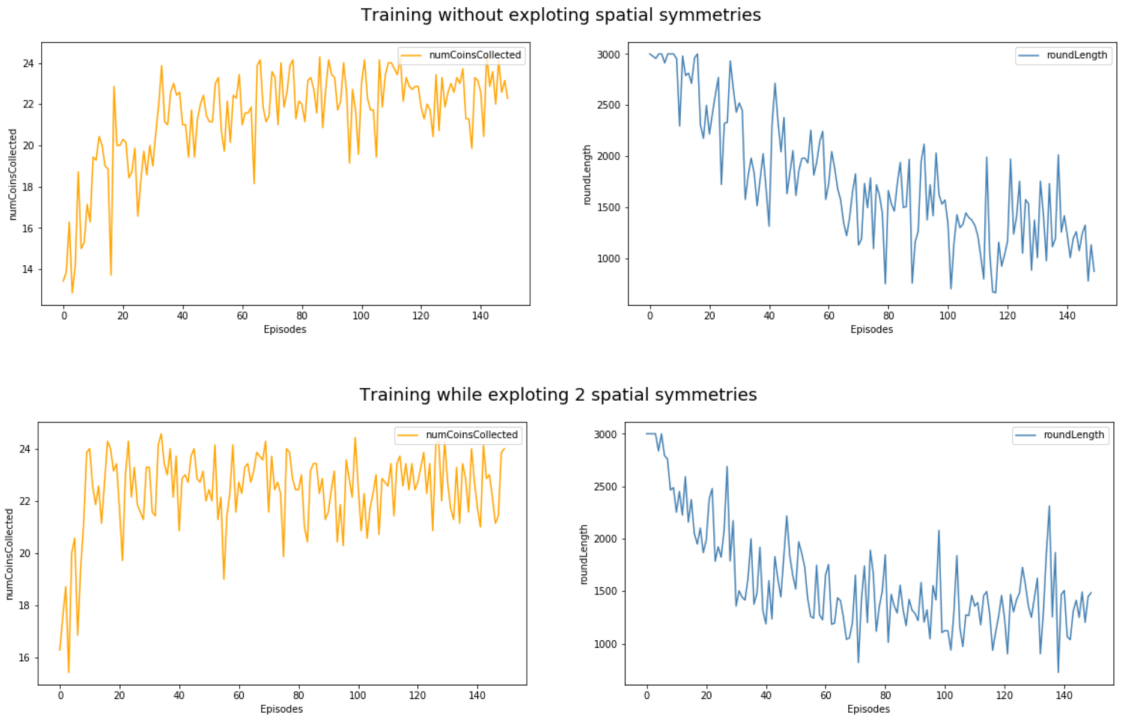
(3) Simulation study: n-step Sarsa vs trace-decay

3.2 Coin collecting agent

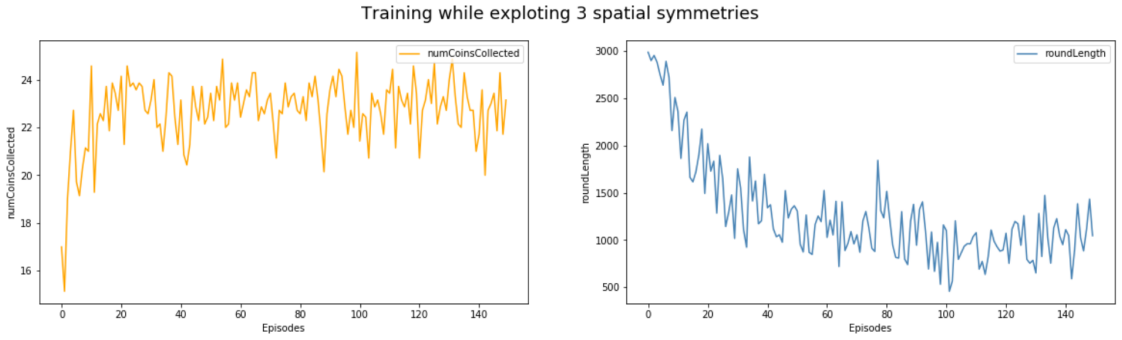
by Alessandro Motta

3.2.1 Exploiting symmetries

The first mission of our agent was to collect coins in an open field without any crates. For this mission we started exploiting the symmetries and had as features: the coin map with look around, the wall map with look around and the coins in quartal; while using an epsilon-greedy policy. We set the end of a game at 3000 steps and multiplied x3 the number of coins that spawned in a game. For the last setting we wrote a function that multiplies the coins. After implementing this function we observed that it was possible for the game to spawned more than a coin at the same position, so we decided to change that. In our first monitoring we wanned to be sure that training while exploiting two spatial symmetries led to a quicker learning process then not exploiting any.



