Exploring the fundamentals of Reinforcement Learning through Q-Learning methods

Levin Ceglie

Matura Paper

Supervised by: Urs Sieber, MSc

Submission date: January 29, 2021

Kantonsschule am Burggraben
Burggraben 21
9000 St.Gallen

KANTONSSCHULE AM BURGGRABEN

Abstract

This work discusses and answers two hypotheses. The first being the following: *One of the most basic solution methods of reinforcement learning, namely tabular Q-Learning, is adequate to solve the Cartpole environment, provided by OpenAI-Gym.* And the second: *Using deep reinforcement learning, an agent can learn to play the game of Snake on a human-level performance.* As stated in the hypotheses themselves, they lie within the field of reinforcement learning. Therefore, this work begins by exploring and explaining the theory underlying reinforcement learning. This work will conduct three different experiments to not only answer the stated hypotheses, but also to improve the understanding of the presented algorithm.

We start by understanding what the concept behind reinforcement learning is and set the foundations for this work with a mathematical framework called the Markov decision process (MDP). The MDP is the base for the different solution methods. The first one being tabular Q-learning. To improve our understanding of said solution method, we start by solving a simple gridworld environment, where we discover the immense importance of parameter optimization.

We apply the same tabular Q-learning algorithm to the Cartpole environment provided by OpenAI-Gym. OpenAI-Gym is a programming library, written in python, for developing and comparing reinforcement learning algorithms. With the parameter optimization method explained in the first experiment (i.e., the gridworld environment), we manage to find a set of parameters, with which the agent is able to solve the Cartpole environment in only 164 episodes. That result confirms the first hypothesis.

Before we tackle the second hypothesis, we look into artificial neural networks. We first discuss their role in reinforcement learning, followed by an overview of their structure. Then we dive deeply into the details of how an artificial neural network functions. We also learn the basics of optimizing an artificial neural network to assume a given function. With that knowledge, we incorporate artificial neural networks into our existing tabular Q-learning algorithm and thus end up with a deep Q-learning algorithm, which allows greater complexity.

Finally, we set up the Snake game as to fit in the framework of an MDP. Most interesting is the state we choose to return to the agent. We find that there are two options when thinking about the state representation given to the agent, a top-down view or a 2D vision, and within those two categories, there are again many different options. We continue by optimizing the agent's parameters as best as possible within a reasonable timeframe. With those optimized parameters, we train sixteen different agents using eight different state representations. Besides the significant performance differences between the state representations, we find that most agents were able to outperform a human test group, consisting of fifteen people, by a very significant margin. To be more specific our best agent averaged a score of 27 on a nine by nine grid, which is equivalent to filling 35% of the grid. Whereas, the human test group achieved an average score of 1.1 that is equivalent

to filling 3% of the grid. Furthermore, our results surpassed ones from literature, as their agent filled 8% of the grid on average. These results confirm our second hypothesis.

Contents

1 Introduction

This Matura Paper is concerned with two hypotheses:

- One of the most basic solution methods of reinforcement learning, namely tabular Q-Learning, is adequate to solve the Cartpole environment, provided by OpenAI-Gym [1].
- Using deep reinforcement learning, an agent can learn to play the game of Snake on a human-level performance.

The motivation behind this Matura Paper can be summed up with the following question. How much can a Matura student with hardly any amount of prior knowledge in the field of reinforcement learning achieve in said field within the scope of a Matura Paper? In order to explore this question and confirm or deny my hypotheses, I will first be concerned with the theory behind reinforcement learning. Following that, I will conduct thorough experiments using different Q-Learning methods.

It is important to mention that this work will only discuss the theory on a high level since it would be too much to cover it in detail within a Matura Paper scope. For a deeper dive into the theory, I can only recommend *Reinforcement Learning, An introduction*, by Sutton and Barto [10].

2 Theoretical Overview

Reinforcement learning is two things simultaneously. For one, it is a problem, but it also refers to the solution-methods of said problem. The so-called *Reinforcement Learning Problem* can be defined in one short sentence as the following: *Given an environment, how does one act to maximize a numerical reward signal* [10]. The solution-methods obviously cannot be defined in just one sentence. Section 3 and 4 will be concerned with those. However, for now, we shall focus on understanding what reinforcement learning is all about.

2.1 Trial and Error

The fundamental process underlying reinforcement learning is *trial and error* [10], which can often be observed in children playing with toys. There is a well known toy, which serves as a good example. The toy I am referring to is a big cube with holes of different shapes, where the goal is to fit given objects through those holes into the cube.

Figure 1: Example toy [7]

Following the process of trial and error, a child with no prior knowledge would either select an object and try it on every hole or select a hole and try every object on it. If we were now to give the child a reward of any kind for each object it fits into the cube, it would quickly learn which object fits through which hole and adapt its behavior, assuming it fancies the reward. This exact idea is what defines reinforcement learning at its core. The following is a more formal definition, of which the terminology will be explained soon.

> *Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward.* [17]

2.2 Markov decision process

What we discussed in the previous section shall now be formalized into a more mathematically usable form. In order to do that, we first have to understand certain terms, which are mostly self-explanatory. First of all the *agent*, he is the one who takes *actions* in an *environment*, which may or may not be affected by his actions. The *reward signal* is given to the agent for every action taken. It is a scalar measurement for how well the agent did. Now one might ask, based on what does the agent choose his actions? Well, last but not least, we have the agent's *state* in an environment, which can also be thought of as the agent's observation of the environment. [11] [10]

To give a concrete example, let us look at our situation from the previous section. In this case, the agent would be the child, and the environment would be the toy, including the big cube and objects. The state would be whatever the child observes through his sensory input. In this case, his visual input would be most important. One action might be to push a cube through a square hole into the larger cube, for which the child would receive a positive reward, whatever that might be. Contrary to that, if he had tried to fit a cube through a round hole, he would have either received a negative reward (i.e., a punishment) or a neutral one. At this point, his state might be him seeing a cube in his hand as well as two holes on a large cube in front of him, one shaped like a square the other like a circle.

The just discussed terminology is part of the *Markov decision process* [10] [16] or MDP for short, which provides a mathematical framework to our agent-environment model.

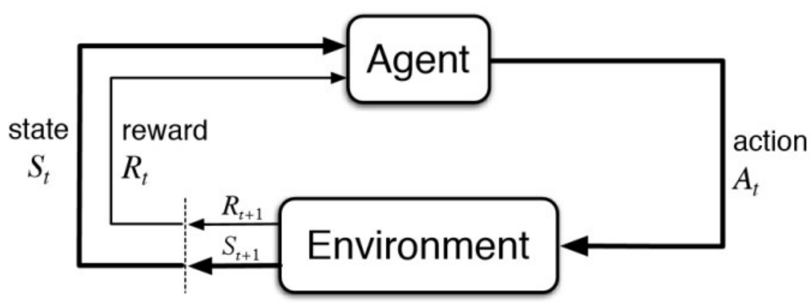

Figure 2: agent-environment model described as an MDP [10]

As visualized in Figure 2 an MDP describes the interactions between an agent and an environment in a sequential order. If we were to form a timeline it would look somewhat like the following: $S_0, A_0, S_1, R_1, A_1, S_2, R_2, A_2, ...$ where the subscript represents the time t. As you might have noticed there is no R_0, which is due to the fact that a reward always follow an action and since there is no A_{-1}, there is also no R_0. To further comprehend this notation, it is important to know that S, $A(s)$ and R are the sets of all possible states, actions in a given state s and rewards respectively. An element of those sets would be written as s, a and r respectively, thus $s \in S$, $a \in A(s)$ and $r \in R$. To omit confusion I want to quickly indicate the difference between S_t and s. S_t is a random variable of the state at time t and s is one of its possible events with the probability $Pr\{S_t = s\}$ of occurring. [10] [11] [9]

However, an MDP is more than just a way to describe a sequence of actions and its consequences. It can also serve as a complete model of the environment's dynamics. Thus it can also probabilistically predict future states and rewards given an initial state and action. Here we introduce the first important function. [10]

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1}$$

The function p answers the question: What is the probability of the agent ending up in s' (the successor state of s) and receiving a reward of r if he is currently in s and takes the action a? In other words p describes the *dynamics* of the MDP. The fact that p fully defines the dynamics of an MDP gives rise to the so-called *Markov property* [10]. If a state includes all the information necessary from past states (i.e., the history) that affects the future states, it is said to have the *Markov property*, which in an MDP every state must have. [9]

2.3 Policy and Value Functions

As mentioned in the definition for reinforcement learning in Section 2.1 the agent does not only care about its immediate reward. Instead, he tries to maximize the expected cumulative reward, which is also called the *return* and is denoted G_t. [10]

However, before we dive deeper, we have to take a step back and look at the difference between an *episodic task* and a *continuous task*. In an episodic task, there is at least one *terminal state*. If the agent reaches it at any time, the episode is terminated, the environment resets, and he starts over. The time step at which he reaches the terminal state is denoted T. Whereas in a continuous task, there are no terminal states. Thus, as given by the name, it is continuous. Hence the environment never resets [10]. From here on out, this work will be assuming an episodic task, as all the conducted experiments fall into that category.

Let us continue by defining the simplest form of the return G_t, which is just the sum of all following rewards. [10]

$$G_t \doteq R_{t+1} + R_{t+2} + ... + R_T \tag{2}$$

This form of G_t is rarely used in real applications. A more frequently used definition for G_t

is the following:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3}$$

This definition [10] has two main advantages over the former. First, we added the parameter $\gamma \in [0, 1]$ called *discount factor*, which enables us to chose how farsighted the return should be. If we set $\gamma = 1$, that means an immediate reward is worth as much as a reward n time steps into the future. The contrary happens when $\gamma = 0$. Then we only care about the immediate reward. The ability to adjust γ can be especially useful when we do not have a perfect model of our environment, which often leads to uncertainty when looking too far into the future [9]. The second advantage is worth mentioning but does not affect this work since it has to do with continuous tasks. The introduction of γ enabled the return to be a finite value (as long as $\gamma < 1$), even if the task is continuous, thus never-ending. In our previous definition of the return in (2), a continuous task would have an infinite undiscounted sum as its return (which is rather hard to work with) since there is no terminal state. [10]

Now as we defined the *return* we can get to *policies* and *value functions*. However, before we define them mathematically, we should understand them as a concept. A *policy*, denoted π, is what an agent uses to chose an action given a state. More accurately, π is a distribution over actions given a state [9]. As a function it would be written as $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$ and return the probability of a being chosen given s. On the other hand, a value function, as suggested by the name, outputs the *value* of a given state. The value of a state s is defined as its expected *return*. It can formally be defined as the following: [10]

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] \tag{4}$$

A value function is always reliant on a policy. In other words, we cannot know the value of a state if we do not know how we will behave. Thus π is found in the subscript. From this first definition, we can easily derive another essential and well-known equation as follows: [10]

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...)|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \qquad (5)
\end{aligned}$$

(5) known as the Bellman Equation gives rise to a recursive property of the value function [10] [9]. To better understand this property, it can be useful to look at a so-called backup diagram shown in Figure 3.

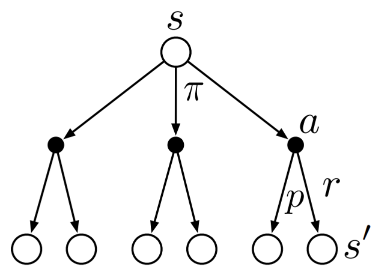

Figure 3: Backup diagram for v_π [10]

The circles and dots represent states and actions, respectively. Now we can easily see why the recursive relationship is true. The value of s ($v_\pi(s)$) is equal to the immediate reward r plus the value of the successor state s', which is discounted because it is one time step into the future $\gamma v_\pi(s')$. From this diagram it also makes sense, why there is a $\mathbb{E}[\cdot]$ in (5). The reason is that it is not certain what r and s' are going to be. They depend on the agent's policy π as well as the environment's dynamics p. Therefore we use their random variables S_{t+1} and R_{t+1} in combination with $\mathbb{E}[\cdot]$. [10]

2.4 Action-Value Function

The *action-value function* is not too different to the ordinary value function discussed in the last section, which takes the state as an input and returns the value of that state when following a particular policy. On the other hand, the action-value function takes a state and an action as an input and returns the value of that state when first taking the given action and following a specific policy thereafter. [10]

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \tag{6}$$

As you can see (6) is hardly different to v_π. Another important difference between these two functions is that q_π allows us to check which action will result in the biggest estimated return, whereas v_π tells us the value if we were to act according to π. This feature of q_π will be quite handy to us later on when it comes to behaving optimally. As one might expect, there is also a Bellman equation for q_π, which can easily be derived from (6) by removing the expectation. [10]

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_a \pi(a'|s') q_\pi(s',a')] \tag{7}$$

Once again, we can form a backup diagram [10] for a better understanding of the recursive property. This time around, it starts with a state-action pair, followed by a state and the different actions available in that state.

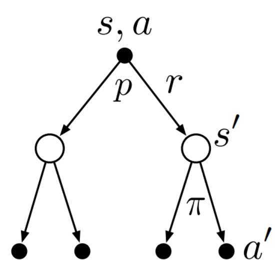

Figure 4: Backup diagram for q_π [10]

2.5 Bellman's Principle of Optimality

Bellman's principle of optimality states [10]: There is always at least one policy that is better or equal to all other policies, called the *optimal policy*. However, what does it mean for a policy to be better than another policy? Well, since our goal is to maximize return, we can compare the corresponding value functions. To demonstrate that, let us assume two policies π and π', where the latter is known to be better. So if $\pi' > \pi$ then $v_{\pi'}(s) > v_\pi(s)$ must hold for all $s \in S$ [10]. This is saying that the expected return must be greater for all states to call a policy better than another. As stated above, there is at least one optimal policy, which means that there can be multiple optimal policies, which we denote π_*. Them being equal to each other and better than every other policy leads to a shared *optimal value function*, defined as the following:

$$v_*(s) \doteq \max_\pi v_\pi(s), \qquad \forall s \in S \tag{8}$$

Similarly a shared *optimal action-value function* exists, defined as:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a), \qquad \forall s \in S \wedge \forall a \in A(s) \tag{9}$$

[10]. Having defined the optimal action-value function q_* we can now easily define the optimal policy because given a state, we can compare the values of the available actions using q_* and pick the one with the highest value. Mathematically we can write that as the following:

$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{10}$$

In section 2.3 we derived the Bellman equations, which made the recursive property of v_π and q_π apparent. Similar to them there exist the *Bellman optimality equations*, which give rise to the recursive properties of v_* and q_*. [10] [9]

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \tag{11}$$

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{12}$$

Once again, we can form a backup diagram to represent the equations graphically

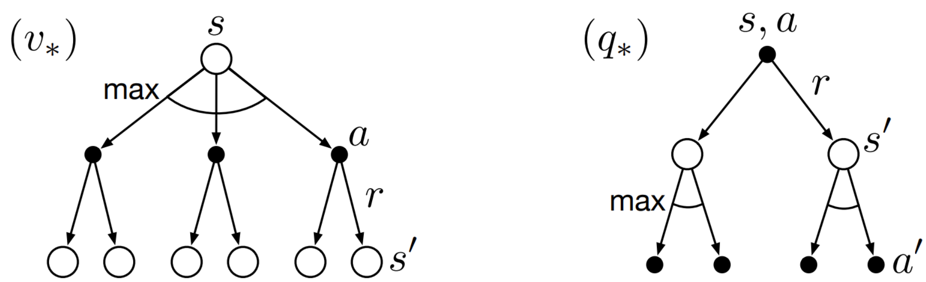

Figure 5: Backup diagram for v_* and q_* [10]

As one can see, these are nearly identical to our previous backup diagrams for v_π and q_π. The only thing that has been added are the arcs, which indicate that the optimal action (i.e., the one with the highest expected return) is being taken, rather than following a given, possibly suboptimal, policy.

3 Tabular Q-Learning

This section will be looking at a first solution method, called *Tabular Q-Learning*. To do so, we will first discuss the algorithm itself and then actually use it in two different experiments. By doing so, we will get a better insight into how the algorithm works and what one has to consider when implementing it.

3.1 The Algorithm

The Q-Learning Algorithm was introduced by Watkins in 1989 [14], which is to this day still seen as a significant breakthrough in the field of reinforcement learning. The goal is to approximate q_* by incrementally updating the action-value function Q. This is done by the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)], \tag{13}$$

where α is the *learning rate*, sometimes also referred to as *step-size*, which defines how drastically we update our estimates. The learning rate is limited to an interval, $\alpha \in]0, 1]$. The smaller α is, the more conservative the learning is. It will rely more on old estimates rather than adopting new information. Setting the learning rate to 1 is the complete opposite. It will overwrite its old estimate. If we rearrange (13) just a little bit, it will become clearer why this is true.

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] && \text{(from (13))} \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a)] - \alpha Q(S_t, A_t) \\ Q(S_t, A_t) &\leftarrow (1 - \alpha) \cdot Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a)] && (14) \end{aligned}$$

As visible in (14) we take $(1 - \alpha)$ of our old estimate and combine it with α of the term in brackets, which at first might not look all too familiar. But if compared to the Bellman optimality equation for q_* (12), we can see where the update rule stems from. So the Q-function will continue to be updated using just one value that is known to be right (R_{t+1}), since the agent, receives it as feedback from the environment until it has completely converged to q_* because then everything left from the arrow is precisely equal to everything on the right side. Note that updating, not only using actual experience but also using current estimates, is called bootstrapping [10]. One value of Q is updated using another value of it. Furthermore, this update rule does not rely on the current policy being followed by the agent. This property is referred to as off-policy learning [10] and it can be quite useful when implementing the algorithm because it does not matter how the agent behaves. As long as every state-action pair is visited often enough, Q will converge to q_*. Watkins proved its convergence in 1992 under certain conditions [13]. However, due to time limitations, this work will not cover the how and why of the convergence.

With this update rule and enough data (experience), the agent can approximate q_*, which he can then use as a policy by always picking the action with the highest value (10). In other words, if one knows q_* he knows π_*. But in order for an agent to obtain enough data, he must also have a policy to operate in a given environment. One could argue to always act greedy with respect to Q, in other words, to always pick the action with the currently highest estimated value. However, when looking at a simple example, it becomes clear why that might not be the best choice.

Imagine you are visiting a city for the first time, and you have to choose a restaurant for every meal since you cannot cook. Your initial estimates of the restaurants are neutral ($Q(\cdot) = 0$), so you randomly chose one of them for the first meal. It happens to be a pleasant experience, and you update your estimate for choosing that restaurant positively. Following the greedy policy with respect to your current estimates (Q), you would continue to pick that same restaurant indefinitely. However, for you, there is no knowing whether there is a restaurant of higher value.

