KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

First Cycle 15 credits

Exploring the parameter space of Q-learning for faster convergence using Snake

ANDERS BLOMQVIST

CHRISTIAN ANDERSSON

Stockholm, Sweden 2022

Exploring the parameter space of Q-learning for faster convergence using Snake

ANDERS BLOMQVIST
CHRISTIAN ANDERSSON

Degree Project in Computer Science and Engineering, First Cycle, 15 credits
Date: June 3, 2022
Supervisor: Johan Jansson
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: Undersöker parameteruppsättningar för snabbare konvergens hos Q-learning med Snake

Abstract

In this paper we explore the field of reinforcement learning which has proven to be successful at solving problems of random nature. Such problems can be video games, for example the classical game of Snake. The main focus of the paper is to analyze the speed, measured in Q-table updates, at which an agent can learn to play Snake by using Q-learning, specifically with a Q-table approach. This is done by changing a set of hyperparameters, one at the time, and recording the effects on training. From this, we were able to train the agent with only 225 000 Q-table updates, which took 7 seconds on a regular laptop processor, and achieve a high score of 52 (34 % cover of the grid). We were able to train the agent with only 100 000 updates but it came with reliability issues such as in some training sessions it did not perform well at all.

Sammanfattning

I denna rapport utforskar vi ämnet reinforcement learning som har visat sig vara en framgångsrik metod för att lösa problem vars problemformulering inte är specifikt detaljerad. Sådana problem kan vara datorspel, till exempel det klassiska spelet Snake. Huvudfokuset i rapporten ligger i att analysera hur snabbt en agent kan lära sig spela spelet Snake. Algoritmen som analyseras är Q-learning med ett Q-table där snabbhet mäts i antalet Q-table uppdateringar. Detta görs genom att ändra en mängd hyperparametrar, en åt gången, och notera dess effekt på träningen. Utifrån detta kunde vi träna agenten med enbart 225 000 Q-table uppdateringar, vilket i realtid tog 7 sekunder på en vanlig bärbar dator, som högst uppnådde 52 poäng (34 % täckning av spelplanen). Vi kunde träna på betydlig mindre uppdateringar, närmare 100 000, men med mindre chans att lyckas bra. Ibland kunde en träningssession resultera i mycket dåliga resultat.

Contents

Chapter 1

Introduction

The field of artificial intelligence (AI) is growing faster than ever and it is being used in a lot of different fields for solving complex tasks. Such tasks are often practically impossible to solve with one specific algorithm. By using various implementations of AI, such as machine learning (ML), where the computer automatically learns and improves over time through experience and use of data, tasks like these can be solved. A prominent example of a very complex task solved with ML is Open AI's Dota 2 implementation of a bot which was the first to defeat the world champions at an esports game [1]. The specific algorithm used in this case was a variant of reinforcement learning (RL) which, for example, has a lot of similarities with how a human teaches a dog to complete tasks. The dog does not know what the end goal is but is making decisions based on previous experiences, such as when a reward was given or not. The dog learns what types of consequences its actions have and what to do in order to get rewarded by the human. Similar to the dog, the computer does not follow a strict policy. Instead the programmer sets up rewards and punishments for different kinds of behavior. The computer is solely making decisions without input from the programmer and tries to avoid punishments meaning that the computer can be trained, just like the dog, to solve problems without a specific problem outline or knowledge about the goal.

By using different training strategies the RL algorithm can achieve better or worse results. However, it depends very much on the amount of training, which can stretch from several hours or in the case of Open AI's Dota 2 bot: many months. Such long training times can be problematic regarding logistics and development but also costly if high end hardware is required. The reinforcement learning algorithm has a lot of parameters, known as hyperparameters, which affect the quality of learning but also time taken. By carefully picking appropriate values for these parameters the learning time can greatly vary.

1.1 Research Question

How should the hyperparameters for a RL algorithm, namely Q-learning, be tuned for achieving the best result possible with regards to the least amount of training?