As we could observe a better learnig curve we decided to exploit the third spetial symmetry. The diagonal spetial symmetry.



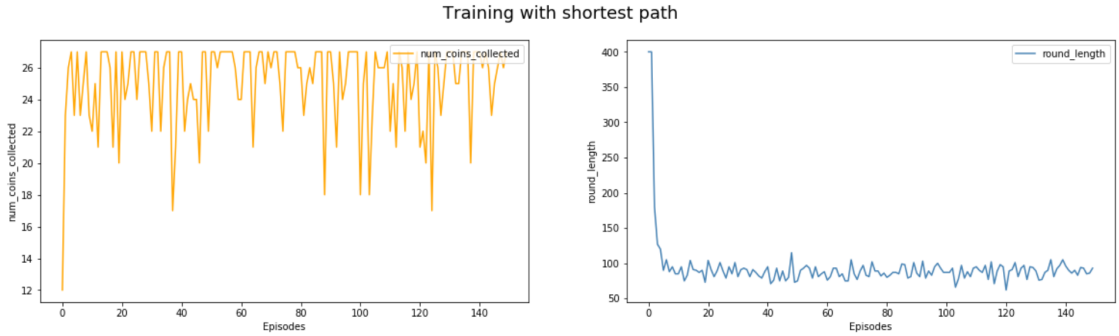
3.2.2 The quartal feature problem

As of this point we started noticed a big problem: Mostly when in playing mode our agent started to get stuck and repeat the same two moves after each step, going up and down or left and right for the rest of the game. This is because in playing mode we set a low epsilon so that he would do less random moves and just select the "good ones" with the idea that while playing he did not need to explore. We discovered that the agent got stuck because when there were only a few coins left he could not decide to which quartal

he should go. The only way he could get out of repeating these moves over and over was after by performing a random action and get out of this state. Since in training mode there was a higher chance of doing a random move (bigger epsilon) he did not get stuck as often as in training mode. We tried to fix this bug by rewriting the coins in quartal feature but the problem persisted. So we decided to use a higher epsilon also when in playing mode. This helped to get out of this bug-state quicker, but we still had one problem, we needed a lot of moves to collect all the coins. We came upwith two different solutions for this problem: 1)Implementing a softmax policy. A softmax will also help us in our next task, since when our agent sets a bomb he cannot allow himself to do at random all actions with the same probability. 2)Implementing a feature that describes where the next nearest coin is.

3.2.3 Shortest path feature

After implementing the shortes path feature we managed to fix the bug and got really good results as we can see in the next graph.