The problem demonstrated by the example is that if an agent was to always *exploit* its current estimates and never *explore* other actions, it would often get stuck on a suboptimal policy and action-value function. A well-known method to address this problem is to use an ε-greedy policy during training [10]. When picking an action using ε-greedy, the action will be chosen at random with probability ε and with probability $1-\varepsilon$, the action will be greedy with respect to the current Q. Thus there will always be some amount of exploration depending on the parameter $\varepsilon \in]0, 1]$.

When using the *Tabular* Q-learning algorithm, the Q-function will be represented by a look-up table (i.e., array) of size $S \times A$. Thus it will contain a numerical value (current estimate) for each possible state-action pair and is usually referred to as the Q-table [10].

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_n
s_0									
s_1									
...									
s_m									

Figure 6: Q-table for the Tabular Q-learning algorithm

The number of actions and states in Figure 6 have no meaning at all. They were chosen arbitrarily to fit nicely. The Figure's sole purpose is to give a graphical representation of the Q-table. In every empty cell belongs the current estimate of the according state-action pair. So every time the Q-function is called, the value is just looked up in the Q-table. Having discussed all that, here is the complete algorithm that was used in the following experiments.

Algorithm 1 Tabular Q-learning algorithm [10]

Parameters: learning rate $\alpha \in]0, 1]$, exploration rate $\varepsilon \in]0, 1]$

Initialize $Q(s, a)$, for all $s \in S, a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

for each episode **do**

  Initialize S

  for each step of episode **do**

    Choose A from S using policy derived from Q using ε-greedy

    Take action A, observe R, S'

    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

    $S \leftarrow S'$

  end for if S is terminal

end for

3.2 Gridworld

The primary purpose of this experiment is to gain a better understanding of how the Q-learning algorithm works. Also, it will become clear how vital the adjustment of the different parameters can be.

3.2.1 The Environment

I created a relatively simple gridworld environment that looks like the following:

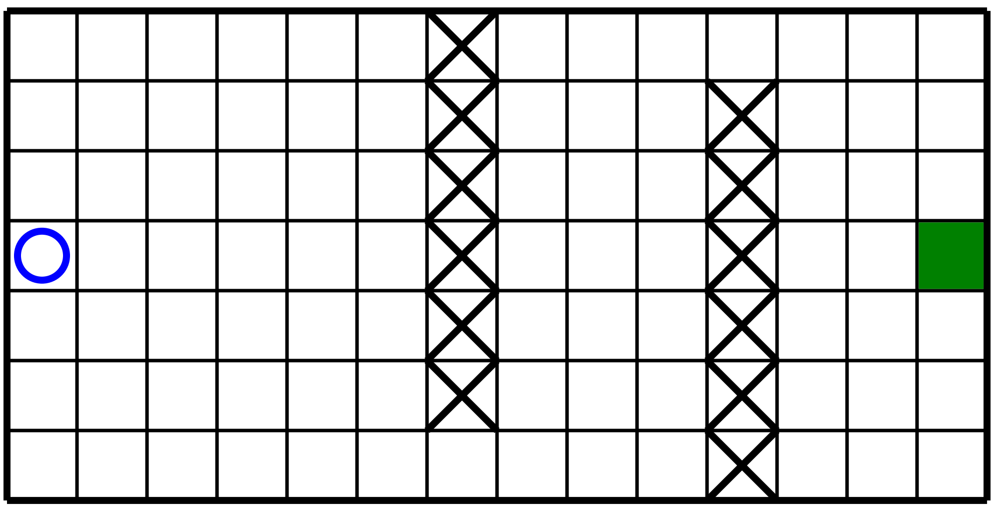

Figure 7: Gridworld environment

In Figure 7 we see the initial state of the environment, where the blue circle is the agent, whose goal is to reach the green rectangle while avoiding the walls represented by the crossed cells. He can move up, down, right, and left, with a reward of -1 on every transition except on the following:

- trying to step out of the grid will result in no movement and a reward of -10
- stepping into the walls will result in death (i.e. termination of the episode) and a reward of -100
- stepping into the green rectangle will also terminate the episode, but with a regular reward of -1 for the transition.

In this setting, the shortest paths between the agent's starting cell and the green cell will result in a total undiscounted reward of -25. Note that there is not one shortest path but multiple. For example, in order to pass the first wall, it does not matter whether you first adjust your vertical position or your horizontal one. There is a shortest path for both options. Contrary to that, there is only one optimal (i.e., highest) score you can achieve, which, as mentioned earlier, is -25. The beauty of this example is that Bellman's principle of optimality (Section 2.5) suddenly becomes quite apparent. There are multiple optimal policies (paths), but there is just one optimal value and action-value function since all optimal policies end up achieving the same optimal score. To clarify, by score, I mean the sum of undiscounted rewards achieved during an episode.

The next important thing is the state-space of the environment. This can significantly impact performance, especially when using a tabular solution method, as the next experiment will demonstrate. Luckily in this experiment, our gridworld is already limited to $7 \cdot 14 = 98$ states because only that many cells are available to the agent. There are multiple ways of how the agent might observe these different states, but the simplest one is probably as coordinates. Since we will be using a 2-dimensional array (i.e., matrix) to represent the environment, it is most comfortable for the coordinates to have the form (row, column) starting with cell $(0, 0)$ in the upper left corner. The agent's goal would thus be located at $(3, 13)$.

3.2.2 Implementation of the Algorithm

Having defined our environment's most important properties, we can get to solving it by implementing Algorithm 1. Since this is a rather simple environment, our goal should be for the agent's Q-table to converge to q_* and do so in as few episodes as possible. The only difference one can make as the developer is adjusting the different parameters of the algorithm and finding the best ones. Before we start our search, it is important to note that there are actually two additional parameters than the algorithm initially suggested. The discount rate defines how farsighted the agent should be, but due to the simplicity and lack of uncertainty in this environment, we will always set $\gamma = 1$. And then, there is the initial value of each state-action pair, which can be chosen arbitrarily except for the terminal states. From this point forward, I will refer to that value as $q0$. To clarify, by performance, at least in this simple experiment, I mean the number of episodes it takes the agent's Q-table to converge to q_*. The fewer episodes it took, the higher the performance. Using the ε-greedy policy, it is not wise to track the score achieved by the agent on each episode and call that his performance. That is due to the fact that with probability ε he will choose a random action, which can lead to a suboptimal score, even though his Q-table might already have converged to q_*. But in this simple environment we can easily calculate the optimal state-action values for the starting state ($s_0 = (3, 0)$). So we could compare them to the agent's current estimates for s_0. After some testing, I found that if they are equal ($Q(s_0) = q_*(s_0)$), it is pretty safe to say that the Q-table has converged close enough to q_*. Note that the Q-table's values will only be calculated to eight decimal places and then rounded. To demonstrate that, I let an agent with arbitrary parameters play until $Q(s_0) = q_*(s_0)$ and at that point, I saved his Q-table and rendered his greedy policy with respect to his Q. Here are the results:

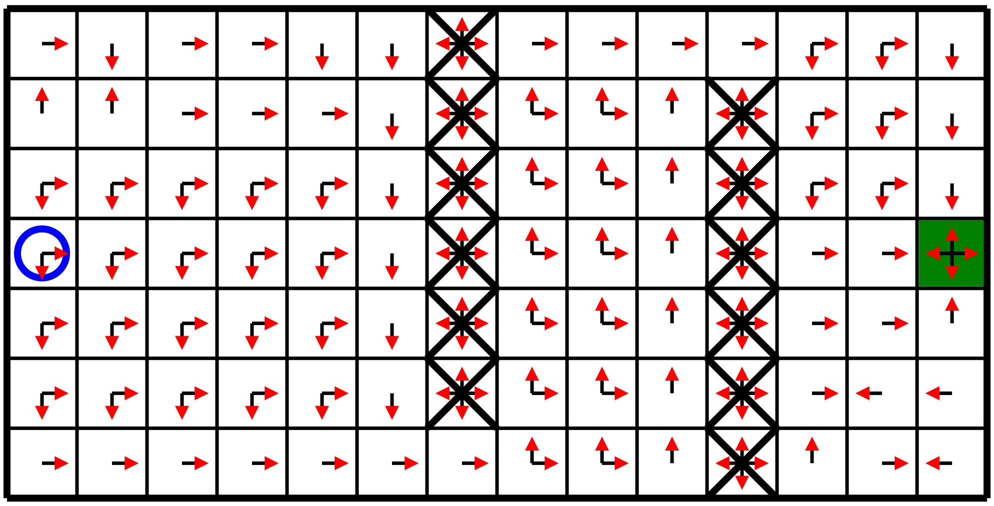

Figure 8: Gridworld, greedy policy after 4490 episodes, $\varepsilon = 0.1, \alpha = 0.5, q0 = 0, \gamma = 1$

The red arrows represent the action with the highest estimated value. If there are two in one cell, then they share the same value. With the exception for the upper left and lower right corners, the agent's Q-table did converge to q_* and thus he has found π_*. And since these corners are never visited, when following π_* they are negligible. To conclude we can assume that the agent's Q-table has converged close enough, as soon as $Q(s_0) = q_*(s_0)$.

3.2.3 Parameter Optimization

As mentioned earlier, our goal is to find the best set of parameters. To do so, we start with arbitrary ones and then adjust one parameter at a time to see how it affects performance. Let us first take a look at ε. After choosing different values for ε, I ran 20'000 episodes 100 times and averaged the performance over those 100 times for each value of ε.

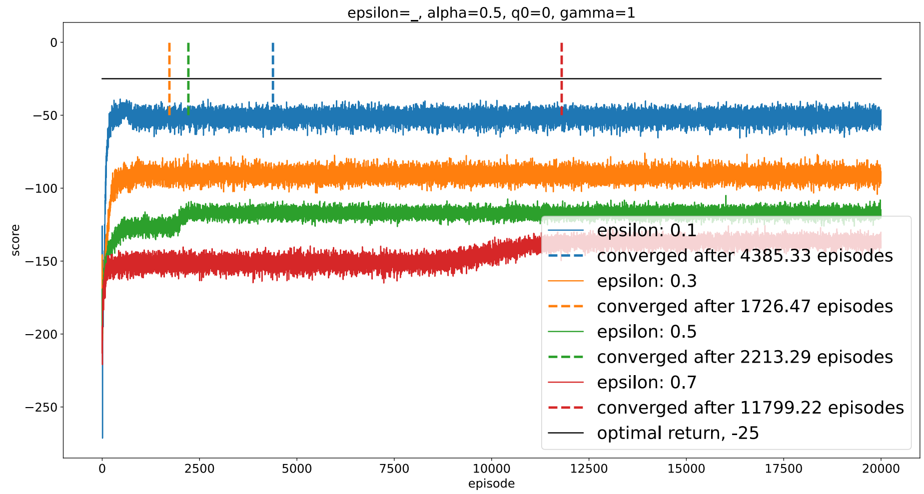

Figure 9: Gridworld, effects of adjusting ε

As visible in Figure 9, there are always 2 different metrics plotted for each ε (color). The dashed vertical lines mark the episode at which the Q converged close enough to q_* as discussed earlier, and other lines show the score of each episode during training. Note that the score achieved during training is not crucial. More important is the amount of episodes it took for the Q-table to converge, which is always clearly visible in the legend. Moreover, I also plotted a horizontal black line for reference, which shows the optimal score of -25. The results suggest a sweet spot around $\varepsilon = 0.3$ at which the Q converges fastest. Note the difference in scores, which is a direct result of a higher value for ε, since the agent is only allowed to act optimally with probability $(1 - \varepsilon)$.

Next up I adjusted α and once again ran 20'000 episodes 100 times for each value of α, here the results:

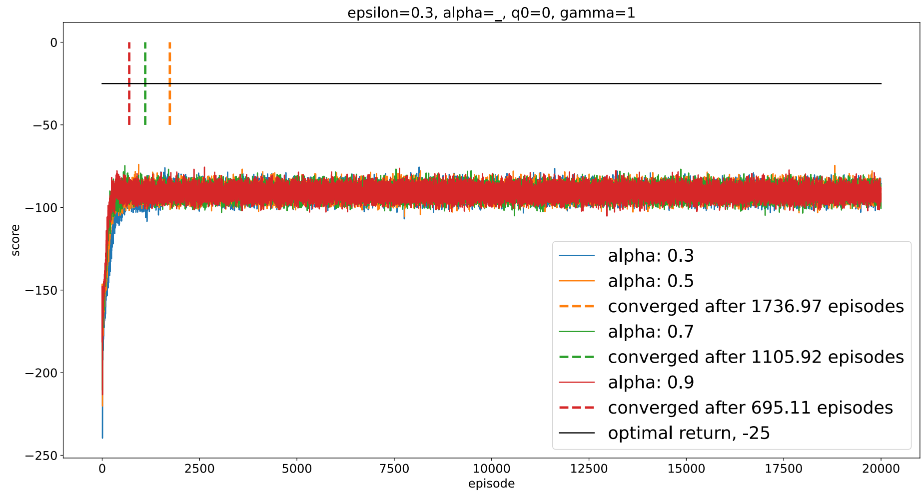

Figure 10: Gridworld, effects of adjusting α

The first thing to notice is that with α set to 0.3, the Q-table did not manage to converge close enough within 20'000 episodes, but as we increase α, the performance also increases, which suggests that if we had set α even higher than 0.9, the performance would have increased as well. However, for this experiment, we do not necessarily seek the absolute best performance. Therefore we will stick to $\alpha = 0.9$.

Last but not least let us adjust $q0$. Again I ran 20'000 episodes 100 times for each value of $q0$.

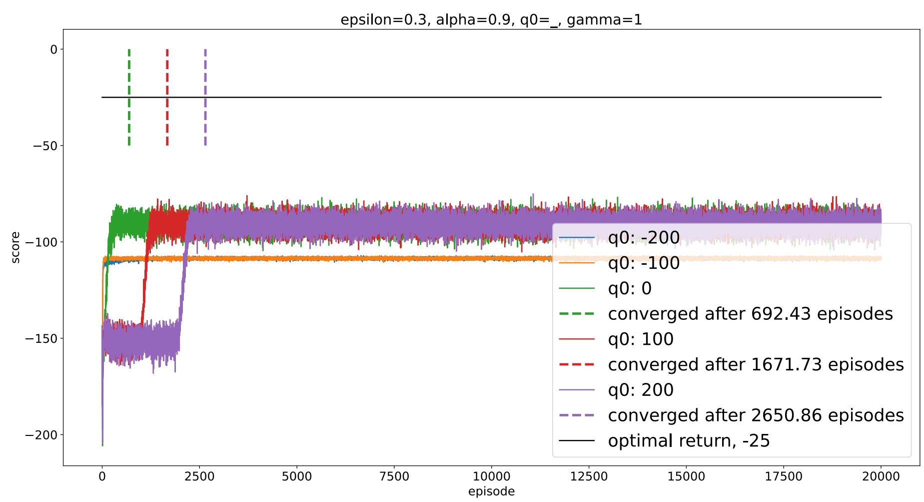

Figure 11: Gridworld, effects of adjusting $q0$, I

After I saw this plot, I was unsure why $q0 = -100$ and $q0 = -200$ did not converge at all. My first thought was, if the initial estimates were too close to a worst-case scenario, e.g., walking right until he steps into the wall, the agent would see the worst-case scenario as a good solution. At that point, it is up to the 0.3 (ε) probability of taking a random action to completely avoid the walls and reach the green rectangle often enough, to realize that it is a better solution. Therefore I reran it with different values for $q0$.

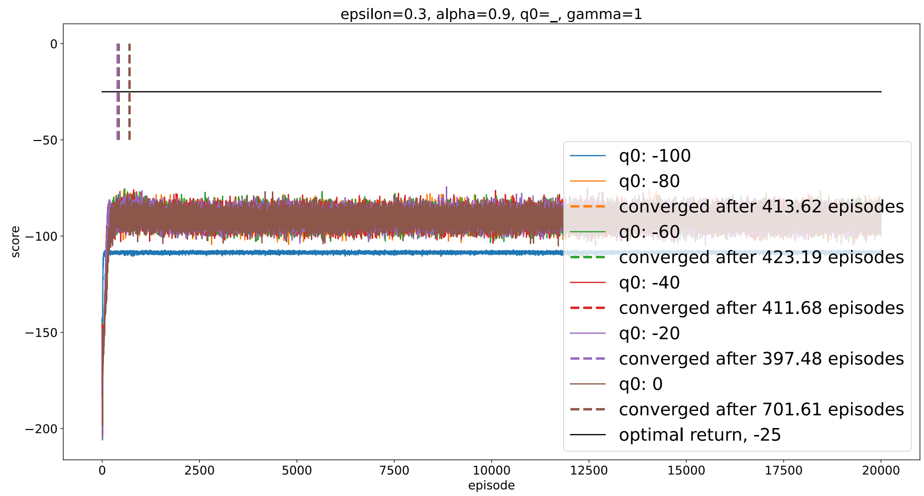

Figure 12: Gridworld, effects of adjusting $q0$, II

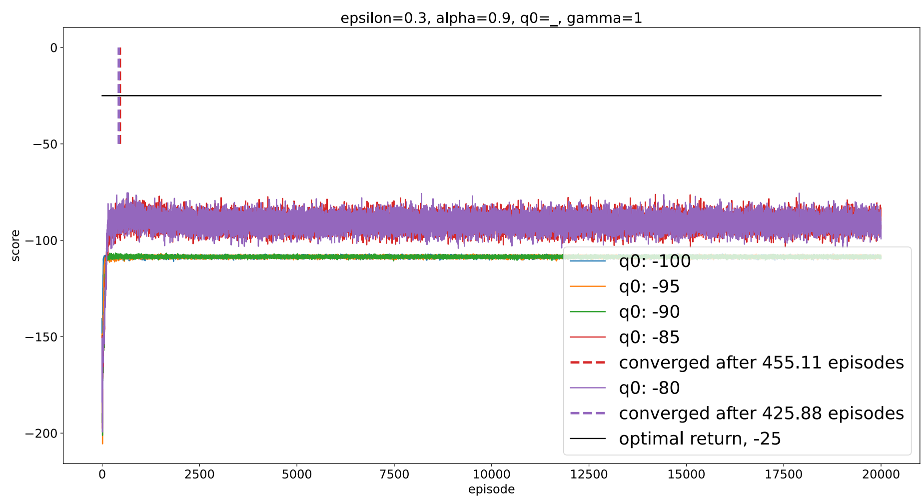

Figure 13: Gridworld, effects of adjusting $q0$, III

After analyzing Figure 12 and 13, my earlier stated hypothesis might be true. To investigate this further, I once again rendered the greedy policies for $q0 = -85$, were it on average converged after about 455 episodes, and also for $q0 = -90$, where it did not converge within 20'000 episodes.