1.2 Societal connection

The program Wallenberg Autonomous systems and Software Program (WASP) states in its long term strategy report that artificial intelligence will come to play a more important role in human society in the future [2]. With widespread changes to many sectors, from the medical sciences to smart cities, a need for faster learning of these intelligent systems might be of importance.

1.3 Scope

We are analyzing the hyperparameters for a Q-learning algorithm in a single discrete environment. We choose to only use one environment instead of multiple. Testing on more environments would be beneficial to get a deeper understanding on how the hyperparameters behave in other circumstances. However this is something we leave to others.

We will be implementing a Q-learning algorithm for the famous Snake game where the score is measured by both food eaten and how much, in percentage, of the playing field is covered by our snake. The training is primarily measured in the amount of Q-table updates, but also episodes (games played).

Chapter 2

Background

2.1 Reinforcement learning concepts

For reinforcement learning problems the learner is called the agent. The agent is the decision maker within an environment. The environment is everything the agent interacts with. The agent is selecting possible actions provided by the environment which also gives feedback to the agent. Feedback consists of rewards defined by the environment which are given upon change of state. A state is often an abstract representation of the environment [3].

2.2 Markov decision processes

Markov decision process (MDP) is a framework describing decision making in discrete, stochastic, sequential environments and how an agent interacts with its environment to achieve a goal [4][3]. The agent inhabiting the environment is, to some extent, able to observe the current state of the environment and, in response, performs an action which subsequently changes the state in a probabilistic manner [5]. After each state transition the agent receives an immediate punishment or reward based on the action performed. The goal of an agent is maximizing a long-term total reward and since the current state affects the possibilities of reaching future state transitions, the actions performed affect not only immediate rewards, but in addition the long-term total reward.

An agent begins by observing the current state and chooses an action based on it. After the decision maker performs an action it moves to a subsequent state and receives a reward or punishment based on the action that it made. In this subsequent (but not always different) state the agent again observes the state and has to choose an action [5].

2.3 Reinforcement learning

Reinforcement learning is an area of machine learning which focuses on goal-oriented machine learning from interaction and how an agent can learn in an uncertain environment. Reinforcement learning does not follow a strict set up policy, but instead explores possible states and actions to figure out which actions yield the highest reward. The action taken in a certain state may not only give an immediate reward, but can also affect rewards in later states. Richard S Sutton and Andrew G Barto describe that trial-and-error search and delayed reward are the two most important distinguishing features of reinforcement learning [3]. The learning agent must have access to some portion of its environment and be able to take steps that affect the state as well as a goal related to the state of the environment. Reinforcement learning thus uses the Markov decision process framework to define how a learning agent interacts with an environment. The agent needs information about its environment which it can interact with as well as a goal.

Reinforcement learning differs from traditional supervised machine learning in that in the former the correct actions for the algorithm to make may not be known. In reinforcement learning there is a tradeoff between exploration and exploitation where exploration describes how the agent explores the environment to learn more about it and which actions to make in certain situations. Exploitation describes how the agent exploits what it has already learned to obtain a reward. Both exploration and exploitation must be balanced in order for the agent to actually learn from its environment with the goal being to improve the performance of the agent over time [3].

2.4 Q-learning

Q-learning is a reinforcement learning algorithm built on MDP which enables the agent to make decisions which can be used to maximize reward. The requirements for Q-learning is a set of possible states $\mathcal{S} = \{s_i\}$ which the agent can perform a set of actions $\mathcal{A} = \{a_i\}$ in. Depending on the state and the action performed by the agent, it is given a numerical reward, r, from the environment. The environment is preferably an abstract representation of the specific reinforcement learning problem. The action, reward and state space is dependent on the environment which in this paper will be finite. A Markov model states that the next state s_{n+1} only depends on the current state s_n and action a_{n+1}. By associating every state change with a reward r_{n+1}, a table,