3.3 Crate agent

by Alessandro Motta

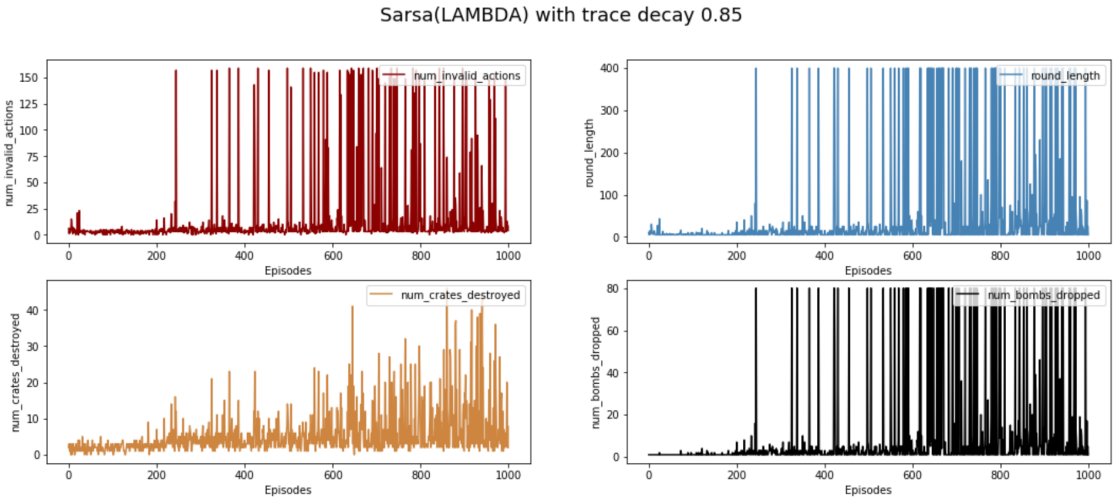
3.3.1 Setup

In this section the task of our agent was to destroy crates in order to collect the coins around the map. Therefore we needed to implement new features and had to change our monitoring in order to see how many bombs were dropped, how many crates were destroyed and other helpful things that we started implementing along the way. A big challenge was to keep track of all the trainings with all the different game, methods and feature settings. Since we tried to train with as much combinations as possible. To help us train as much combinations as possible we wrote a shell script (train.sh) so that with one execution of the code we could train multiple agents for certain number of episodes. Afterwards we wrote a `simulation_study.py` to make our agent train during the night with different combinations.

3.3.2 Agent behavior in relation to rewards

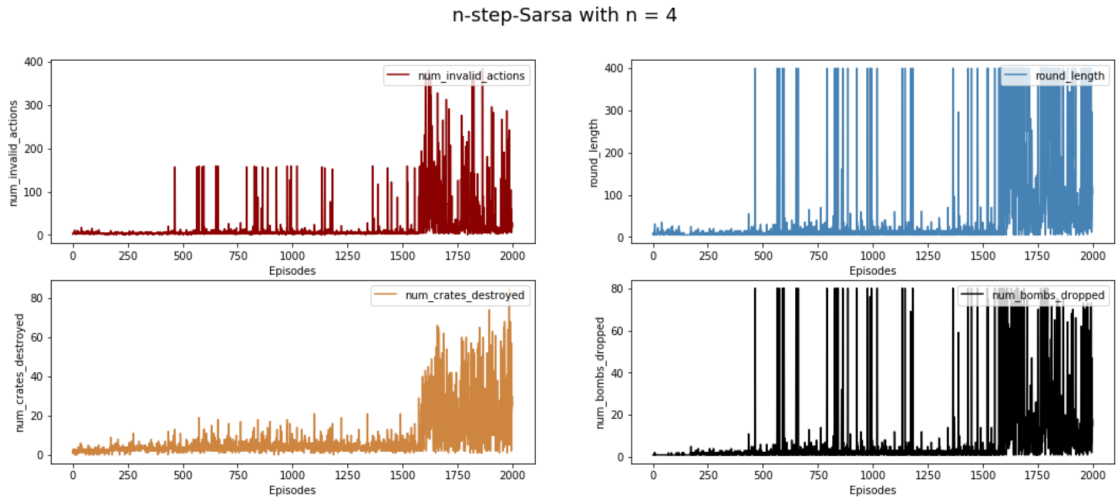
We tried to fine tune our reward system but did not really come to the results we were hoping for. During all our test we had some really inconsistent results. We observed that really small changes in the reward system could make really big differences in the behavior of the agent. The most challenging thing was trying to make the number of crates that were being destroyed during the game converge. As we can see in the following graph the behavior of the bot was really polarized. We understood while training that no matter what rewards we gave him he could only learn two different behaviors.

1. Dropping a bomb everytime he could was good. We gave him a small reward for dropping bombs, a big one for collecting coins, destroying crates and a negative reward for dying. This led that directly when spawning he started dropping bombs. We can see this in the graph when he drops 80 bombs per round (we trained for a round length of 400 steps and he can drop a bomb every 5 rounds, $400 \setminus 5 = 80$). Sadly sometimes the crate layout is made so that if you release a bomb at the beginning of the game you have no escape and you die in the fifth round.



2. Never dropping a bomb was good. We used the same rewards as before but no rewards for dropping bombs. Everytime he dropped bombs he did not manage to escape, so he died directly. Leading to thinking that dropping them is bad.