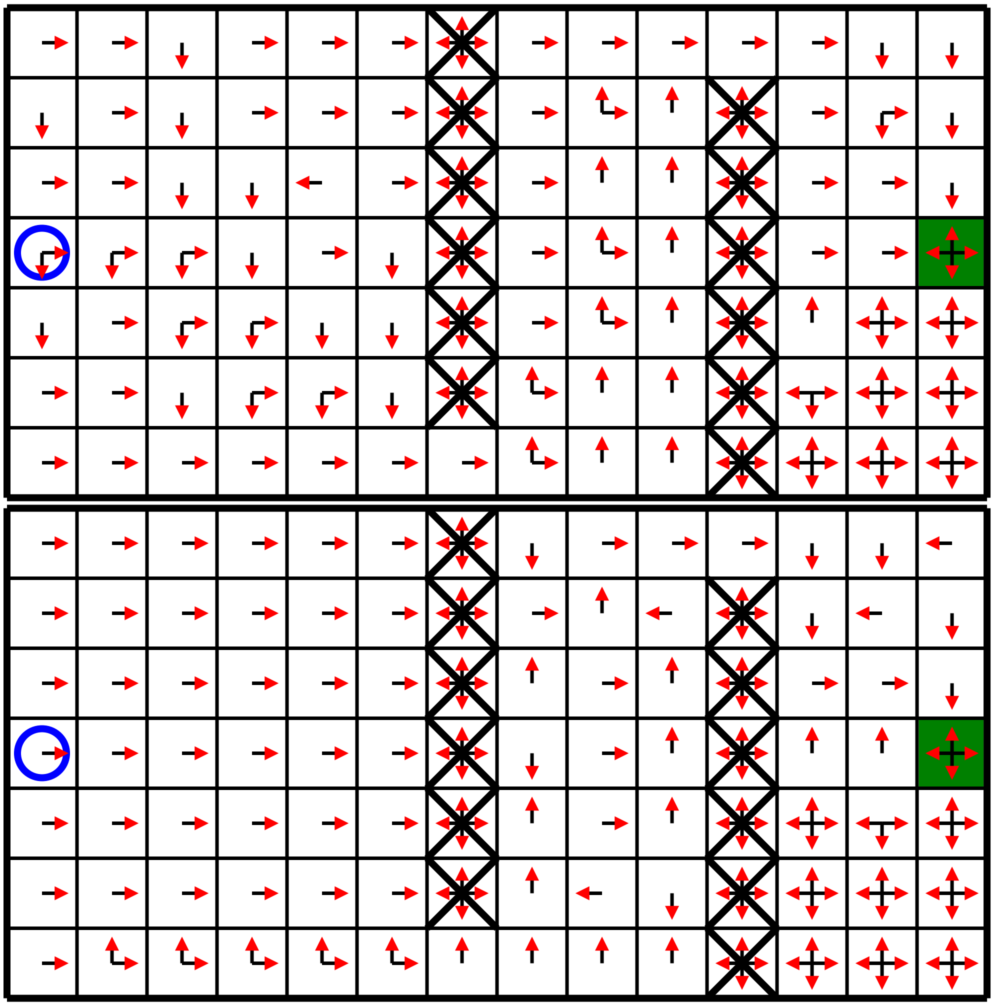

Figure 14: Gridworld, greedy policies, top: $q0 = -85$ after 430 episodes, bottom: $q0 = -90$ after 20'000 episodes, both: $\varepsilon = 0.3, \alpha = 0.9, \gamma = 1$

As visible in Figure 14, for $q0 = -85$, the greedy policy with respect to Q after only 430 episodes is an optimal one, whereas for $q0 = -90$, it is clearly suboptimal and it prefers to walk straight into the walls. This loosely confirms my hypothesis. To sum up, we can safely say that having initial estimates below or near a worst-case scenario can lead to a suboptimal policy and should be omitted in this environment. Thus, to be on the safe side, one should start with somewhat optimistic estimates.

3.2.4 Takeaway

To conclude this experiment, I want to emphasize how crucial it is to optimize the parameters to the best of one's ability. It really can make a significant difference in the learning performance, as we saw with $q0$ in this specific environment.

3.3 Cartpole

This section will be concerned with my first hypothesis, which states: *One of the most basic solution method of reinforcement learning, namely tabular Q-Learning, is adequate to solve the Cartpole environment, provided by OpenAI-Gym [1].*

The Cartpole environment consists of a pole, which is attached to a cart. The goal is to prevent the pole from falling over. To do so, there are two actions available to the agent, either pushing the cart to the right or to the left.

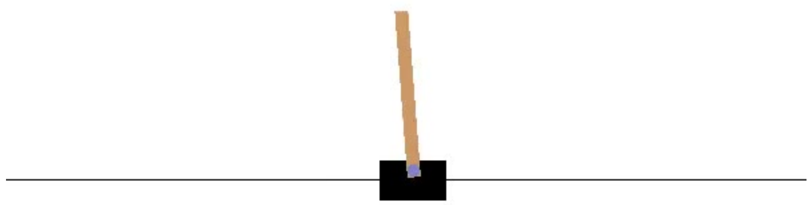

Figure 15: Cartpole environment provided by OpenAI-Gym [1]

The state-space of this environment was described by OpenAI [1] as follows:

Table 1: Cartpole State-Space

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418 rad	0.418 rad
3	Pole Angular Velocity	-Inf	Inf

As we can see it consists of four numbers, which each have a specific range. Hence we are dealing with a continuous state-space, which cannot be represented in a finite Q-table. Thus we somehow have to transform the given continuous state-space into a discrete one. This is actually less complicated than it might sound. Let us take the Cart Velocity as an example, which *theoretically* has a range from -inf to inf. But after letting an agent, who picks actions at random play 2500 episodes, I created the following plot containing the observed Cart Velocity at each time step.

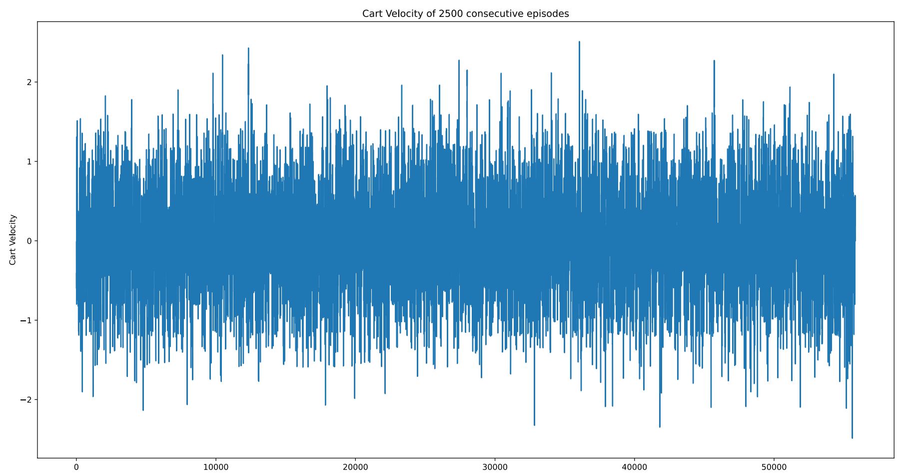

Figure 16: Cartpole, observed Cart Velocity

As one can see, in practice, the Cart Velocity only really ranges from -2 to 2. In addition to that, we have to discretize the received observation. We do that by splitting up the range from -2 to 2 into smaller chunks. For example, we say everything between -2 and -1.8 is put into the first chunk (0), everything between -1.8 and -1.6 into the second chunk (1), and so on. By doing that, we end up with a finite state space, which can then be represented in an agent's Q-table. Note that the number of chunks we divide the continuous space into is also a parameter, as well as the smaller adjusted range for each observation. Also note that when implementing it, the first chunk would actually range from -inf to -1.8 to omit any errors, as the Cart Velocity could theoretically go below -2. The same goes for the last chunk, which would range from 1.8 to inf.

However, before we dive any deeper into actually solving the environment, we have to define what it means. OpenAI considers it solved if the average return is greater than or equal to 195.0 over 100 consecutive trials [1]. Their requirements do not make it clear whether it has to be achieved during training or in testing, because in training the agent acts ε-greedy, whereas in testing we let him chose his actions greedily. I decided that the agent must achieve it during training.

In the gridworld experiment, we used a fixed value for ε during training. Thus the agent will always explore at the same rate during training. However, a well known and also by tests of mine shown to be effective strategy is to let the agent explore a lot initially, and overtime as he gets better, we slowly lower the exploration rate (ε) [4]. This is actually a relatively intuitive strategy when you do not know anything about your environment, it makes sense just to try a bunch of things, and as you get to know it better, you can start to actually do things that you expect to result in a high reward. Another welcomed advantage is that by doing so, we will be able to achieve our required average return during training because as we saw in Figure 9 the achieved score during training is highly affected by ε.

After following the same procedure as in the gridworld experiment to optimize the parameters, I came up with a set of parameters, with which the agent was able to solve the environment in only 164 episodes.

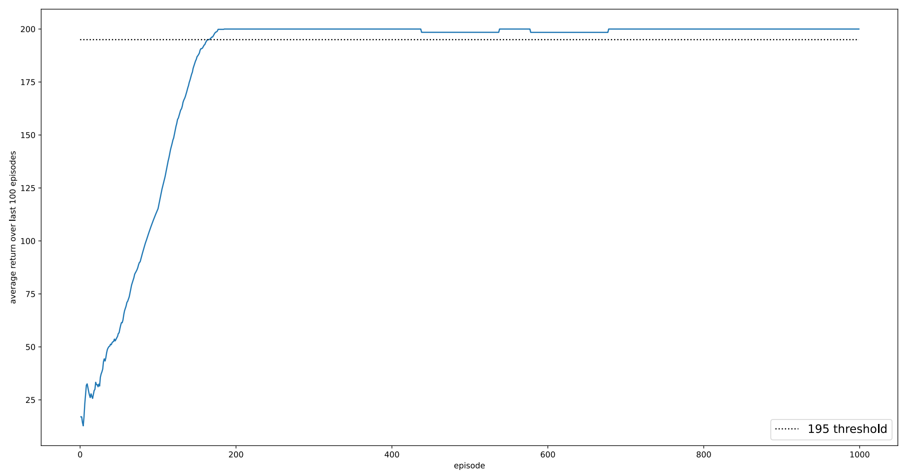

Figure 17: Cartpole, solved after 164 episodes

As visible in the plot above the agent did successfully solve the environment. The Following parameters were used (in Source Code E) to achieve the results.

- obs_high=[2.4, 2, 0.21, 1.8]
- obs_low=[-2.4, -2, -0.21, -1.8]
- obs_chunks=[1, 1, 12, 12]
- q0=0
- gamma=1
- epsilon_function=1/(episode+1)
- alpha=0.1
- S_T_reward=-2000

There are two more things I need to mention. First, there is a parameter called S_T_reward, which I added to give the agent additional punishment when he fails. So every transition into S_T, the terminal state, is punished with a reward of -2000. The second thing worth mentioning is the function I used to slowly decrease epsilon over time. When plotted, it looks like the following.

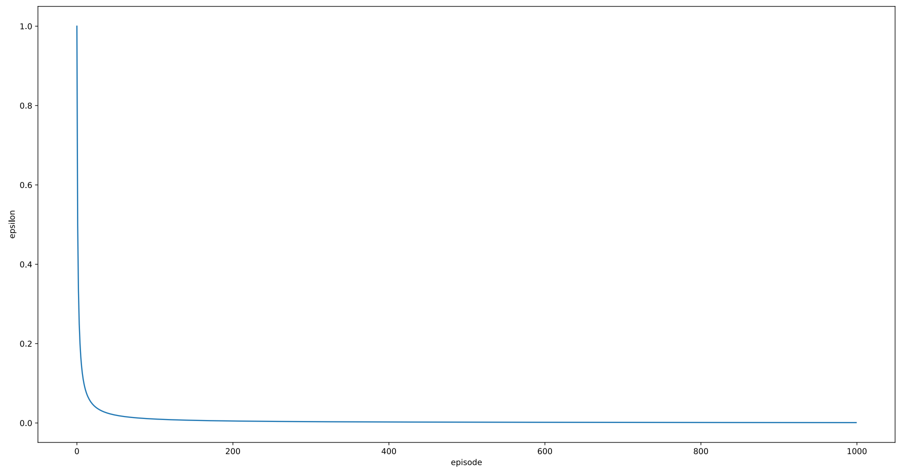

Figure 18: Cartpole, decreasing epsilon

The agent starts by only exploring, but he then very quickly decreases his exploration rate. After 100 episodes, his probability to explore is below 0.01, which is vital to solving the environment since we want him to reach the threshold during training. If the exploration rate was, for example, 0.1, he might fail every 10^{th} episode, which would lead to not reaching the required threshold.

With these results, I can confirm my first hypothesis, which states: *One of the most basic solution methods of reinforcement learning, namely tabular Q-Learning, is adequate to solve the Cartpole environment, provided by OpenAI-Gym [1].*

4 Deep Q-Learning

In the previous section, we used a simple look-up-table to represent the Q-function. This approach worked fine for the former simple experiments since they had a rather small state-action space. For the Gridworld the look-up-table had a size of $7 \times 14 \times 4 = 392$ and for the Cartpole experiment it was $12 \times 12 \times 2 = 288$. In this section, the goal will be for an agent to learn how to play Snake, which will have a way bigger state-action space. Therefore we are switching from a tabular to a function approximation method, namely Deep Q-Learning. This will allow for greater complexity and also reduce the computation needed. [8] [9, Lecture 6]

4.1 Neural Networks

The currently most popular function approximator for reinforcement learning must be the neural network. Note that by neural network, this work refers to the artificial one and not to the biological one.

4.1.1 The Structure of a Neural Network

A neural network is essentially just a function that can be scaled to arbitrarily many input and output dimensions. It is often thought of as very complicated, but as we start to break it down, one will quickly realize that it is easy to understand how the function works. Let us first look at a small neural network to understand what it does. Note that this work only considers linear layers for the neural network.

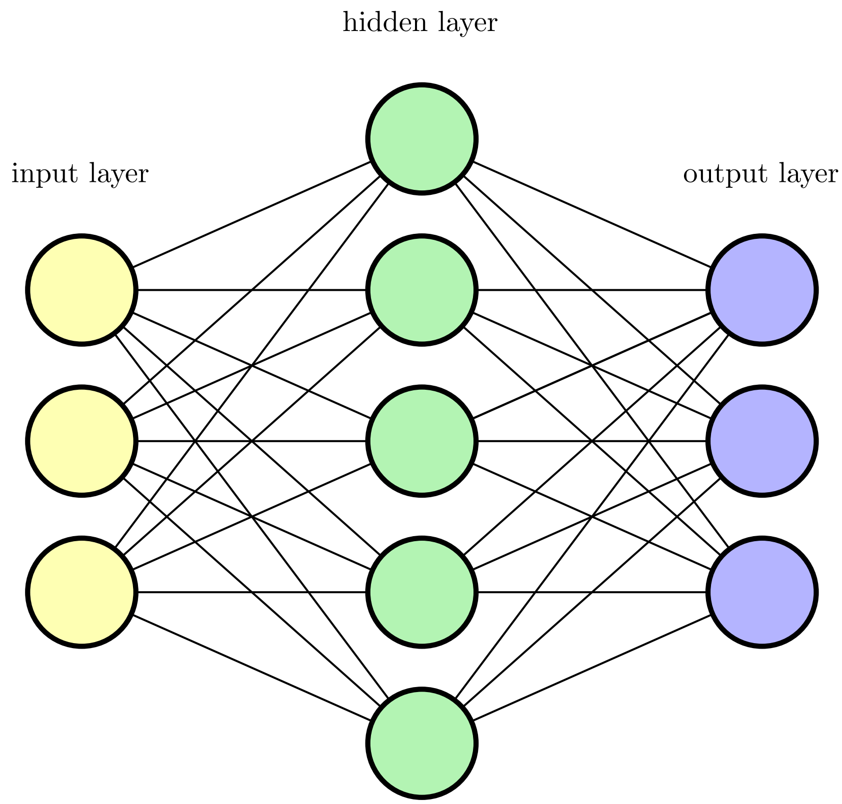

Figure 19: Simple neural network

Figure 19 shows a simple neural network consisting of three layers: an input, a hidden, and an output layer. To understand how a forward pass through the neural network works, we will zoom in on one neuron of the hidden layer.

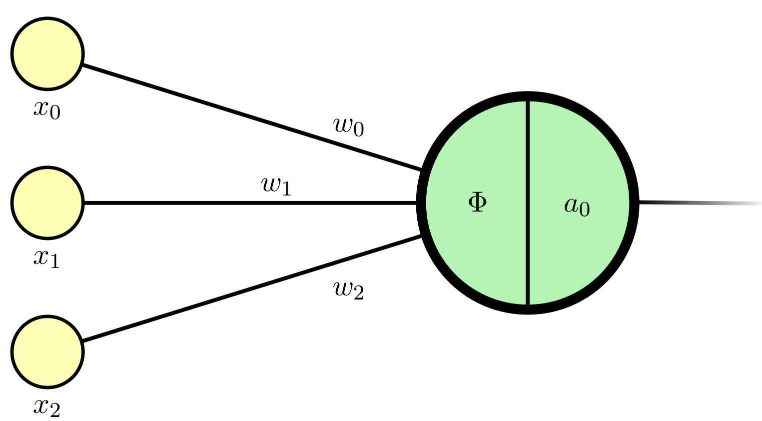

Figure 20: Zooming in on one neuron

Visible in Figure 20 we can see said neuron, which has three incoming values x_0, x_1 and x_2, which all have an according weight w_0, w_1, w_2. Now the first thing that is calculated as these values are passed to the neuron is Φ. That is done by multiplying every input value by its according to weight and adding a bias β afterward. To write it down formally, it would look like the following:

$$\Phi(\vec{x}, \vec{w}) = \sum_i x_i w_i + \beta = \langle \vec{x}, \vec{w} \rangle + \beta \tag{15}$$

As one can see, there are two ways to write Φ. The first is just a simple summation as described above, but in the second one, we build the dot product of the vectors $\vec{x}$ and $\vec{w}$ and add a bias β afterward, which will make things easier. In this example, these two vectors would be defined like this:

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}, \quad \vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

Here I want to define one more, later important, vector θ as follows:

$$\vec{\theta} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \beta \end{bmatrix}$$

This vector contains all the parameters of the neural network. In this case, there are only four, as we are only working with a specific neuron of the neural network and only considering what happens there, but the concept and the math behind it could quickly be scaled up to the whole network in Figure 19.

4.1.2 Activation Function

Now we know how to calculate Φ for a certain neuron, but there is one more step to get to a_0, the actual value that the neuron passes on. We have to use an *activation function* on Φ. The primary reason for that is to add non-linearity to the neural network. Without a non-linear activation function, all of the linear layers, no matter how many there are, of a neural network would collapse into a single linear layer. Furthermore, non-linearity can be extremely important when approximating a more complicated function. Some functions simply cannot be approximated well through a linear approximator. Note that we will not be using any activation function on our output neurons, as they are the Q-values for a given state. And if we were to use an activation function on them, we would limit their ability to accurately approximate the Q-value as an activation function often squishes or cuts off a given input.