$\mathcal{Q}$, of $|\mathcal{S}|$ rows and $|\mathcal{A}|$ columns can be created where $\mathcal{Q}(s_n, a_n) = r_n$. This is the simplest form of a Q-function (Q for quality) which is the basis for the Q-learning algorithm. Initially, the Q-table is filled with zeros. For the agent to efficiently use this Q-table it needs to be carefully updated. This is handled by the following update rule, equation 2.1 [6].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot argmaxQ(s_{t+1}, a) - Q(s_t, a_t)) \quad (2.1)$$

which consist of the following parameters:

- $Q(s_t, a_t)$, Q-value at time step t (current Q-value)
- α, learning rate
- r_t, reward when action a_t is taken from state s_t.
- γ, gamma (discount rate).
- $argmaxQ(s_{t+1}, a)$, maximum reward which can be obtained from the next state.

A brief explanation of these hyperparameters are given in section 2.5 table 2.1. By using this rule for each iteration step (an iteration ends when state s_{t+1} is a final or termination state, this is also referred to as an episode) the agent will fill the Q-table with values. When the Q-table has been thoroughly updated it becomes an easy task for the agent to make a decision. Simply by choosing the action a which has the highest reward (Q-value) for state s:

$$a \in argmaxQ(s, a_t), a_t \in A(s) \quad (2.2)$$

By using equation 2.2 the agent can advance through the environment and maximize its rewards over a long period of time by constantly updating the Q-table.

For simple environment state spaces the Q-table approach is viable. It becomes problematic if the state space becomes too large due to it requiring significantly more computational power to update the Q-table. In such cases function approximations can greatly reduce the problem complexity [3]. They can handle problems with very large and continuous action and state spaces which is common in more advanced environments.

2.5 Explanation of the hyperparameters

The named hyperparameters in this report are listed and briefly described in table 2.1. They are essential for the performance of the Q-learning algorithm.

Parameter	**Description**
Grid size	The size of the playing field for which the agent moves around in. This includes walls around each edge.
Food	A positive value given to the agent when eating a food which progresses the game.
Passive	An either positive or negative value given to the agent when a move does not progress or end the game.
Death	A negative value given to the agent when it reaches an end state, such as a wall has been run into.
Learning rate, α	The rate at which the agent learns a new behaviour.
Gamma, γ	A factor which says how important future rewards are compared to current.

Table 2.1: Hyperparameters within the Q-learning algorithm which will be investigated later in the report.

2.6 Previous research

Previous papers have explored how to optimize an agent for the purpose of playing Snake using a deep neural network. The ones we have read have not explored playing Snake using Q-table learning.

Anton Finnson and Victor Molnö explored in the paper *Deep Reinforcement Learning for Snake* how to configure an agent using several variations of a deep Q-neural network to play Snake as best as possible. The state was the entire game environment consisting of the snake, the apple and the walls. They determined that a Q-table works well for simple problems, but was infeasible for other more complex problems due to the large size of the state set. They therefore trained agents using only deep convolutional neural networks and adjusted several parameters to attain the highest possible score. After training on 56 million state-action pairs, which took over five hours, they achieved an average score of 30 points and a high score of 66 points[7].

Chapter 3

Methods

3.1 Implementation of the Snake environment

In order to analyze hyperparameters for Q-learning a Snake environment and a Q-learning algorithm had to be set up. The environment was made using Open AI Gym which is a toolkit for developing and comparing reinforcement learning algorithms [8]. There exist Open AI Gym Snake environments but we chose to create our own implementation from scratch in order to fit our needs. The environment serves the agent with the action and observation space which had to be simplified because of the Q-table limitation as discussed in 2.4.