After this upsetting experience we created a custom event for giving the agent only a reward for setting a bomb at a position where he did not commit suicide. This also did not help directly since he would place the bombs at the right position but then die and get a good reward anyway.



3.4 Simulation study

by Alessandro Motta

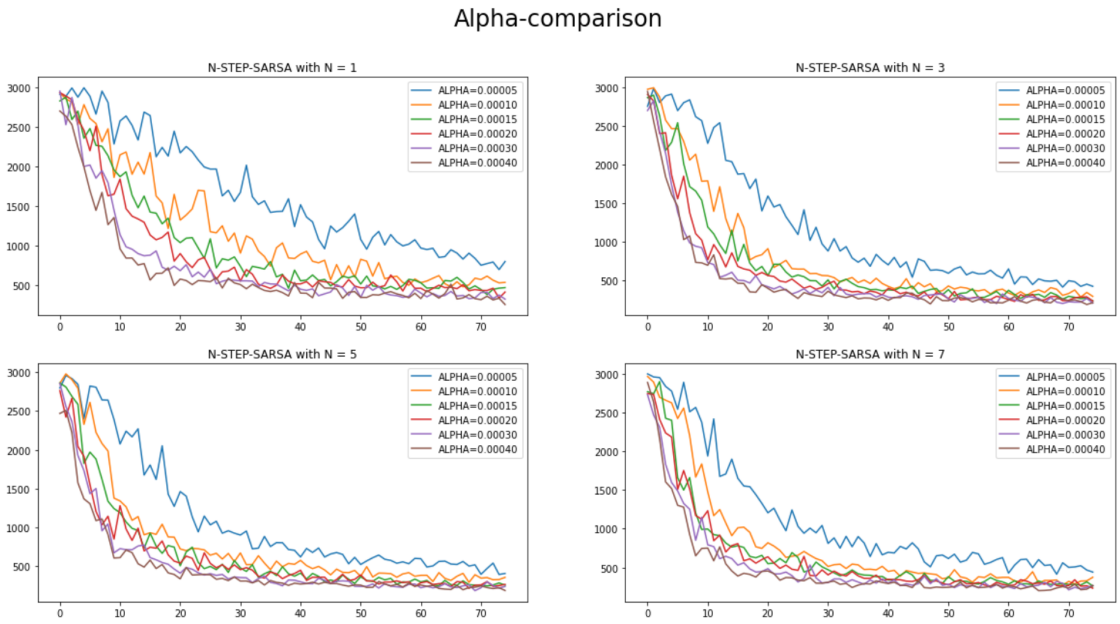
3.4.1 Goals of the simulation study

Thanks to our simulation study function we were able to make a comparison between n-step Sarsa and trace-decay for our linear agent. The parameters and the methods that we wanted to compare:

1. The n for the n-step-Sarsa method
2. The λ for the Sarsa(λ) method
3. The learning rate (α)
4. The two methods with the different settings