There are many activation functions to choose from. The rectified linear unit or ReLU for short is a prevalent choice and was also proven to work rather well [2]. Moreover, it is a straightforward function. It merely takes the maximum between 0 and the given input. If one were to plot it, it would look like the following.

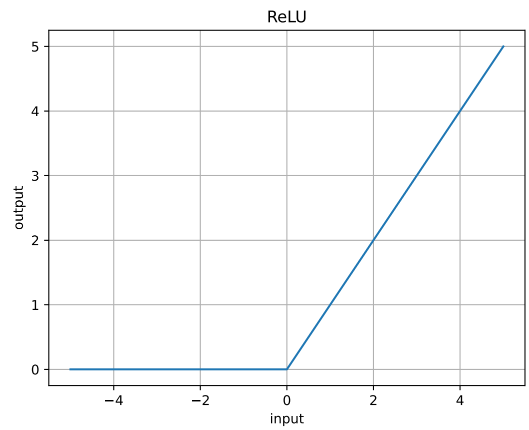

Figure 21: The ReLU activation function

Formally we can define it like this,

$$R(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

or even simpler,

$$R(x) = \max(0, x) \tag{16}$$

We can now finish our calculation of a_0 by simply passing Φ through R, thus:

$$a_0 = R(\Phi(\vec{x}, \vec{w})) = \max(0, \langle \vec{x}, \vec{w} \rangle + \beta) \tag{17}$$

4.1.3 Optimization Algorithm

Now we know how a neuron transforms input signals into a single output signal. With that, we essentially know how a forward pass through a neural network works, because the same calculations happen in every neuron. But as mentioned earlier, our goal is to approximate a given function and to do so, we rather obviously need some way to adjust the parameters $\vec{\theta}$ so that the accuracy of the neural network, when compared to the given function, improves. Note that said function is usually unknown. Only data points are given. To improve, we first have to know how much, if at all, the approximation is off. We need a so-called *loss* *function* that gives us a numerical value to measure the accuracy. Once again, there are many to choose from. We will simply take the squared error as our loss function.

For the following we have to define one more term, $\hat{y}(\vec{x}, \vec{\theta})$ the output vector of a neural network dependent on the input $\vec{x}$ and the parameters $\vec{\theta}$.

With that we can now properly define our loss function as follows.

$$L(\vec{\theta}, \vec{x}, \vec{y}) = \sum_i (\hat{y}_i(\vec{x}, \vec{\theta}) - y_i)^2, \tag{18}$$

where $\vec{y}$ is the target output vector for our input $\vec{x}$. By squaring the difference between the approximated output and the target output, L will always return a positive number and return 0 if the neural network perfectly approximated the target function for the given input. Hence we want to choose the parameters $\vec{\theta}$ to minimize L. If we were to form the gradient $\nabla_{\vec{\theta}} L(\vec{\theta}, \vec{x}, \vec{y})$, which is the vector of steepest ascent w.r.t. $\vec{\theta}$ or in other words how do we have to adjust our parameters $\vec{\theta}$ as to most quickly increase L. A logical thought would be to simply take the negative value of the gradient, $-\nabla_{\vec{\theta}} L(\vec{\theta}, \vec{x}, \vec{y})$, and adjust our parameters with that. This method is known as stochastic gradient descent (SGD). For any given data point $(\vec{x}, \vec{y})$ it can perform a parameter update to lower the loss. Formally the update rule for $\vec{\theta}$ would look like this:

$$\vec{\theta} \leftarrow \vec{\theta} - \eta \cdot \nabla_{\vec{\theta}} L(\vec{\theta}, \vec{x}, \vec{y}) \tag{19}$$

Here we reintroduce a learning rate under a different variable η since its implications differ slightly from the one in Section 3. Here it regulates by which amount the parameters follow the steepest descent from their current position. The adjustment of the learning rate can have a massive impact on performance. That is because all we are trying to achieve with the SGD is to find a good enough local minimum in our loss function since it is, in practice, not feasible to land in the global minimum. However, in order to end up in a decent local minimum, our learning rate cannot be too high, as we would end up overshooting all the local minima, but it also cannot be too low for two reasons: For one, the lower the learning rate, the longer it will take to reach a local minimum and second it might get stuck in a bad local minimum.

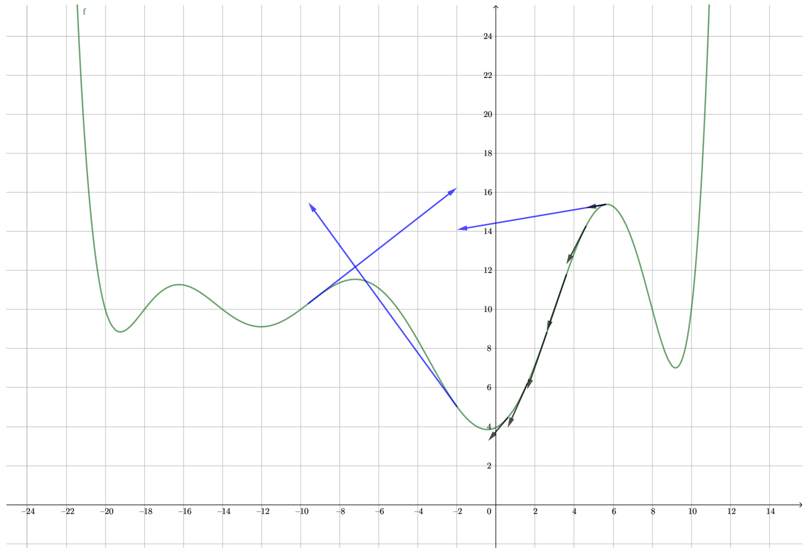

Figure 22: implications of adjusting η

Figure 22 shows how adjusting η affects the search for a local minimum. In blue, we see a terrible choice for η, it is way too high, and it is just jumping around like crazy. In black, on the other hand, we see a close to optimal choice for η. This is obviously all with respect to the given starting point for the parameter. In this case, we only have a single one starting at 5.6. Note that the parameters of the neural network will be initiated mostly at random, and also rather obviously the dimensionality is not to compare. The quite small neural network in Figure 19 already has 38 different parameters to adjust. Thus, its loss function's input space would be 38-dimensional, which is beyond us humans to imagine.

4.2 The Algorithm

One of the first to combine reinforcement learning with neural networks was Gerald Tesauro in 1995 with TD-Gammon [12]. However, probably more significant was the breakthrough from DeepMind when they managed to play several Atari games on a superhuman level through deep reinforcement learning, the combination of deep neural networks and reinforcement learning algorithms [8]. The algorithm I used to play snake only differs slightly from Algorithm 1 and is also very similar to the one suggested by DeepMind [8]. The apparent difference is that we no longer use a look-up-table but rather a neural network to represent our Q-function. Furthermore, in the previous Tabular-Q-Learning algorithm, we used every transition played by the agent only once to update the Q-table. This is very data inefficient and problematic due to the heavy correlation between transitions that happen one after another. Therefore this algorithm uses an experience replay mechanism [8] [6] which stores every transition experienced by the agent in $\mathcal{D}$ for later use. On every step, a specific amount (n) of data points are randomly sampled from the thus far collected data (agent's memory) $\mathcal{D}$ and are then individually used to optimize the parameters of the neural network by using *Adam* [3], an algorithm for stochastic optimization. This algorithm is essentially a better, more optimized version of the previously introduced SDG method. If n data points are sampled, there will be n gradient descent parameter updates per step.

One more thing to mention is the design of the neural network. The size and amount of the hidden layers can be chosen freely, but the dimensionality of the input layer has to match the observed state's dimensionality. Furthermore the output layer must have a neuron for every possible action, or in other words, the output layer will consist of the Q-values for the individual action. This architecture for a neural network is referred to as a Q-Network [8].

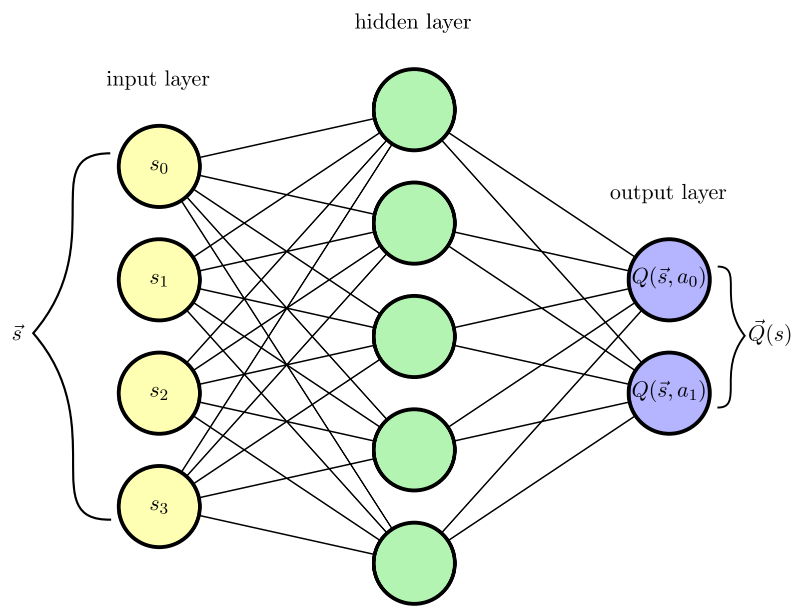

Figure 23: Q-Network architecture, observed state as input and Q-values as output (size and amount of hidden layers may vary)

Having discussed all of that, I can now present the Deep Q-Learning algorithm, which I used in the following experiment.

Algorithm 2 Deep Q-Learning algorithm [10] [8] [3]

Initialize replay memory $\mathcal{D}$ to capacity N

Initialize Q-function with random parameters $\vec{\theta}$

for each episode **do**

  Initialize episode, s_0

  for t=0, T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \max_a Q(s_t, a; \vec{\theta})$

    Take action a_t, observe r_{t+1}, s_{t+1}

    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$

    Sample n random transitions from $\mathcal{D}$

    for each transition $(s_i, a_i, r_{i+1}, s_{i+1})$ **do**

      Set $y_i = \begin{cases} r_{i+1} & \text{for terminal } s_{i+1} \\ r_{i+1} + \gamma \max_{a'} Q(s_{i+1}, a'; \vec{\theta}) & \text{for non-terminal } s_{i+1} \end{cases}$

      Perform gradient descent step on $(y_i - Q(s_t, a_t; \vec{\theta}))^2$

    end for

    $s_t \leftarrow s_{t+1}$

  end for

end for

Notice that for the loss, we only consider the Q-value of the action that was actually taken in that transition and not the whole output vector. That is due to the fact that we only know the immediate reward that followed said action. Thus we can only know the target value for that specific action.

4.3 Snake

This section will address my second hypothesis, which states: *Using deep reinforcement learning, an agent can learn to play the game of Snake on a human-level performance*. In order to test this hypothesis, I implemented Algorithm 2 on a self-made game environment. Therefore this section will first elude to the most important aspects of said environment.

4.3.1 The Environment

Snake is a relatively well-known game [18], but there are often slight differences between different versions. The following is explaining my particular game. Everything plays within a small grid, where a snake has to try and eat a fruit, which spawns and respawns at a random (non-occupied) location every time it gets eaten by the snake. Initially, the snake only consists of a head, but it grows by one body unit with every fruit eaten. At all times, there are four different actions available to the agent: up, down, right, or left. They all result in a movement by one unit. The body will follow the head's movement with a matching delay. The score is defined as the number of body units. Note that in this case, the score and the rewards received by the agent do not correlate in the same way as in the previous experiments.

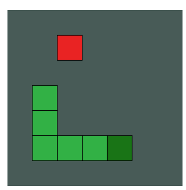

Figure 24: Snake snapshot, score = 5

In Figure 24, we can see a snapshot of the game, where the snake consists of five body units, and thus, the score is equal to five. In this case, it plays in a seven by seven grid, where the maximum achievable score is 48, at which point the game would end due to winning. However, there are two more ways the game can end. Both result from failure. For one, if the head hits the border (i.e., tries to leave the grid), and second if the head hits its own body.

There are two more things to consider when designing the environment. How will the state observed by the agent look like? Which type of actions should be rewarded and which actions should be punished? In this experiment, the answers to these questions will be considered parameters. There will be many options to choose from, especially for the state representation, but more on that later.

4.3.2 The Agent

We will now look at the agent since he has more aspects to consider than those in the previous experiments. The agent is essentially made up of two main components, he has a Q-network, and he knows how to use the Q-network and optimize it based on the implemented algorithm. These two components give rise to the agent's parameters. Following is a list I curated with all the relevant parameters.

- Environment
  - State type
  - Rewards
- Agent
  - Q-Network, size and amount of hidden layers
  - Size of the agent's memory $\mathcal{D}$
  - Discount factor, γ
  - Learning rate, η
  - Exploration rate, ε
  - Number of weight updates per step, n

Now similar to the previous experiments, all one has to do is to try and find the best performing parameters through trial and error. This is easily said, but in reality, this is the part that often takes the longest, especially in experiments where the task at hand is rather complicated, which results in long training times until we see any sort of results from the agent. This is also dependent on the hardware being used.

4.3.3 State Types

We will now look at what I found to be the most interesting parameter, the state type. The question here is, how should the agent perceive his environment. There are two main ways of going about this. One is to give the agent a top-down view, i.e., what we humans see when we play snake. The second option is to embrace the 2D environment and imagine what one would see in it. For example, we could give the agent the distances from his head to the border in eight different directions, and we could also add binary vision for his body and the fruit. Binary vision means that going from his head in a specific direction, is there a fruit or not?

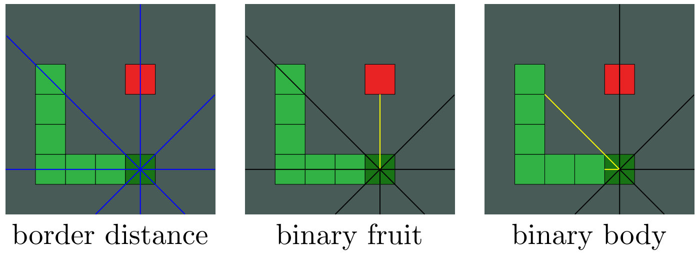

Figure 25: Snake state type, 2D vision

A simple example to illustrate and further understand the second concept is visible in Figure 25. There we can see a graphic for each object included in the vision. In the left one, we can see how the distances to the border are being measured. In the second and third ones, the binary vision is represented by the black and yellow lines. Yellow means that there is an object detected, and black means there is not. In total, this creates an input vector with 24 elements, where every line represents an element. The distances are simply real numbers. The binary vision lines are either a 1 if an object is detected or a 0 otherwise. Note that this is simply an example and this concept does not stipulate the need for eight directions or for the binary vision. It could also be only four directions for the border distances and maybe a distance measure for the fruit instead of a binary one.

We will now quickly look at the different options for the top-down representation for the agent's state. The first one that comes to mind is capturing the whole grid and feeding it to the agent, where different numbers represent the different objects on the grid. However, there are once again many ways to go about this. One that I found to work better than the former is only to show the agent a part of the grid centered at his head. This representation solves two problems with the previous one. First, the need to specifically represent the snake's head is no more since it is always at the center of the agent's visible sub-grid. Second, this state representation is independent of the grid size. This gives the advantage that an agent trained on a five by five grid can also play on a nine by nine or any other size. In Figure 26 one can see an illustration of the just discussed representation, where the agent would only be able to see the yellow highlighted sub-grid centered at the head.

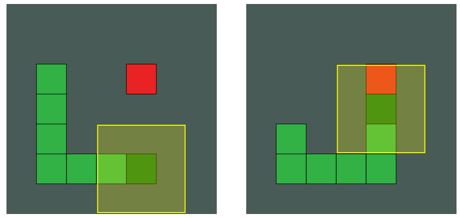

Figure 26: Snake top-down view, 3x3 sub-grid centered at the head

I will now present to you the different state types that were used in the later following results. Note: Blue lines are distance measurements, black/yellow lines are binary vision, yellow highlighted areas represent a top-down view, cyan arrows are 2D vectors from the head to the fruit.