Instead of using the whole grid as observation space, as done in Finnson and Molnö's implementation, which is an unimaginably large number because of the endless possibilities where the snake head, body and food can be in. Our simplified observation space reduces the possible states $\mathcal{S}$ to only 144 states. This is achieved by an array of six values: [north, east, south, west, col, row] where the north, east, south and west are boolean values which denote whether the snake has a nearby wall (a wall is including the snake's body) in either direction - one cell away. The col and row variables are integer values ranging from [-1, 0, 1] which states how the snake is positioned relative to the food. A negative value indicates that the snake is below the food (row or column), while a positive is above. A zero indicates when the snake is in the same row/column as the food. This means we have: $\underset{2\cdot2\cdot2\cdot2}{24} * 3 * 3 = 144$ possible combinations of states.

The environment also supplies the agent with rewards when performing actions. Three rewards were created: passive, food and death, where passive is given when the snake moves without dying or eating food. Food reward is given when the snake eats food and death reward when the snake collides with a wall or its body. The score counter is incremented after each food eaten, however this variable is only for performance measures.

3.2 Q-learning implementation

A simple algorithm for training the agent was implemented using the update rule, equation 2.1 as its core. The algorithm is described in pseudo code below, algorithm 1. It begins with resetting the environment for each new episode and will play until maximum steps reached or if the snake dies. Maximum steps is reached when the agent does not eat any food and never runs into a wall. As described in section 2.3 a balance between exploration and exploitation needs to be met. This is done by introducing the parameter epsilon, ϵ, which starts off high and gradually reduces towards its minimum value after each episode t. It follows an exponential curve as equation 3.1. Epsilon is a percentage based value where a value of 1.0 results in a 100 % chance of exploration and 0.0 results in a 100 % chance of exploitation.

$$\epsilon(t) = min + (max - min)e^{-dt} \tag{3.1}$$

where d is the decay rate. Both epsilon and decay rate are hyperparameters of interest and epsilon is the only hyperparameter in the implementation which gradually changes after each episode. All other parameters are constant. Epsilon determines whether the agent should perform a random action, which encourages exploration, or choose an action by equation 2.2 which is exploitation.

Algorithm 1 Training the agent using Q-table

```
 1: for $n \leftarrow 0$ through TOTAL_EPISODES do
 2:     reset environment, save initial state s
 3:     steps $\leftarrow$ 0
 4:     while steps $<$ MAX_STEPS do
 5:         a $\leftarrow$ select action depending on epsilon
 6:         take action a, save tuple state, reward from action
 7:         update qtable at (state, a) with update rule
 8:         if reward $>$ 0 then
 9:             steps $\leftarrow$ 0
10:         if action taken led to end state then then
11:             break while
12:         steps $\leftarrow$ steps + 1
13:     reduce epsilon
```

Furthermore an algorithm for playing using the Q-table was also implemented, algorithm 2. It has the equation 2.2 at its core for selecting the optimal action depending on state from the Q-table. This algorithm does not take random actions as the training one does. With algorithm 2, benchmarks of the agent were made by saving the score from each episode. The score is retrieved from the environment after an action has been taken. Each benchmark consists of letting the agent play 100 games in order to get a fair average and high score. Playing more games than 100 would not increase the average score, instead a higher high score would be possible. However this high score is unlikely to reappear due to the large amount of games played. The benchmark ran on the same grid size of 17x12 which results in an effective play area (without walls) of 15x10. Note that the training is done on different grid sizes.

TESTING

Algorithm 2 Let agent play using trained Q-table

```
1:  qtable ← load table from training
2:  scores ← [ ]
3:  for n ← 0 through 100 do
4:      reset environment
5:      steps ← 0
6:      while steps < MAX_STEPS do
7:          a ← argmax(qtable[s])
8:          take action a, save state, reward from action
9:          if reward > 0 then
10:             steps ← 0
11:         if action taken led to end state then then
12:             break while
13:         steps ← steps + 1
14:     add score from environment to scores
```

3.3 Hyperparameters

The specific hyperparameters which were analyzed are found in table 3.1. Each parameter has a default value and an interval. Starting out from the default parameter setup, one parameter was tested at a time to determine the effect a change had on the learning time. We believed that changing one parameter was the best method for recording the effects of it. The parameter which was changed, had training done for each value in the interval with algorithm 1. This procedure generated all data represented under 4.1. The default values for each hyperparameter are shown below and were assumed after trial-and-error when the agent could achieve a satisfactory score and behavior.