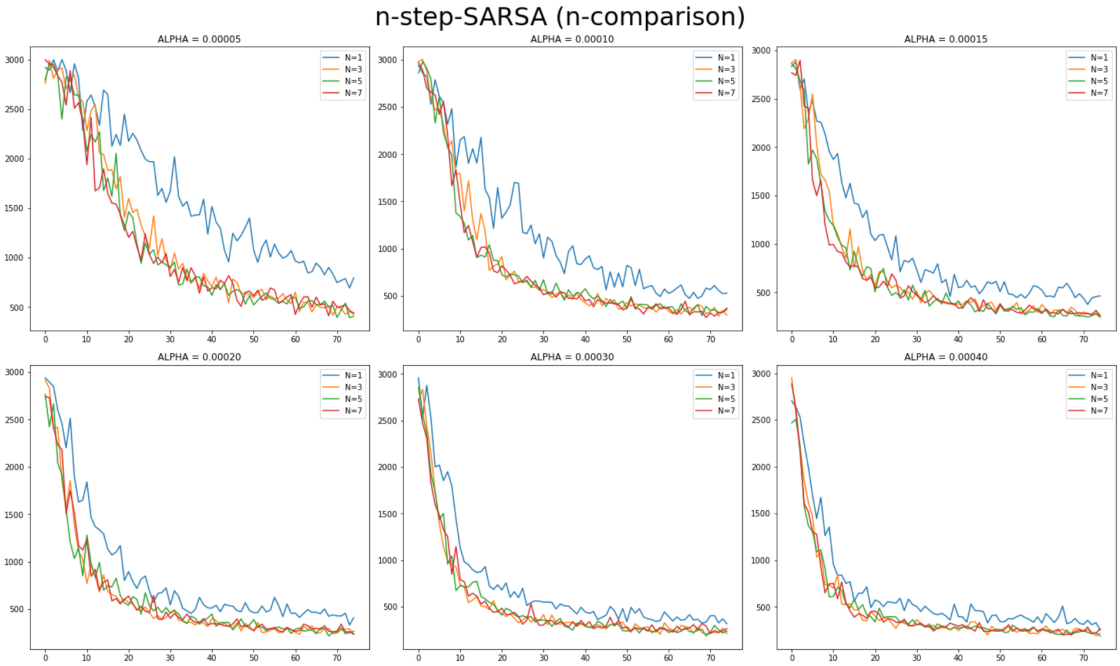
α-comparison & n comparison

All the graphs of the simulation study are in the file `simulation_study.py`. The thing that helped us out the most of the simulation study was the learning-rate (α) comparison. We had been using a learning rate that was too low. We could use a higher learning rate to get faster results.



We chose this graph since the difference of how the learnig-rate can affect the training is really easy to see. On the x axis we can observe the number of Episodes on the y axis the number of the steps the game lasted. We used the coin collector agent (without shortest path) to make this study since we were already sure of its behavior and knew that it can converge. This means that to know how good the agent is we want to see that the episodes last less. Due to the fact that after collecting all the coins the episode was over. In the light blue we can see the smallest α value. It is observable that a bigger α leads to better results faster (see the brown or purple line for example). From this graph we could also deduct that using a $n > 1$ for the n-step Sarsa method was better. We also made a n-comparison for n-step-Sarsa to be sure of this.

We did the same thing for the Sarsa(λ) method but noticed only a big difference between the different alpha values but not for the λ values.

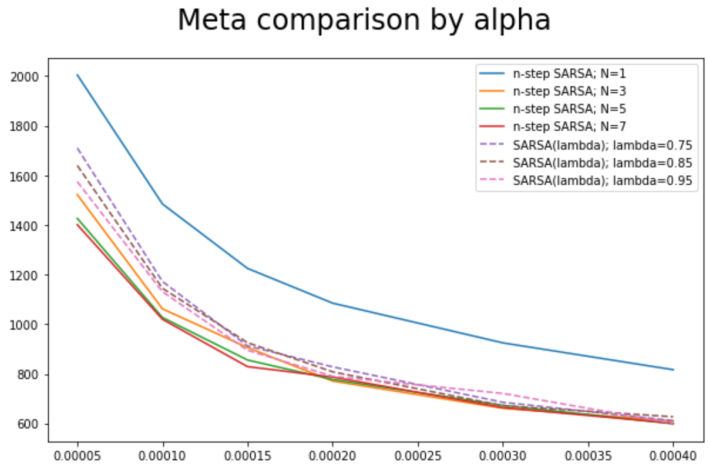


n-step Sarsa vs Sarsa(λ)

Last but not least we wanned to compare the two methods with each other. For this we took inspiration from a graph that we saw in the book [??] and plotted a similar graph. On the x-axis we can see the mean of the round length for the first 50 episodes and on the y-axis the different alphas. Note:

1. The graph depends also on the number of features that are used. For this plot we used 166 features.

2. If we would have used a bigger α we might have seen how the lines would have started to go up (not learning as fast as our current α's).


Meta comparison by alpha
n-step SARSA; N=1
n-step SARSA; N=3
n-step SARSA; N=5
n-step SARSA; N=7
SARSA(lambda); lambda=0.75
SARSA(lambda); lambda=0.85
SARSA(lambda); lambda=0.95
2000
1800
1600
1400
1200
1000
800
600
0.00005
0.00010
0.00015
0.00020
0.00025
0.00030
0.00035
0.00040