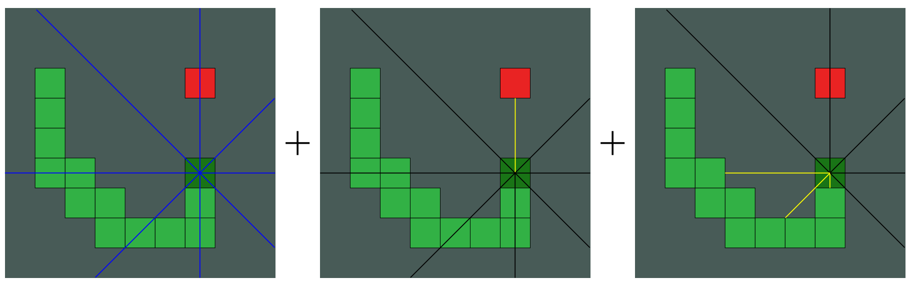

Figure 27: Snake state type #1

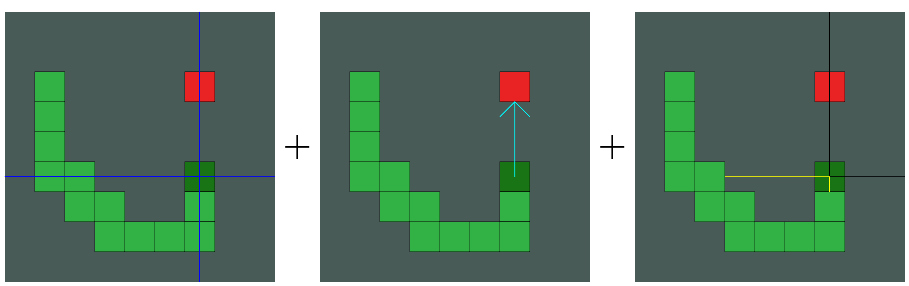

Figure 28: Snake state type #2

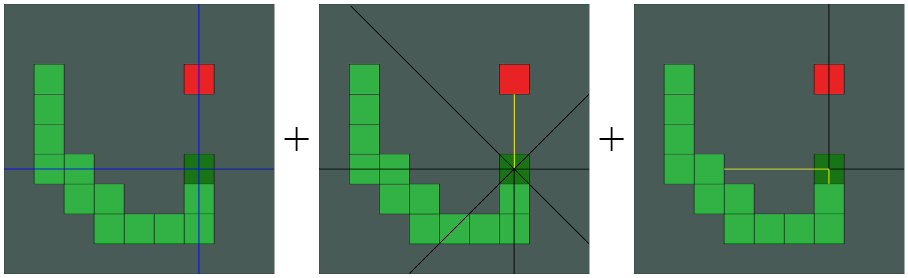

Figure 29: Snake state type #3

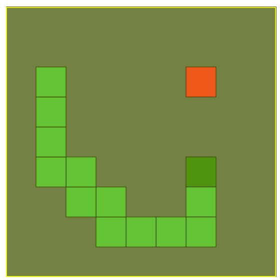

Figure 30: Snake state type #4

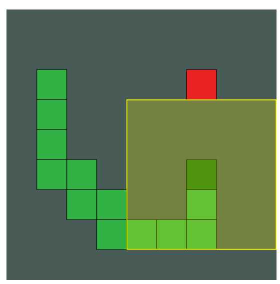

Figure 31: Snake state type #5

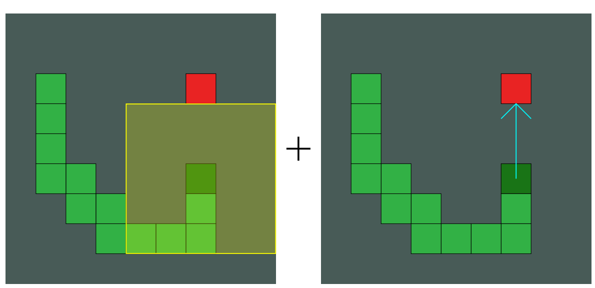

Figure 32: Snake state type #6

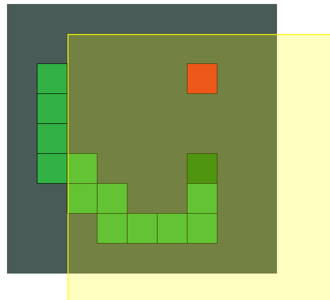

Figure 33: Snake state type #7

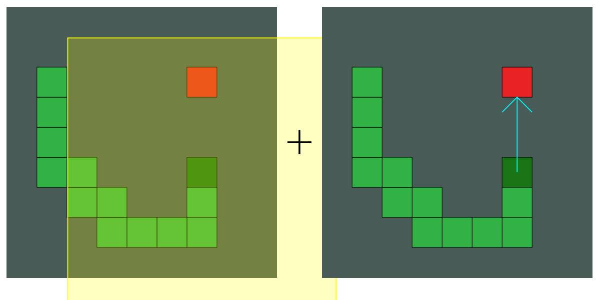

Figure 34: Snake state type #8

4.3.4 Parameter Optimization

I began by optimizing every parameter but the state type to compare the different state types with an optimized set of parameters in the end. Note that the "optimized set of parameters" is by no means optimal. It is merely what I found to work best within a limited time of trial and error testing. Also, I had to choose one state type to optimize the other parameters with and then operated under the assumption that the parameters work similarly well on the other state types. These are compromises I had to make to keep it realistic time-wise. Following is the set of optimized parameters, which I later used to compare the different state types.

- Rewards:
  - -50 for loosing
  - 50 for eating a fruit
  - 100 for winning
  - 0 otherwise
- Q-Network, 1 hidden layer with 128 neurons
- $\mid \mathcal{D} \mid$ = 10'000
- γ = 0.98

- η = 5e-5
- $\varepsilon = (0.9998)^s$, s increments with every step, starting at 0
- $n = 32$, weight updates per step

One further optimization I found to impact performance significantly is to let the agent start his training on a small grid and then work his way up to the final grid size, instead of immediately dropping him into the final grid size. This often causes an increase in performance, because on a small grid the agent is more likely to experience a positive reward (i.e., eating a fruit) than on a large grid. This is especially true in the beginning when the agent acts randomly.

4.3.5 Training Methodology

For each state type, two agents will be trained to check for the run-to-run variance. All 16 of them will be trained in the same manner except for the two with state type #4. The reason being is that an agent with said state type can only play in one specific grid size since we cannot change the input dimensions. Once we create his Q-network with a specific amount of input neurons, for example, in a 3x3 grid, we need nine input neurons, one for each cell. The agent is then bound to that specific grid size as we cannot change the input neurons of his Q-network. That would require initializing a new and untrained one. So all the agents except for the two with state type #4 are trained as follows.

- 200 episodes on a 3x3 grid
- 400 episodes on a 5x5 grid
- 800 episodes on a 7x7 grid
- 1600 episodes on a 9x9 grid

The two agents with state type #4 will spend all 3000 episodes on a 9x9 grid. As one can probably tell, the goal for all agents is to be able to play on a 9x9 grid.

4.3.6 Results

The following is an overview of the agent's performance during training.

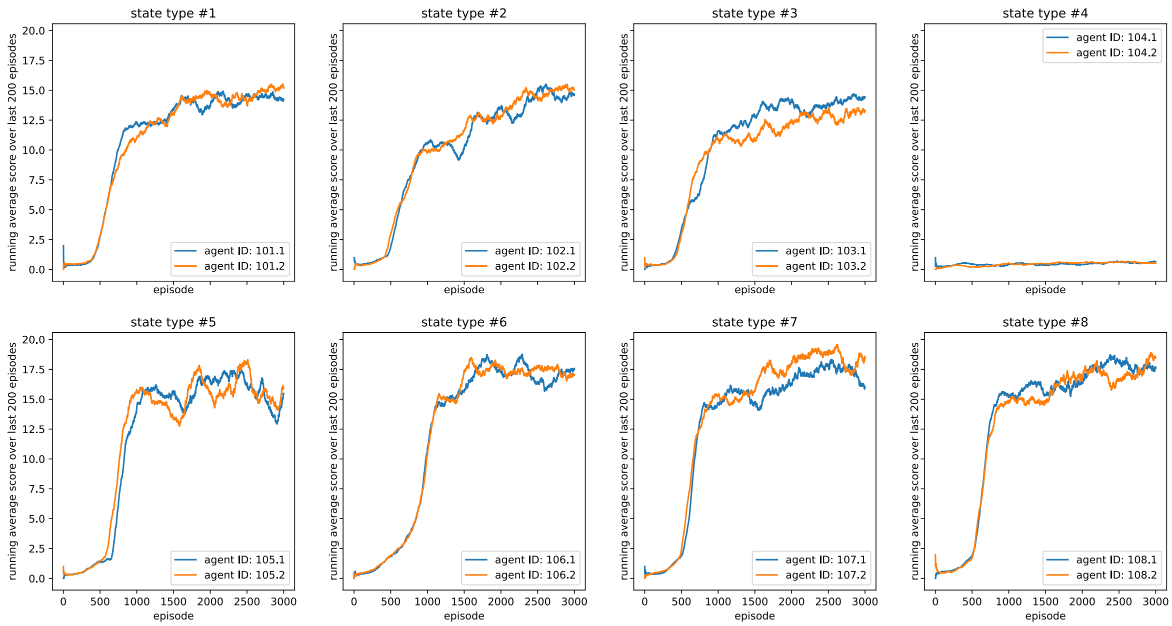

Figure 35: Snake, training performances

From Figure 35 one can already get an impression of the performances. But to thoroughly test the agents, each of them played 1000 episodes in a testing mode. That means the exploration rate is zero, thus they always act greedily. The results from that testing and further information about the agents are presented in Table 2.

agent ID	weight updates in millions	training duration in minutes	high score during training	average score during testing	high score during testing
101.1	18.9	558	28	10.016	26
101.2	17.79	534	31	13.539	28
102.1	12.62	401	32	17.166	33
102.2	12.71	403	29	14.168	32
103.1	17.53	485	28	10.266	34
103.2	21.67	583	27	5.179	28
104.1	49.62	1055	4	0.417	4
104.2	55.07	1287	5	0.325	3
105.1	17.07	436	44	23.590	42
105.2	17.39	444	40	19.288	41
106.1	9.88	283	38	21.846	44
106.2	10.34	293	37	21.812	45
107.1	13.16	304	46	26.652	46
107.2	13.57	314	47	27.575	49
108.1	13.1	303	46	27.101	59
108.2	13.33	308	43	26.844	51

Table 2: Snake, final results

Especially visible in Figure 35, agent 104.1 and 104.2 performed significantly worse. That is partly because the state type did not allow for the previously discussed optimization strategy, where the agent starts on a small grid, which slowly grows to the final grid size. In a way this shows the immense importance of this strategy, but we cannot ignore the fact that the parameters besides the state type were optimized using this strategy. Also the state type itself may simply be a bad performing one. Therefore we cannot fully credit the lack of the optimization for the significantly worse performance.

Besides the failure of the state type #4 there are other notable results we can conclude from Figure 35 and Table 2. We can confidently say that the top-down view overall delivered better results than the 2D vision. We can also conclude that the addition of the vector between the snake's head and the fruit in state types #6 and #8 compared to #5 and #7 resulted in negligible improvements, if any. But, the most important question to answer with these results is the previously stated hypothesis. This is more complicated than anticipated because we somehow have to define human-level performance in the snake game. To do so, I asked 15 people, including classmates and the teacher of a game programming class, to play my specific Snake version. Each of them played 30 episodes, and here are the results.

overall average score (450 episodes)	**best player's average score (30 episodes)**	**high score (within 450 episodes)**
1.1	3.17	19

Table 3: Human test results

From these results, we can conclude that even the worse state types, except for #4, managed to outperform the human test group by a significant margin, especially when we compare our on average best performing agent 107.2, with an average score of 27.575 to the human average of 1.1. With these results I can confirm my second hypothesis.

To further investigate the failure of #4, I was curious to see how the other state types would perform if they were trained on a 9x9 for 3000 episodes. Therefore I trained four more agents with state type #1, #2, #6, and #8. Following are the results presented in an identical manner.

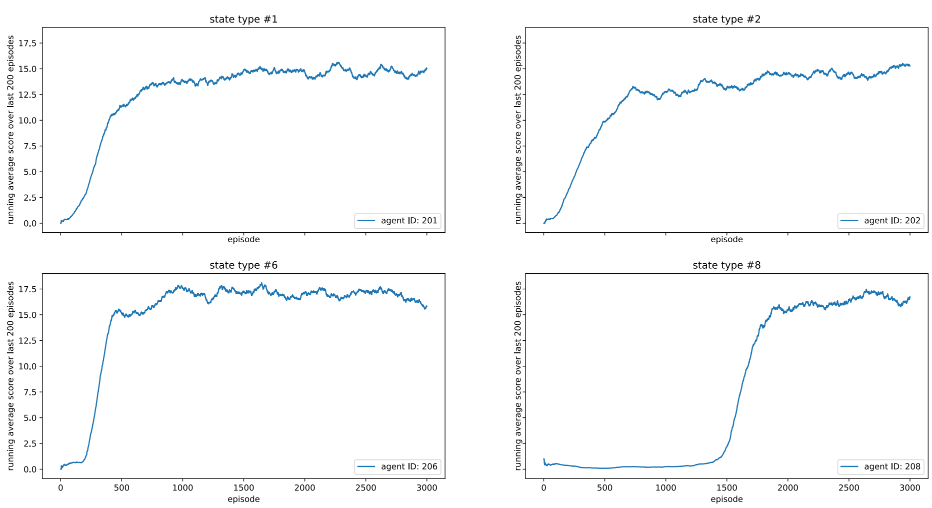

Figure 36: Snake, training performances for the investigation of state type #4

agent ID	weight updates in millions	training duration in minutes	high score during training	average score during testing	high score during testing
201	21.45	622	33	8.081	31
202	13.99	448	29	14.741	29
206	14.49	460	41	22.666	42
208	30.82	846	37	23.264	39

Table 4: Snake, final results for the investigation of state type #4

Figure 36 and Table 4 show how agents 201, 202, 206 performed similar to their counterparts (101.1, 101.2, 102.1, ...). However, with agent 208, we see different behavior, likely due to the large state type. It took him significantly longer to show any sign of learning. Nevertheless, he did manage to perform in the end, though his counterparts (108.1, 108.2) performed significantly better not only during training but also during testing. Nevertheless, we can now safely conclude that state type #4 is simply a bad choice, and its bad performance is not entirely due to the lack of the previously discussed optimization strategy.

5 Conclusion

Motivated by a single question: *How much can a Matura student with hardly any amount of prior knowledge in the field of reinforcement learning achieve in said field within the scope of a Matura Paper?* I sought to answer two hypotheses.

- One of the most basic solution methods of reinforcement learning, namely tabular Q-Learning, is adequate to solve the Cartpole environment, provided by OpenAI-Gym [1].
- Using Deep reinforcement learning, an agent can learn to play the game of Snake on a human-level performance.

As already discussed in Section 3.3, the Cartpole environment is said to be solved if the average score over the last 100 episodes is ≥ 195. By Implementing the tabular Q-learning algorithm (Algorithm 1), the agent solved the environment in 164 episodes. In a paper by Swagat Kumar [5], he managed to exceed the 195 threshold in 300 episodes with the standard tabular Q-learning algorithm and in about 150 episodes with a deep Q-learning algorithm utilizing Prioritized Experience Replay (PER). Considering all of that, I found my results to be satisfactory and I was able to confirm my first hypothesis.

In Section 4.3, I managed to confirm my second hypothesis. There I implemented a deep Q-learning algorithm (Algorithm 2) on a self-made Snake game. I then began by optimizing every parameter but the state type to compare the different state types in the end. There, one was able to see the significance of the state representation and the sometimes drastic differences in performance that they caused. Finally, I managed to create an agent, which outperformed a test group of humans by a significant margin. The agent's (ID: 108.1) average score was 27.1, which translates to filling 35% of the grid. Its impressive high score was 59, which is equivalent to filling 74% of the grid. For comparison, I found a paper [15], in which the authors implemented a similar deep Q-learning algorithm. Their agent managed to achieve an average score of 9.04, which in their Snake game translates to filling 8% of the grid. Its high score was 17, which again translates to filling 14% of the grid. The most significant difference in their implementation, compared to mine, was that they took the raw pixels (240x240) as the state representation and combined that with convolutional layers in their neural network. To sum up I was once again very please with my results.

6 Reflection

The journey this work took me on was genuinely extraordinary. As I had just embarked upon this journey, I could have never imagined all the things I would learn. I started with no knowledge in the field of reinforcement learning nor in deep learning. Also, I had only just discovered programming for myself and had scarcely any amount of experience with Python. I had just finished my first small project, where I programmed the Snake game, which, in a way, inspired me to create an agent who could beat me in it. However, I frankly had no idea whether it was doable or not. I only had a goal but no clear path to it. Nevertheless, I think that it turned out great. I started small with a simple environment and a tabular Q-learning algorithm and worked my way up to implementing a deep Q-learning algorithm in my self-made Snake game.

On this journey, I did not only improve my Python skills by a significant margin. I also got to know LaTeX, which amazed me in many ways as a text writing tool. However, besides all the fantastic software and Python libraries I discovered, I am also glad that I took my time to understand the theory behind it all. Where the book by Richard Sutton and Andrew Barto [10] as well as the cost-free online available course by David Silver [9] played an immense and crucial role.

Finally, I would like to extend my sincere thanks to my supervisor Urs Sieber for his professional advice. I especially appreciate that I was able to work on my project independently.

Declaration of Original Work

I confirm with my signature that this paper is entirely my own work and that any assistance given by others was restricted to advice and proof-reading. All sources employed in preparation of the paper and all quotations used are clearly cited and due acknowledgment is given for all help provided by others. I am aware of the definition of plagiarism in the Matura paper guidelines and that submission of work which has been plagiarised is a serious breach of the Matura regulations (Art. 1quarter of the Maturitätsprüfungsreglements des Gymnasiums).

Date and signature: ....

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[2] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[3] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] R. Koppula. Exploration vs. exploitation in reinforcement learning. `https://www.manifold.ai/exploration-vs-exploitation-in-reinforcement-learning`. [Online; accessed 5-November-2020].

[5] S. Kumar. Balancing a cartpole system with reinforcement learning–a tutorial. *arXiv preprint arXiv:2006.04938*, 2020.

[6] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[7] Melissa and D. Store. Shape sorting cube. `https://www.amazon.co.uk/Melissa-Doug-Shape-Sorting-Cube/dp/B0731PF3TR`, 2020. [Online; accessed 09-September-2020].

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[9] D. Silver. Introduction to rl with david silver. `https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZBiG_XpjnPrSNw-1XQaM_gB&ab_channel=DeepMind`, 2015. [Online; accessed 12-September-2020].

[10] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[11] C. Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[12] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[13] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[14] C. J. C. H. Watkins. Learning from delayed rewards. 1989.

[15] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou. Autonomous agents in snake game via deep reinforcement learning. In *2018 IEEE International Conference on Agents (ICA)*, pages 20–25. IEEE, 2018.

[16] Wikipedia contributors. Markov decision process — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=975754122`, 2020. [Online; accessed 12-September-2020].

[17] Wikipedia contributors. Reinforcement learning — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=976038269`, 2020. [Online; accessed 12-September-2020].

[18] Wikipedia contributors. Snake (video game genre) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Snake_(video_game_genre)`, 2020. [Online; accessed 14-December-2020].

List of Figures

List of Tables

List of Codes

A Source Code: Gridworld Environment

```python
##############################################################
# This code contains my self-made gridworld environment
##############################################################


import numpy as np
import drawSvg as draw

#inspired by the Gridworld example found in:
#Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An
    introduction. MIT press, 2018.

class Gridworld:
    def __init__(self, m=7, n=14, goal_cords=[3,13], wall_cords_list=[]):
        #initializing grid, player's position, inner walls,
        #and the goal's position

        self.m = m
        self.n = n
        self.n_actions = 4
        self.game_matrix = np.full((m,n),fill_value=-1)
        self.player_pos = [m//2, 0]
        self.player_pos_temp = [m//2, 0]
        self.wall_cords_list = wall_cords_list
        self.goal_cords = goal_cords
        if len(wall_cords_list) == 0:
            for i in range(6):
                wall_cords_list.append([i,6])
                wall_cords_list.append([6-i,10])


    def move_player(self, action):
        #taking in an action and returning the player's theoratical position
        self.player_pos_temp = self.player_pos.copy()
        if action == 0:
            self.player_pos_temp[0] -= 1
        elif action == 1:
            self.player_pos_temp[1] += 1
        elif action == 2:
            self.player_pos_temp[0] += 1
        elif action == 3:
```

```
            self.player_pos_temp[1] -= 1
        else:
            print('invalid action')

    def step(self, action):
        #this function adjusts the environment based on the agent's action
        #and returns: his new position, reward and a boolean to indicate
        #whether the episode is done or not

        self.move_player(action)

        #checking whether the agent stepped into an inner wall
        for cords in self.wall_cords_list:
            if cords == self.player_pos_temp:
                reward = -100
                done = True
                self.player_pos = self.player_pos_temp
                return self.player_pos, reward, done

        #checking whether the agent tried to leave the grid
        if (self.player_pos_temp[0] >= self.m or
            self.player_pos_temp[0] < 0 or
            self.player_pos_temp[1] >= self.n or
            self.player_pos_temp[1] < 0):
            reward = -10
            done = False
            return self.player_pos, reward, done

        #checking whether the agent reached the goal
        elif self.player_pos_temp == self.goal_cords:
            reward = -1
            done = True
            self.player_pos = self.player_pos_temp
            return self.player_pos, reward, done

        #if none of the above check then it was a "standard" action
        else:
            reward = -1
            done = False
            self.player_pos = self.player_pos_temp
            return self.player_pos, reward, done

    def reset(self):
```

```python
        #reseting the agent's position (the environment)
        self.player_pos = [self.m//2, 0]
        self.player_pos_temp = [self.m//2, 0]
        return self.player_pos

    def print_matrix(self):
        #for developing purposes
        temp = np.full((self.m,self.n), fill_value=-1)
        temp[tuple(self.goal_cords)] = 0
        for cords in self.wall_cords_list:
            temp[tuple(cords)] = -100
        temp[tuple(self.player_pos)] = 1
        print(temp)

    def print_matrix_symbols(self):
        #to quickly check how it is looking and where the agent is located
        #9 = goal, -9 = inner wall, 1 = agent
        temp = np.zeros((self.m,self.n), dtype=int)
        temp[tuple(self.goal_cords)] =  9
        for cords in self.wall_cords_list:
            temp[tuple(cords)] = -9
        temp[tuple(self.player_pos)] = 1
        print(temp)

    def draw(self, path, save=True):
        #saves a .svg file of the current environment state

        d = draw.Drawing(142,72, origin=(-1,-1)) #creating canvas

        #drawing inner grid
        for i in range(1,self.m):
            d.append(draw.Line(0,10*i,140,10*i,
                               stroke='black', stroke_width=0.5))
        for i in range(1,self.n):
            d.append(draw.Line(10*i,0,10*i,70,
                               stroke='black', stroke_width=0.5))

        #drawing outer walls
        d.append(draw.Line(0,0,140,0, stroke='black', stroke_width=1))
        d.append(draw.Line(0,10*self.m,140,10*self.m,
                               stroke='black', stroke_width=1))

        d.append(draw.Line(0,0,0,70, stroke='black', stroke_width=1))
```

```
        d.append(draw.Line(10*self.n,0,10*self.n,70,
                                    stroke='black', stroke_width=1))

        #drawing inner walls
        for cords in self.wall_cords_list:
            d.append(draw.Line(10*(cords[1]),10*(self.m-1-cords[0]),
                                        10*(cords[1]+1),10*(self.m-1-cords[0]+1),
                                                                stroke='black'))
            d.append(draw.Line(10*(cords[1]),10*(self.m-1-cords[0]+1),
                                        10*(cords[1]+1),10*(self.m-1-cords[0]),
                                                                stroke='black'))

        #drawing player and goal
        d.append(draw.Circle(self.player_pos[1]*10+5,
                                    (self.m-1-self.player_pos[0])*10+5,3.5,
                                    stroke='blue', fill='white', stroke_width=1))
        d.append(draw.Rectangle(self.goal_cords[1]*10+.75,
                                    (self.m-1-self.goal_cords[0])*10+.75, 8.25, 8.5,
                                    stroke='green', fill='green'))
        if save:
            d.saveSvg(path)
        else:
            return d

    def draw_greedy_policy(self, q_table, path):
        d = self.draw('', save=False)
        arrow = draw.Marker(-0.1, -0.5, 0.9, 0.5, scale=4, orient='auto')
        arrow.append(draw.Lines(-0.1, -0.5, -0.1, 0.5, 0.9, 0,
                                                fill='red', close=True))


        states = [(i,j) for i in range(self.m) for j in range(self.n)]

        for s in states:
            m = np.argmax(q_table[s])
            optimal_actions = []

            for i,v in enumerate(q_table[s]):
                if v == q_table[s][m]:
                    optimal_actions.append(i)

            if s == (3,0):
                print(optimal_actions)
```

```python

            for o_a in optimal_actions:
                if o_a == 0:
                    d.append(draw.Line(10*(s[1])+5,10*(self.m-1-s[0])+5,
                                    10*(s[1])+5, 10*(self.m-1-s[0])+7,
                                    stroke='black', stroke_width=0.5,
                                                    marker_end=arrow))
                elif o_a == 1:
                    d.append(draw.Line(10*(s[1])+5,10*(self.m-1-s[0])+5,
                                    10*(s[1])+7, 10*(self.m-1-s[0])+5,
                                    stroke='black', stroke_width=0.5,
                                                    marker_end=arrow))
                elif o_a == 2:
                    d.append(draw.Line(10*(s[1])+5,10*(self.m-1-s[0])+5,
                                    10*(s[1])+5, 10*(self.m-1-s[0])+3,
                                    stroke='black', stroke_width=0.5,
                                                    marker_end=arrow))
                elif o_a == 3:
                    d.append(draw.Line(10*(s[1])+5,10*(self.m-1-s[0])+5,
                                    10*(s[1])+3, 10*(self.m-1-s[0])+5,
                                    stroke='black', stroke_width=0.5,
                                                    marker_end=arrow))
        d.saveSvg(path)




if __name__ == '__main__':

    ###
    #demonstration of how to use the draw_greedy_policy function
    ###

    #initialize environment
    env = Gridworld()

    #load a q-table
    q_table = np.load('q_table.npy', allow_pickle=True)

    #pass the q-table to the function
    #and name the file
    env.draw_greedy_policy(q_table, 'gridworld_q0=-90.svg')
```

Code 1: Source Code for the Gridworld Environment

B Source Code: Gridworld Solver

```
##############################################################
# This code implements the tabular Q-learning algorithm in a
# self-made gridworl environment (described in Section 3.2)
##############################################################


import numpy as np
import matplotlib.pyplot as plt
from gridworld_env import Gridworld
import math

#partly inspired by:
#https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/

class GridworldSolver:
    def __init__(self, q0=0, gamma=1, epsilon=0.1, alpha=0.1):
        #initializing the gridworld, the parameters and other
        #later useful variables
        self.env = Gridworld()

        self.n_actions = self.env.n_actions
        self.q0 = q0

        self.gamma = gamma
        self.epsilon = epsilon
        self.alpha = alpha

        self.optimal_s0 = [-27, -25, -25, -35]

    def init_q_table(self):
        #initializing q-table with user-inputed value
        self.q_table = np.full(((self.env.m, self.env.n, self.n_actions,)),
                                            fill_value=self.q0, dtype=float)

        #setting q-value of terminal states to 0
        self.q_table[tuple(self.env.goal_cords)] = [0 for _ in range(self.n_actions)]
        for cords in self.env.wall_cords_list:
            self.q_table[tuple(cords)] = [0 for _ in range(self.n_actions)]

```

```python

    def update_q_value(self, state, new_state, action, reward, done, alpha):
        #function to update the q-table based on observations
        self.q_table[state][action] += (alpha * (reward + self.gamma *
                                        np.max(self.q_table[new_state]) -
                                        self.q_table[state][action]))

    def one_episode(self, epsilon, alpha):
        #reseting the environment
        done = False
        state = tuple(self.env.reset())
        score = 0
        while not done:
            #choosing an action based on e-greedy action selection
            action = np.argmax(self.q_table[state])
            if np.random.uniform() < epsilon:
                action = np.random.randint(0,self.n_actions)



            #taking the selected action and observing its consequences
            new_state, reward, done = self.env.step(action)
            new_state = tuple(new_state)

            #keeping track of the reward
            score += reward

            #updating the q-table
            self.update_q_value(state, new_state, action, reward, done, alpha)

            state = new_state

        return score

    def n_episodes(self, n):
        #initializing q-table, parameters, and score_history
        #to keep track of the agent's performance
        self.init_q_table()

        score_history = []
        solved_after = -1
        epsilon = self.epsilon
        alpha = self.alpha
```

```

        #playing n episodes and keeping track of the agent's performance
        for e in range(n):

            score = self.one_episode(epsilon, alpha)
            score_history.append(score)

            ###
            #uncomment the following, if one would like to save the q-table
            #at the point were it was "optimal" for the first time
            #"optimal" as described in Section 3.2
            ###

            #if (list(self.q_table[(3,0)]) == self.optimal_s0
            #                        and solved_after == -1):
            #    np.save('q_table.npy', self.q_table)
            #    solved_after = e

        return score_history, solved_after

    def N_n_episodes(self, N, n):
        #this is to test different sets of parameters,
        #it takes the average of multiple runs
        s = []
        solved_average = []

        for _ in range(N):
            l = self.n_episodes(n)
            s.append(l[0])
            solved_average.append(l[1])

        return list(np.average(s, axis=0)), np.average(solved_average)

    def plot_scores(self, score_histories, labels, solved_after, title='plot')
    :
        #plots the agent's score of the individual episodes
        #the episode on which he solved

        plt.rcParams.update({'font.size': 15})
        for i,score_history in enumerate(score_histories):
            plt.plot(score_history, label=f'{labels[0]}: {labels[1][i]}',
                    color=f'C{i}')
            if solved_after[i] != -1:
```

```
                plt.plot([solved_after[i],solved_after[i]],[-50,0],
                        color=f'C{i}', linestyle='--', linewidth=3,
                        label=f'converged after {solved_after[i]} episodes')

        plt.plot([-25 for _ in range(len(score_histories[0]))], color='k',
                                                label='optimal return, -25')
        plt.xlabel('episode')
        plt.ylabel('score')
        plt.legend(loc='lower right', fontsize='x-large')
        plt.title(title)
        plt.show()


    def __str__(self):
        #to check the currently used parameters
        return (f'epsilon={self.epsilon}, alpha={self.alpha}, '+
                f'q0={self.q0}, gamma={self.gamma}')
```

Code 2: Source Code for the Gridworld Solver

C Source Code: Gridworld Main Script

```python
###############################################################
# This code is a demonstration how to use the GridworldSolver
# (utilizing multiprocessing to speed it up)
# To execute, please place gridworld_env.py, gridworld_solver.py
# and gridworld_solver.py in the same directory
###############################################################


import gridworld_solver
import multiprocessing as mp
import time
import numpy as np

#here one can change the parameters other than the one being adjusted
#every parameter other than the one being adjusted must have a default value
def one_process(epsilon, q0=0, gamma=1, alpha=.5, N=100, n=20000):
    solver = gridworld_solver.GridworldSolver(q0, gamma, epsilon, alpha)
    return solver.N_n_episodes(N,n)

#this is needed in order for the multiprocessing to work
if __name__ == '__main__':

    #keeping track of important metrics
    s0 = time.time()
    score_histories = []
    solved_after = []

    #at index 0 is the parameter that is being adjusted as str
    #at index 1 is a list of different values to be tested for said parameter
    parameters = ['epsilon', [0.1,0.3,0.5,0.7]]

    #using multiprocessing to
    p = mp.Pool(mp.cpu_count())
    l = p.map(one_process,parameters[1])
    for game in l:
        score_histories.append(game[0])
        solved_after.append(game[1])

    #saving the data
    np.save(f'data_{parameters[0]}.npy', np.array([score_histories,
                                    solved_after, parameters], dtype=object))
```

```python

    plotter = gridworld_solver.GridworldSolver()
    #the tile has to be adjusted manually
    plotter.plot_scores(score_histories, parameters, solved_after,
                        title='epsilon=_, alpha=0.5, q0=0, gamma=1')

    #to see how long it took
    print(f'calculations took {round((time.time()-s0)/60)} minutes')

###
#if one does not want to compare different values for a parameter
#and would simply like to try one specific set of parameters
#here is how to do it
###

#set parameters and create solver object
solver = gridworld_solver.GridworldSolver(epsilon=0.3,alpha=0.9, q0=-90)
#play a specific amount of episodes
s, solved = solver.n_episodes(20000)
#save the q-table (optional)
np.save('q_table.npy', solver.q_table)
```

Code 3: Source Code for the Gridworld Main Script

D Source Code: Cartpole Solver

```
##############################################################
# This code implements the tabular Q-learning algorithm in the
# Cartpole environment provided by OpenAI-Gym
##############################################################

#partly inspired by:
#https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/


import numpy as np
import matplotlib.pyplot as plt
import gym
import math
import multiprocessing as mp
import sys

class CartpoleSolver:
    def __init__(self, obs_high=[2.4, 3.5, 0.21, 1.2],
                obs_low=[-2.4, -3.5, -0.21, -1.2],
                obs_chunks=[10, 10, 10, 10],
                q0=0, gamma=1, epsilon=1, alpha=1, epsilon_min=0.1,
                alpha_min=0.1, decrease_methode=[0,0], S_T_reward=-1000):

        #initializing variables
        self.n_actions = 2
        self.obs_high = np.array(obs_high)
        self.obs_low = np.array(obs_low)
        self.obs_chunks = np.array(obs_chunks)
        self.obs_chunks_delta = (self.obs_high - self.obs_low) / obs_chunks
        self.q0 = q0
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_min = epsilon_min
        self.alpha = alpha
        self.alpha_min = alpha_min
        self.state_list = []
        self.decrease_methode = decrease_methode
        self.S_T_reward = S_T_reward

        #initializing environment
```

```
        self.env = gym.make('CartPole-v0')

    def discretize_state(self, state):
        #turning a continuous state into a descrete one
        discrete_state = list(((state - self.obs_low) /
                            self.obs_chunks_delta).astype(np.int))
        discrete_state = [min(self.obs_chunks[i]-1,
                        discrete_state[i]) for i in range(4)]
        return tuple(discrete_state)

    def update_q_value(self, q_table, state, new_state,
                        action, reward, done, alpha):
        #updating the Q-table
        if done:
            #if it is done then the q_table[next_state]
            #values should all be zero, hence that part is missing
            q_table[state][action] += (alpha *
                                        (reward - q_table[state][action]))
        else:
            q_table[state][action] += (alpha *
                                        (reward +
                                        self.gamma*np.max(q_table[new_state]) -
                                        q_table[state][action]))
        return q_table

    def decrease(self, v, v_min, episode, dm):
        #function to decrease a parameter over time
        #0-3 are different decreasing methods to choose from
        if dm == 0:
            return np.maximum(v_min,
                        np.minimum(1., 1.0 - math.log10((episode + 1) / 25)))
        elif dm == 1:
            return np.maximum(v_min, v*(0.99)**episode)
        elif dm == 2:
            return np.maximum(v_min, 1.0 / np.sqrt(episode+1))
        elif dm == 3:
            return 1/(episode+1)

    def one_episode(self, epsilon, alpha, render=False):
        #reseting the environment
        done = False
        state = self.discretize_state(self.env.reset())
```

```
        score = 0
        while not done:
            if render:
                self.env.render()

            #choosing an action based on e-greedy action selection
            action = np.argmax(self.q_table[state])
            if np.random.uniform() < epsilon:
                action = self.env.action_space.sample()

            #taking the selected action and observing its consequences
            new_state, reward, done, _ = self.env.step(action)
            self.state_list.append(new_state)
            new_state = self.discretize_state(new_state)

            #keeping track of the reward
            score += reward

            #adding extra punishment if the agent failed
            if done and score < 200:
                reward = -1000

            #updating the q-table
            self.update_q_value(self.q_table, state, new_state,
                                action, reward, done, alpha)

            state = new_state

        return score

    def n_episodes(self, n, render=False):
        #initializing q-table, parameters, and score_history
        #to keep track of the agent's performance
        self.q_table = np.full((tuple(self.obs_chunks)+(self.n_actions,)),
                                fill_value=self.q0, dtype=float)
        score_history = []
        epsilon = self.epsilon
        alpha = self.alpha

        #playing n episodes and keeping track of the agent's performance
        for e in range(n):

            score = self.one_episode(epsilon, alpha, render=render)
```

```python
            score_history.append(score)

            #this would stop the agent and save his q_table
            #as soon as he reaches the 195 threshold
            #if e > 100:
            #    if np.average(score_history[-100:]) >= 195:
            #        np.save('q_table.npy', self.q_table)
            #        print(e)
            #        sys.exit()


            epsilon = self.decrease(epsilon, self.epsilon_min,
                                    e, self.decrease_methode[0])
            alpha = self.decrease(alpha, self.alpha_min,
                                    e, self.decrease_methode[1])


        return score_history

    def N_n_episodes(self, N, n, render=False):
        #this is to test different sets of parameters,
        #it takes the average of multiple runs

        s = []

        for _ in range(N):
            s.append(self.n_episodes(n))

        return list(np.average(s, axis=0))

    def plot_runing_average(self, score_histories, labels, title='plot'):
        #plots the average score from the last 100 episodes at each episodes
        for score_history,label in zip(score_histories,labels[1]):
            y = [np.average(score_history[max(0,i-100):i])
                    for i in range(len(score_history))]
            plt.plot(y, label=f'{labels[0]}: {label}',
                color=f'C{score_histories.index(score_history)}')
            plt.plot(y)

        plt.plot([195 for _ in range(len(score_histories[0]))], color='k',
                    label='195 threshold', linestyle=':')

        plt.legend(loc='lower right', fontsize='x-large')
```

```
        plt.xlabel('episode')
        plt.ylabel('average return over last 100 episodes')
        plt.title(title)
        plt.show()

    def plot_scores(self, score_histories, labels, title='plot'):
        #plot the score of each episode
        for score_history, label in zip(score_histories, labels[1]):
            plt.plot(score_history, label=f'{labels[0]}: {label}',
                     color=f'C{score_histories.index(score_history)}')
        plt.legend()
        plt.title(title)
        plt.show()

    def __str__(self):
        #to check the currently used parameters
        return (f'obs_high={self.obs_high}, obs_low={self.obs_low}, ' +
                f'obs_chunks={self.obs_chunks}, q0={self.q0}, ' +
                f'gamma={self.gamma}, epsilon={self.epsilon}, ' +
                f'alpha={self.alpha}, epsilon_min={self.epsilon_min}, ' +
                f'alpha_min={self.alpha_min}, S_T_reward={self.S_T_reward}, '+
                f'decrease_methode={self.decrease_methode}')
```

Code 4: Source Code for the Cartpole Solver

E Source Code: Cartpole Main Script

```python
##############################################################
# This code is a demonstration how to use the CartpoleSolver
# To execute, please place cartpole_solver.py and
# cartpole_main.py in the same directory
##############################################################


import cartpole_solver
import numpy as np

#to keep track of the scores
score_histories = []

#at index 0 is the parameter that is being adjusted as str
#at index 1 is a list of different values to be tested for said parameter
parameters_to_be_tried = ['alpha',[0.1,0.1,0.1,0.1,0.1]]

#iterates through the values to be tried
for p in parameters_to_be_tried[1]:
    solver = cartpole_solver.CartpoleSolver(
                        obs_high=[2.4, 2, 0.21, 1.8],
                        obs_low=[-2.4, -2, -0.21, -1.8],
                        obs_chunks=[1, 1, 12, 12], q0=0,
                        gamma=1, epsilon=1, alpha=p,
                        epsilon_min=0.01, alpha_min=p,
                        decrease_methode=[3,0], S_T_reward=-2000)
    #(amount of agents, amount of episodes) to be tested on a value
    l = solver.N_n_episodes(1,500)
    score_histories.append(l)
    print(solver.__str__())

#setting the title for the plot
title = f'adjusting {parameters_to_be_tried[0]}'

#saving the data (optional)
#so it can be laoded later to plot it again
np.save(f'data_{parameters_to_be_tried[0]}.npy',
        np.array([score_histories, parameters_to_be_tried, title],
                                                dtype=object))

#plotting the results
```

```python
solver.plot_runing_average(score_histories,
                            parameters_to_be_tried, title=title)

###
#example of loading data and plotting
###

#initializing the CartpoleSolver object containing plotter function
plotter = cartpole_solver.CartpoleSolver()

#laoding the data
data = np.load('data_[name].npy', allow_pickle=True)

#setting the title and plotting the data
title = f'adjusting {data[1][0]}'
plotter.plot_runing_average(data[0],data[1],title)


###
#alternatively one can test a single specific set of parameters
###

#initializing solver with specific parameters
solver = cartpole_solver.CartpoleSolver(
                obs_high=[2.4, 2, 0.21, 1.8],
                obs_low=[-2.4, -2, -0.21, -1.8],
                obs_chunks=[1, 1, 12, 12], q0=0,
                gamma=1, epsilon=1, alpha=0.9,
                epsilon_min=0.01, alpha_min=0.9,
                decrease_methode=[3,0], S_T_reward=-2000)

#either by a single agent (n = amounts of episodes)
score_history = solver.n_episodes(n=1000,render=False)

#or by many agents (N = amount of agents)
score_histories = solver.N_n_episodes(N=10,n=1000,render=False)
```
Code 5: Source Code for the Cartpole Main Script

F Source Code: Snake Trainer

```python
##############################################################
# This code comibines the Snake environment and the DQL agent
# It combines them in a python object, which can be used to
# train an agent with freely adjustable parameters
##############################################################


import snake_agent
import snake_env
import os
import torch
import time
import sys
import numpy as np

class SnakeTrainer:
    def __init__(self, agent_id, updates_per_step, gamma, lr,
                 epsilon, epsilon_decay, epsilon_min, statetype,
                 nr_hiddenlayers, nr_neurons, max_mem_length,
                 gridlength, r_fruit, r_collision, r_step,
                 lr_decay=False, lr_milestones=None, lr_gamma=None,
                 render_bool=False):

        #selecting device, preferably the gpu
        self.device = torch.device('cuda' if torch.cuda.is_available()
                                            else 'cpu')

        #paths for data collection
        self.agent_id = agent_id
        self.path = (os.path.dirname(os.path.abspath(__file__))
                     + f'/results/{agent_id}/')

        self.modelpath = self.path + 'checkpoint.tar'
        self.settingspath = self.path + 'settings.txt'
        self.datapath = self.path + 'data.tar'
        self.memorypath = self.path + 'memory.tar'
        self.replaypath = self.path + 'replay.npz'

        #initializing important variables
        self.updates_per_step = updates_per_step
        self.render_bool = render_bool
```

```python

        #initializing the environment
        self.env = snake_env.Environment(gridlength=gridlength,
            snakex=gridlength//2, snakey=gridlength//2,
            fruitspawnset=False, r_fruit = r_fruit,
            r_collision = r_collision, r_step = r_step,
            statetype=statetype)

        self.statetype = statetype
        self.rewards_fcs = [r_fruit,r_collision,r_step]

        obs_space = len(self.env.env_reset())

        #initializing the agent
        self.agent = snake_agent.Agent(gamma=gamma, lr=lr, epsilon=epsilon,
            epsilon_decay=epsilon_decay, epsilon_min=epsilon_min,
            obs_space=obs_space, act_space=4, device=self.device,
            updates_per_step=self.updates_per_step, lr_decay=lr_decay,
            lr_milestones=lr_milestones, lr_gamma=lr_gamma,
            max_mem_length=max_mem_length,
            nr_hiddenlayers=nr_hiddenlayers, nr_neurons=nr_neurons)

        #initializing data collection
        if not os.path.isdir(self.path):
            self.init_data()
        else:
            self.load_agent()


    def init_data(self):
        #creates a directory and a first textfile with agent's specs
        os.makedirs(self.path)
        textfile = open(self.settingspath, 'a')
        textfile.write(f"agent_id: {self.agent_id}"
                    +f"device: {self.device} \nDQN: \n{self.agent.dqn}\n"
                    +f"\noptimizer: \n{self.agent.optimizer}\n"
                    +f"\ngamma: {self.agent.gamma} \nepsilon_decay & _min: "
                    +f"{self.agent.epsilon_decay}, {self.agent.epsilon_min}"
                    +f"\nstatetype: {self.env.statetype} \nupdates_per_step: "
                    +f"{self.agent.updates_per_step} \nmax_mem_length: "
                    +f"{self.agent.max_mem_length} \nrewards(f,c,s): "
                    +f"{self.env.r_fruit}, {self.env.r_collision}, "
                    +f"{self.env.r_step} \nlr_decay: {self.agent.lr_decay}")
```

```python
        if self.agent.lr_decay:
            textfile.write(f"lr_scheduler: \n{self.agent.scheduler}\n"
                    +f"lr_milestones: {self.agent.lr_milestones} \nlr_gamma: "
                    +f"{self.agent.lr_gamma}")
        textfile.close()

        #since its the first time this agent palys
        # new data collection lists are created
        self.episode_nr = 0
        self.epsilon_history = []
        self.stime_history = []
        self.q_history = []
        self.replay_memory = []
        self.replay_memorytemp = []
        self.updates = 0
        self.main_time = 0

    def load_agent(self):
        #laods all that is necessary

        #loading model
        checkpoint = torch.load(self.modelpath)
        self.agent.dqn.load_state_dict(checkpoint['model_state_dict'])
        self.agent.optimizer.load_state_dict(
                                    checkpoint['optimizer_state_dict'])
        if self.agent.lr_decay:
            self.agent.scheduler.load_state_dict(checkpoint['lr_scheduler'])
        print("model was loaded")

        #loading data
        checkpoint = torch.load(self.datapath)
        self.epsilon_history = checkpoint['epsilon_history']
        self.stime_history = checkpoint['sruvival_time_history']
        self.q_history = checkpoint['q_history']
        self.env.scorelist = checkpoint['scorelist']
        self.episode_nr = checkpoint['episode_nr']
        self.agent.epsilon = checkpoint['epsilon']
        self.updates = checkpoint['updates']
        self.main_time = checkpoint['main_time']
        print("data was loaded")

        #loading agent's memory
        self.agent.memory = torch.load(self.memorypath)
```

```
        print("memory was loaded")

        #loading replay memory
        self.replay_memorytemp = []
        if os.path.exists(self.replaypath):
            self.replay_memory = np.load(self.replaypath,
                                allow_pickle=True)['replay'].tolist()
        print("replay memory was loaded")

    def save_model(self):
        #saves the model
        if self.agent.lr_decay:
            torch.save({
                'model_state_dict': self.agent.dqn.state_dict(),
                'optimizer_state_dict': self.agent.optimizer.state_dict(),
                'lr_scheduler': self.agent.scheduler.state_dict(),
                }, self.modelpath)
        else:
            torch.save({
                'model_state_dict': self.agent.dqn.state_dict(),
                'optimizer_state_dict': self.agent.optimizer.state_dict()
                }, self.modelpath)

    def save_data(self):
        #saves the data
        torch.save({
            'episode_nr': self.episode_nr,
            'epsilon': self.agent.epsilon,
            'epsilon_history': self.epsilon_history,
            'sruvival_time_history': self.stime_history,
            'q_history': self.q_history,
            'scorelist': self.env.scorelist,
            'updates': self.updates,
            'main_time': self.main_time
            }, self.datapath)

    def save_memory(self):
        #save the memory
        torch.save(self.agent.memory,self.memorypath)

    def save_replay(self):
        np.savez_compressed(self.replaypath,
                        replay=np.array(self.replay_memory, dtype=object))
```

```python


    def train_n_episodes(self, n, gridlength, epsilon_reset=False):
        #loading new environment incase the gridlength changed
        self.env = snake_env.Environment(gridlength=gridlength,
            snakex=gridlength//2, snakey=gridlength//2,
            fruitspawnset=False, r_fruit = self.rewards_fcs[0],
            r_collision = self.rewards_fcs[1], r_step = self.rewards_fcs[2],
            statetype=self.statetype)

        if os.path.exists(self.datapath):
         self.load_agent()

        if epsilon_reset:
            self.agent.epsilon = 1

        #starting the main for loop to play n episodes
        for e in range(n):
            self.episode_nr += 1
            self.env.env_reset()
            current_body = []
            for b in self.env.body.list:
                current_body.append(b)
            self.replay_memorytemp.append((self.env.snake.x,
                    self.env.snake.y, self.env.fruit.x,
                    self.env.fruit.y, current_body))

            main_time_temp = time.time()

            for i in range(1500):
                #receiving the observation from the environment
                #choosing an action and take it
                #receiving reward
                state = self.env.get_state()
                state = state.to(self.device)
                action = self.agent.act(state)
                reward = self.env.env_step(action)

                #keeping track of the estimated total reward at s_0
                if i == 0:
                    self.q_history.append(torch.max(
                                    self.agent.dqn.forward(state)).item())

```

```
                if self.render_bool:
                    self.env.render(60, (0,0,0))

                #saving the frame for later replay
                current_body = []
                for b in self.env.body.list:
                    current_body.append(b)
                self.replay_memorytemp.append((self.env.snake.x,
                        self.env.snake.y, self.env.fruit.x,
                            self.env.fruit.y, current_body))


                #checking if he collided or fille the whole grid
                if reward == self.rewards_fcs[1]:
                    done = True
                elif reward == 100:
                    #incase the agent ever manages to fill the whole grid
                    done = True
                else:
                    done = False

                #receiving new state
                new_state = self.env.get_state()
                new_state = new_state.to(self.device)

                #saving the agent's experience
                self.agent.savedata(state, action,
                                    reward, new_state, done)

                #as soon as there is enough data the training will start
                if len(self.agent.memory) > self.updates_per_step:
                    self.agent.train()
                    self.updates += self.updates_per_step

                #if he collided with either the wall or himself,
                # the episode is over
                if done:
                    break

            #keeping track of important metrics
            self.epsilon_history.append(self.agent.epsilon)
            self.env.env_store_score()
            self.stime_history.append(i+1)
```

```
            print(f"episode: {self.episode_nr}"
                +f" score: {self.env.scorelist[-1]}"
                +f" surv_time: {i+1}"
                +f" epsilon: {round(self.agent.epsilon,3)}"
                +f" q: {round(self.q_history[-1])}")


            #keeping track of time taken
            self.main_time += time.time()-main_time_temp

            self.replay_memory.append(self.replay_memorytemp)
            self.replay_memorytemp = []

            #every 50th episode everything is saved
            if self.episode_nr%50 == 0:
                print("saving...")
                self.save_model()
                self.save_data()
                self.save_memory()
                self.save_replay()
                print("done saving!")

    def test_n_episodes(self, n, gridlength, record=False, render=True):
        #loading new environment incase the gridlength changed
        #recort = True will save a .png of every frame
        self.env = snake_env.Environment(gridlength=gridlength,
            snakex=gridlength//2, snakey=gridlength//2,
            fruitspawnset=False, r_fruit = self.rewards_fcs[0],
            r_collision = self.rewards_fcs[1], r_step = self.rewards_fcs[2],
            statetype=self.statetype)

        if os.path.exists(self.datapath):
         self.load_agent()

        score_list = []

        import pygame
        clock = pygame.time.Clock()
        for e in range(n):
            self.env.env_reset()
            rewards = []
            for i in range(1500):
                if render:
```

```
                clock.tick(10)
            #receiving the observation from the environment
            #choosing an action and take it
            #receiving reward
            state = self.env.get_state()
            state = state.to(self.device)
            action = self.agent.act_greedy(state)
            reward = self.env.env_step(action)
            rewards.append(reward)

            #keeping track of the estimated total reward at s_0
            if i == 0:
                q0 = torch.max(self.agent.dqn.forward(state)).item()

            if render:
                self.env.render(60, (0,0,0), record=record,
                path=self.path + f"frame{i}.png")


            #checking if he collided or fille the whole grid
            if reward == self.rewards_fcs[1]:
                done = True
            elif reward == 100:
                done = True
            else:
                done = False

            #receiving new state
            new_state = self.env.get_state()
            new_state = new_state.to(self.device)

            #if he collided with either the wall or himself,
            # the episode is over
            if done:
                break

        real_q0 = 0
        for j in range(len(rewards)):
            real_q0 += rewards[j] * (self.agent.gamma**(j))

        print(f'score: {self.env.score}, surv_time: {i},'
            + f' est q0: {round(q0)}, real q0: {round(real_q0)}')
        score_list.append(self.env.score)
```

```
        return score_list

    def watch_replay(self, episodes, gridlengths):
        #used to watch a replay of a specific training episode
        import pygame
        pygame.init()
        clock = pygame.time.Clock()
        for e,g in zip(episodes,gridlengths):
            print(self.env.scorelist[e])
            for snakex,snakey,fruitx,fruity,bodylist in self.replay_memory[e]:
                clock.tick(10)
                self.env.renderreplay(gridlength=g, size=45,
                                    snakex=snakex, snakey=snakey,
                                    fruitx=fruitx, fruity=fruity,
                                            bodylist=bodylist)
```

Code 6: Source Code for the Snake Trainer

G Source Code: Snake Agent

```
############################################################
# This code contains the Python object for the DQL agent.
# the agent is made up of a DQN and the implemented algorithm
############################################################

#partly inspired by:
#https://github.com/philtabor/Youtube-Code-Repository/blob/master/
    ReinforcementLearning/DeepQLearning/simple_dqn_torch_2020.py


import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import deque
import random
import numpy as np
import time
import os


class DQN(nn.Module):
    def __init__(self, obs_space, act_space, nr_hiddenlayers, nr_neurons):
        super(DQN, self).__init__()
        #modular deep Q-network
        #amount and size of linear hidden layers are freely adjustable
        self.nr_hiddenlayers = nr_hiddenlayers
        nr_neurons.insert(0,obs_space)
        nr_neurons.append(act_space)
        self.nr_neurons = nr_neurons

        temp = []
        for i in range(self.nr_hiddenlayers+1):
            temp.append(nn.Linear(self.nr_neurons[i],self.nr_neurons[i+1]))

        self.layers = nn.ModuleList(temp)


    def forward(self, state):
        #forward pass thorugh the network
        x = state
        for i in range(self.nr_hiddenlayers):
```

```
            x = F.relu(self.layers[i](x))

        return self.layers[-1](x)


class Agent:
    def __init__(self, gamma, lr, epsilon, epsilon_decay, epsilon_min,
                 obs_space, act_space, device, updates_per_step,
                 lr_decay = False, lr_milestones = None, lr_gamma = None,
                 nr_hiddenlayers=1, nr_neurons=[128], max_mem_length=1000):
        #deep q-learning agent
        #initializing variables and constants
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.lr = lr
        self.device = device
        self.act_space = act_space
        self.obs_space = obs_space
        self.updates_per_step = updates_per_step
        self.lr_decay = lr_decay
        self.lr_milestones = lr_milestones
        self.lr_gamma = lr_gamma
        self.max_mem_length = max_mem_length

        #initializing agent's memory
        self.memory = deque(maxlen=self.max_mem_length)

        #initializing agent's deep-Q-network
        self.dqn = DQN(self.obs_space, self.act_space,
                        nr_hiddenlayers, nr_neurons).to(self.device)
        self.optimizer = torch.optim.Adam(self.dqn.parameters(), lr=lr)

        #initializing learning rate scheduler, if lr decay is wanted
        if self.lr_decay:
            self.scheduler = torch.optim.lr_scheduler.MultiStepLR(
                                            self.optimizer,
                                            milestones=self.lr_milestones,
                                            gamma=self.lr_gamma)


    def savedata(self, state, action, reward,
```

```
                                new_state, gameovertemp):
        #used to append a transition to the agent's memory
        self.memory.append((state, action,
                            reward, new_state, gameovertemp))

    def act(self, state):
        #choosing action with epsilon-greedy strategy
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.act_space)
        else:
            qvalues = self.dqn.forward(state)
            return torch.argmax(qvalues).item()

    def act_greedy(self, state):
        #choosing action completely greedy
        qvalues = self.dqn.forward(state)
        return torch.argmax(qvalues).item()

    def train(self):
        #sampling a specific number of datapoints from memory
        trainsample = random.sample(self.memory, self.updates_per_step)

        #iterates through every datapoint (transition) and
        # performs a weight update on the Q-network
        for state, action, reward, new_state, done in trainsample:
            if done:
                target = reward
            else:
                target = reward + (self.gamma
                        * torch.max(self.dqn.forward(new_state)).item())

            prediction = self.dqn.forward(state)[action].to(self.device)
            target = torch.tensor(target).to(self.device)
            self.optimizer.zero_grad()
            loss = F.mse_loss(target, prediction)
            loss.backward()
            self.optimizer.step()
            if self.lr_decay:
                self.scheduler.step()

        #adjustment of the exploration rate epsilon
        if self.epsilon > self.epsilon_min:
            self.epsilon = self.epsilon * self.epsilon_decay
```

```
        else:
            self.epsilon = self.epsilon_min

    def get_lr(self):
        #for development used to check current learning rate
        for param_group in self.optimizer.param_groups:
            return param_group['lr']
```

Code 7: Source Code for the Snake Agent

H Source Code: Snake Environment

```python
##############################################################
# This code contains my self-made Snake environment
# adjusted to be used as a Markov decision process
##############################################################


import random
import numpy as np
import torch
import drawSvg as draw

class Snake:
    def __init__(self, spawnx, spawny):
        #only considers the head and its movement
        self.x = spawnx
        self.y = spawny
        self.changelist = []


    def act(self, action):
        #takes action and keep a record of it in the changelist
        # for delayed body movement
        if action == 0:
            self.x += 1
            self.changelist.append((1, 0))
        if action == 1:
            self.x -= 1
            self.changelist.append((-1, 0))
        if action == 2:
            self.y += 1
            self.changelist.append((0, 1))
        if action == 3:
            self.y -= 1
            self.changelist.append((0, -1))

class Body:
    def __init__(self, len):
        #keeps track of the snake's body
        self.len = len
        self.list = []

```

```
    def grow(self, snake):
        #function to grow the body
        self.len += 1
        self.list.append([snake.x,snake.y])

    def update(self, changelist, snake):
        #function to position each body part at its position
        #uses the snake's head position and the changelist
        # to calculate the position
        for i in range(0,self.len):
            x = snake.x
            y = snake.y
            for n in range(1,(i+2)):
                ch = changelist[-n]
                x = x-ch[0]
                y = y-ch[1]
            self.list[i] = ([x,y])

class Fruit:
    #used for the fruit
    def __init__(self, gridlength, spawnset = False,
                            spawnx = None , spawny = None):
        #spawns the fruit at a either given or random position
        if spawnset:
            self.x = spawnx
            self.y = spawny
        else:
            self.x = random.randint(0,(gridlength-1))
            self.y = random.randint(0,(gridlength-1))

class Environment:
    def __init__(self, gridlength, snakex, snakey, r_fruit,
                r_collision, r_step, fruitspawnset,
                fruitx = None, fruity = None, statetype = 1):

        #initiliazing variables, snake, fruit and body
        self.gridlength = gridlength
        self.snakex = snakex
        self.snakey = snakey
        self.snakecolor = '#197416'
        self.fruitcolor = '#E92323'
        self.bodycolor = '#32B145'
        self.background = '#485B57'
```

```
        self.fruitx = fruitx
        self.fruity = fruity
        self.fruitspawnset = fruitspawnset
        self.snake = Snake(self.snakex, self.snakey)
        self.fruit = Fruit(self.gridlength,
                           spawnset=self.fruitspawnset,
                           spawnx=self.fruitx, spawny=self.fruity)
        self.body = Body(0)
        self.score = 0
        self.scorelist = []
        self.gamewon = False
        self.statetype = statetype

        #adjustable rewards given to the agent
        self.r_fruit = r_fruit
        self.r_collision = r_collision
        self.r_step = r_step

    def collisionfruit(self):
        #checks whether the snake has collided with the fruit
        if self.snake.x == self.fruit.x and self.snake.y == self.fruit.y:
            return True

    def collisionborder(self):
        #checks whether the snake has collided with the border
        if (self.snake.x == self.gridlength or self.snake.x == -1 or
            self.snake.y == self.gridlength or self.snake.y == -1):
            return True

    def collisionself(self):
        #checks whether the snake has collided with its body
        for b in self.body.list:
            if self.snake.x == b[0] and self.snake.y == b[1]:
                return True

    def check_fruit_spawn(self):
        #makes sure the fruit can only spawn on a free cell
        self.fruit = Fruit(self.gridlength)
        for b in self.body.list:
            if self.fruit.x == b[0] and self.fruit.y == b[1]:
                self.check_fruit_spawn()
            else:
                pass
```

```
        if self.fruit.x == self.snake.x and self.fruit.y == self.snake.y:
            self.check_fruit_spawn()
        else:
            pass

    def get_state(self):
        #this function is needed,
        # because there are different state types to choose from
        if self.statetype == 1:
            return self.get_state1()
        elif self.statetype == 2:
            return self.get_state2()
        elif self.statetype == 3:
            return self.get_state3()
        elif self.statetype == 4:
            return self.get_state4()
        elif self.statetype == 5:
            return self.get_state5()
        elif self.statetype == 6:
            return self.get_state6()
        elif self.statetype == 7:
            return self.get_state7()
        elif self.statetype == 8:
            return self.get_state8()

    def get_state1(self):
        #returns a 24-dimensional vector,
        #consisting of the distances between the snake's head and:
        #the border
        #+ binary vision for the fruit and its body
        #for each object 8 different directions get checked:
        #if there is no object in that direction the distance is 0
        #border distances to walls
        #and diagonals

        #walls, if it is 1 then one more step in that direction is lethal
        b0 = float(self.gridlength-self.snake.x)
        b2 = float(self.snake.y+1)
        b4 = float(self.snake.x+1)
        b6 = float(self.gridlength-self.snake.y)

        b1 = False
        b3 = False
```

```
        b5 = False
        b7 = False
        #diagonals
        for i in range(self.gridlength):
            if ((self.snake.x+i >= self.gridlength or
                self.snake.y+i >= self.gridlength) and
                (not b7)):
                b7 = float(np.sqrt(i**2+i**2))
            if ((self.snake.x-i < 0 or
                self.snake.y-i < 0) and
                (not b3)):
                b3 = float(np.sqrt(i**2+i**2))
            if ((self.snake.x+i >= self.gridlength or
                self.snake.y-i < 0) and
                (not b1)):
                b1 = float(np.sqrt(i**2+i**2))
            if ((self.snake.x-i < 0 or
                self.snake.y+i >= self.gridlength) and
                (not b5)):
                b5 = float(np.sqrt(i**2+i**2))

        f0 = 0.0
        f1 = 0.0
        f2 = 0.0
        f3 = 0.0
        f4 = 0.0
        f5 = 0.0
        f6 = 0.0
        f7 = 0.0

        s0 = 0.0
        s1 = 0.0
        s2 = 0.0
        s3 = 0.0
        s4 = 0.0
        s5 = 0.0
        s6 = 0.0
        s7 = 0.0

        if self.snake.y == self.fruit.y:
            if self.snake.x < self.fruit.x:
                f0 = 1.0
            else:
```

```
                f4 = 1.0
        if self.snake.x == self.fruit.x:
            if self.snake.y < self.fruit.y:
                f6 = 1.0
            else:
                f2 = 1.0


        for i in range(self.gridlength):
            if (self.snake.x+i == self.fruit.x and
                self.snake.y+i == self.fruit.y):
                f7 = 1.0
            if (self.snake.x-i == self.fruit.x and
                self.snake.y-i == self.fruit.y):
                f3 = 1.0
            if (self.snake.x+i == self.fruit.x and
                self.snake.y-i == self.fruit.y):
                f1 = 1.0
            if (self.snake.x-i == self.fruit.x and
                self.snake.y+i == self.fruit.y):
                f5 = 1.0

        for b in self.body.list:
            if self.snake.y == b[1]:
                if self.snake.x < b[0]:
                    s0 = 1.0
                else:
                    s4 = 1.0
            if self.snake.x == b[0]:
                if self.snake.y < b[1]:
                    s6 = 1.0
                else:
                    s2 = 1.0
            for i in range(self.gridlength):
                if (self.snake.x+i == b[0] and
                    self.snake.y+i == b[1]):
                    s7 = 1.0
                if (self.snake.x-i == b[0] and
                    self.snake.y-i == b[1]):
                    s3 = 1.0
                if (self.snake.x+i == b[0] and
                    self.snake.y-i == b[1]):
                    s1 = 1.0
```

```python
                if (self.snake.x-i == b[0] and
                    self.snake.y+i == b[1]):
                    s5 = 1.0

        return torch.tensor([b0, b1, b2, b3, b4, b5, b6, b7,
                             f0, f1, f2, f3, f4, f5, f6, f7,
                             s0, s1, s2, s3, s4, s5, s6, s7])

    def get_state2(self):
        #four directions for distances to border
        #four directions for binary vision to body
        #relative coordinates to fruit
        #up, down, right and left

        #border
        b0 = float(self.gridlength-self.snake.x)
        b1 = float(self.snake.y+1)
        b2 = float(self.snake.x+1)
        b3 = float(self.gridlength-self.snake.y)

        #own body
        s0 = 0.0
        s1 = 0.0
        s2 = 0.0
        s3 = 0.0

        for b in self.body.list:
            if self.snake.x == b[0]:
                if self.snake.y >= b[1]:
                    s1 = 1.0
                if self.snake.y <= b[1]:
                    s3 = 1.0
            if self.snake.y == b[1]:
                if self.snake.x <= b[0]:
                    s0 = 1.0
                if self.snake.x >= b[0]:
                    s2 = 1.0

        #fruit
        fx = float(self.fruit.x - self.snake.x)
        fy = float(self.fruit.y - self.snake.y)
        return torch.tensor([b0, b1, b2, b3,
                             s0, s1, s2, s3,
```

```
                            fx, fy])

    def get_state3(self):
        #for border and body  in 4 directions
        #border with distances and body with binary vision
        #for the fruit a binary vision in 8 directions

        #border
        b0 = float(self.gridlength-self.snake.x)
        b1 = float(self.snake.y+1)
        b2 = float(self.snake.x+1)
        b3 = float(self.gridlength-self.snake.y)

        #body
        s0 = 0.0
        s1 = 0.0
        s2 = 0.0
        s3 = 0.0

        for b in self.body.list:
            if self.snake.x == b[0]:
                if self.snake.y >= b[1]:
                    s1 = 1.0
                if self.snake.y <= b[1]:
                    s3 = 1.0
            if self.snake.y == b[1]:
                if self.snake.x <= b[0]:
                    s0 = 1.0
                if self.snake.x >= b[0]:
                    s2 = 1.0

        #fruit
        f0 = 0.0
        f1 = 0.0
        f2 = 0.0
        f3 = 0.0
        f4 = 0.0
        f5 = 0.0
        f6 = 0.0
        f7 = 0.0

        if self.snake.x == self.fruit.x:
            if self.snake.y >= self.fruit.y:
```

```
                f2 = 1.0
            if self.snake.y <= self.fruit.y:
                f6 = 1.0
        if self.snake.y == self.fruit.y:
            if self.snake.x <= self.fruit.x:
                f0 = 1.0
            if self.snake.x >= self.fruit.x:
                f4 = 1.0
        for i in range(-self.gridlength, self.gridlength):
            if ((self.snake.x+i) == self.fruit.x and
                        (self.snake.y+i) == self.fruit.y):
                if i >= 0:
                    f7 = 1.0
                if i <= 0:
                    f3 = 1.0
            if ((self.snake.x+i) == self.fruit.x and
                        (self.snake.y-i) == self.fruit.y):
                if i >= 0:
                    f1 = 1.0
                if i <= 0:
                    f5 = 1.0

        return torch.tensor([b0, b1, b2, b3,
                            s0, s1, s2, s3,
                            f0, f1, f2, f3, f4, f5, f6, f7])

    def get_state4(self):
        #capture the whole grid as 2d vector
        #snake = -0.5
        #fruit = 1
        #body = -1
        state = np.zeros((self.gridlength, self.gridlength))

        #to omit any erros as the snake-head tried to leave the grid
        if not self.snake.x >= self.gridlength or self.snake.x < 0:
            if not self.snake.y >= self.gridlength or self.snake.y < 0:
                state[self.snake.y][self.snake.x] = -0.5
        state[self.fruit.y][self.fruit.x] = 1
        for b in self.body.list:
            state[b[1]][b[0]] = -1

        state = torch.tensor(state)
        state = torch.flatten(state)
```

```
        state = state.float()
        return state

    def get_state5(self):
        #captures a 5by5 2d array around the snakes head,
        #body = -1
        #border = -1
        #fruit = 1
        state = np.zeros((5,5))
        for i in range(-2,3):
            for j in range(-2,3):
                x = self.snake.x + i
                y = self.snake.y + j

                if (x >= self.gridlength or x < 0 or
                    y >= self.gridlength or y < 0):
                    state[j+2][i+2] = -1
                for b in self.body.list:
                    if x == b[0] and y == b[1]:
                        state[j+2][i+2] = -1
                if x == self.fruit.x and y == self.fruit.y:
                    state[j+2][i+2] = 1

        state = torch.tensor(state).float()
        return torch.flatten(state)

    def get_state6(self):
        #same as get_state5
        #+ the vector from the snakes head to the fruit
        state = np.zeros((5,5))
        for i in range(-2,3):
            for j in range(-2,3):
                x = self.snake.x + i
                y = self.snake.y + j

                if (x >= self.gridlength or x < 0 or
                    y >= self.gridlength or y < 0):
                    state[j+2][i+2] = -1
                for b in self.body.list:
                    if x == b[0] and y == b[1]:
                        state[j+2][i+2] = -1
                if x == self.fruit.x and y == self.fruit.y:
                    state[j+2][i+2] = 1
```

```

        state = torch.tensor(state).float()
        state = torch.flatten(state)

        head_to_fruit = torch.tensor([self.fruit.x-self.snake.x,
                                        self.fruit.y-self.snake.y]).float()

        return torch.cat((state, head_to_fruit))

    def get_state7(self):
        #captures a 9by9 2d array around the snakes head,
        #body = -1
        #border = -1
        #fruit = 1
        state = np.zeros((9,9))
        for i in range(-4,5):
            for j in range(-4,5):
                x = self.snake.x + i
                y = self.snake.y + j

                if (x >= self.gridlength or x < 0 or
                    y >= self.gridlength or y < 0):
                    state[j+4][i+4] = -1
                for b in self.body.list:
                    if x == b[0] and y == b[1]:
                        state[j+4][i+4] = -1
                if x == self.fruit.x and y == self.fruit.y:
                    state[j+4][i+4] = 1

        state = torch.tensor(state).float()
        state = torch.flatten(state)

        head_to_fruit = torch.tensor([self.fruit.x-self.snake.x,
                                    self.fruit.y-self.snake.y]).float()

        return torch.cat((state, head_to_fruit))

    def get_state8(self):
        #same as get_state7
        #+ the vector from the snakes head to the fruit
        state = np.zeros((9,9))
        for i in range(-4,5):
            for j in range(-4,5):
```

```
                x = self.snake.x + i
                y = self.snake.y + j

                if (x >= self.gridlength or x < 0 or
                    y >= self.gridlength or y < 0):
                    state[j+4][i+4] = -1
                for b in self.body.list:
                    if x == b[0] and y == b[1]:
                        state[j+4][i+4] = -1
                if x == self.fruit.x and y == self.fruit.y:
                    state[j+4][i+4] = 1

        state = torch.tensor(state).float()
        state = torch.flatten(state)

        head_to_fruit = torch.tensor([self.fruit.x-self.snake.x,
                                self.fruit.y-self.snake.y]).float()

        return torch.cat((state, head_to_fruit))

    def env_step(self, action):
        #takes in an action and puts it through the environment
        self.snake.act(action)
        self.body.update(self.snake.changelist, self.snake)
        if self.collisionfruit():
            self.body.grow(self.snake)
            self.body.update(self.snake.changelist, self.snake)
            self.score += 1
            if self.score == (self.gridlength**2 - 1):
                #incase the agent ever manages to fill the whole grid
                return 100
            self.check_fruit_spawn()
            return self.r_fruit
        elif self.collisionborder():
            return self.r_collision
        elif self.collisionself():
            return self.r_collision
        else:
            return self.r_step

    def env_reset(self):
        #resets the environment and return fresh state
        self.score = 0
```

```
        self.snake = Snake(self.snakex, self.snakey)
        self.fruit = Fruit(self.gridlength,
                           spawnset=self.fruitspawnset,
                           spawnx=self.fruitx, spawny=self.fruity)
        self.body = Body(0)
        return self.get_state()

    def env_store_score(self):
        #used to save a list of scores when the agent is training
        self.scorelist.append(self.score)
        self.score = 0


    def render(self, size, textcolor, record=False, path=None):

        #can be used to render the environment
        import pygame
        pygame.init()
        pygame.font.init()
        self.screen = pygame.display.set_mode(((self.gridlength*size),
                                               (self.gridlength*size)))
        pygame.display.init()
        self.screen.fill(self.background)
        for b in self.body.list:
            x = b[0]*size
            y = b[1]*size
            pygame.draw.rect(self.screen, self.bodycolor,
                                            ((x,y),(size,size)))
        pygame.draw.rect(self.screen, self.snakecolor,
                    (self.snake.x*size, self.snake.y*size, size, size))
        pygame.draw.rect(self.screen, self.fruitcolor,
                    (self.fruit.x*size, self.fruit.y*size, size, size))
        pygame.display.update()
        if record:
            pygame.image.save(self.screen,path)

    def save_svg(self,path):
        #can be used to save an svg of the current game state
        d = draw.Drawing(self.gridlength*10,
                        self.gridlength*10, origin=(0,0))
        b = draw.Rectangle(0,0,self.gridlength*10,
                           self.gridlength*10, fill='#485B57')
        s = draw.Rectangle(self.snake.x*10,self.snake.y*10,10,10,
```

```
                        fill='#197416',stroke='black', stroke_width=0.02)
        f = draw.Rectangle(self.fruit.x*10,self.fruit.y*10,10,10,
                        fill='#E92323',stroke='black', stroke_width=0.02)
        d.append(b)
        d.append(s)
        d.append(f)
        for b in self.body.list:
            r = draw.Rectangle(b[0]*10,b[1]*10,10,10,
                        fill='#32B145',stroke='black', stroke_width=0.02)
            d.append(r)
        d.saveSvg(path)

    def renderreplay(self,gridlength, size,
                    snakex, snakey, fruitx, fruity, bodylist):
        #can be used to watch an episode of the agent's training history
        import pygame
        screen = pygame.display.set_mode(((gridlength*size),
                                            (gridlength*size)))
        pygame.display.init()
        screen.fill((96,96,96))
        for b in bodylist:
            x = b[0]*size
            y = b[1]*size
            pygame.draw.rect(screen, self.bodycolor, ((x,y),(size,size)))
        pygame.draw.rect(screen, self.snakecolor,
                            (snakex*size, snakey*size, size, size))
        pygame.draw.rect(screen, self.fruitcolor,
                            (fruitx*size, fruity*size, size, size))
        pygame.display.update()
```
Code 8: Source Code for the Snake Environment

I Source Code: Snake Main Script

```python
###############################################################
# This code is a demonstration how to use the SnakeTrainer
# To execute, please place snake_agent.py, snake_env.py,
# snake_trainer.py and snake_main.py in the same directory
###############################################################


import snake_trainer
import numpy as np

#initializing the trainer
trainer = snake_trainer.SnakeTrainer(agent_id=0, updates_per_step=32,
                    gamma=0.98, lr=0.00005, epsilon=1, epsilon_decay=0.9998,
                    epsilon_min=0.01, statetype=8, nr_hiddenlayers=1,
                    nr_neurons=[128], max_mem_length=10000, gridlength=9,
                    r_fruit=50, r_collision=-50, r_step=0, lr_decay=False,
                    lr_milestones=[80000],
                    lr_gamma=0.5, render_bool=False)


#training the agent
trainer.train_n_episodes(n=200,gridlength=3,epsilon_reset=False)
trainer.train_n_episodes(n=400,gridlength=5,epsilon_reset=False)
trainer.train_n_episodes(n=800,gridlength=7,epsilon_reset=False)
trainer.train_n_episodes(n=1600,gridlength=9,epsilon_reset=False)

#watching a replay of his training games
trainer.watch_replay(episodes=[150,2100],gridlengths=[3,9])

#testing the agent
scores = trainer.test_n_episodes(n=10,gridlength=9,render=False,record=False)
print(f'average: {np.average(scores)}')
print(f'high score: {max(scores)}')
```

Code 9: Source Code for the Snake Main Script