Parameter	**Default value**	**Interval**
Grid size	(15,10)	(7,7), (10,7), (15,10), (17,12), (30,25)
Reward passive	-0.04	0.1, 0.04, 0.01, 0.0, -0.01, -0.04, -0.1
Reward food	1	10, 5, 1, 0.5, 0.04
Reward death	-1	-0.04, -0.5, -1, -5, -10
Learning rate	0.1	0.0001, 0.001, 0.01, 0.1, 0.5, 0.9
Gamma	0.90	0.01, 0.5, 0.9, 0.99, 0.999, 0.9999
Min epsilon	0.0001	0.5, 0.2, 0.01, 0.001, 0.0001
Decay rate	0.01	0.2, 0.1, 0.01, 0.0005, 0.0001

Table 3.1: List of hyperparameters which were tested with their default values and tested interval. Note that the grid size parameter is the full grid including walls.

Chapter 4

Results

4.1 Parameter tests

The results from the tests conducted are shown in figures 4.1 - 4.9. Each parameter has all of its interval values, from table 3.1, in the same figure seperated by color. For each graph the average score is shown on the y-axis while the episode count is on the x-axis. The average score for all graphs is calculated over the last 50 episodes. The title of each graph says which hyperparameter being varied, accompanied with a textbox containing the different parameter values being tested. Other values remain at its default value shown in table 3.1. In each graph the rate of convergence is most interesting, meaning that the line assuming the optimal value the fastest has the best rate of convergence.

4.1.1 Grid size

Varying the grid size parameter is shown in figure 4.1. The figure shows similar rates of convergence for the four grid sizes 7*7, 10*7, 15*10 and 17*12, while the grid size 30*25 takes longer to converge. The lines converging to different average scores is caused by the differing grid sizes allowing the snake to grow to different lengths.

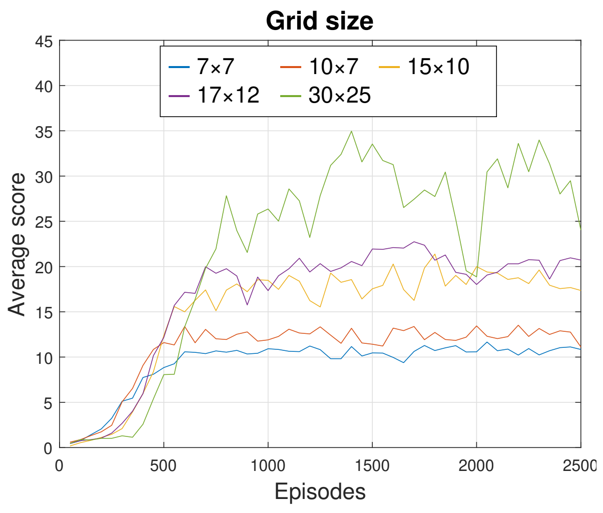

Figure 4.1: Average score from testing different grid sizes.

The most interesting aspect of figure 4.1 is how fast each curve converges towards its maximum value. By looking this aspect we see that the 7*7, 10*7 and 15*10 are the ones who are converging the fastest. It is apparent that, the larger the grid size, the slower it converges. One thing to keep in mind is that larger grid sizes will increase the amount of Q-table updates in an episode which involves more computational work.

4.1.2 Rewards

The passive reward from figure 4.2 is a reward, or punishment, given to the agent when an action does not progress the game. A positive reward encourages the agent to stay alive while a negative punishment forces the agent to quickly hunt for the food. The default parameter value for eating food is +1 which is, relative to the tested passive interval values, much higher.

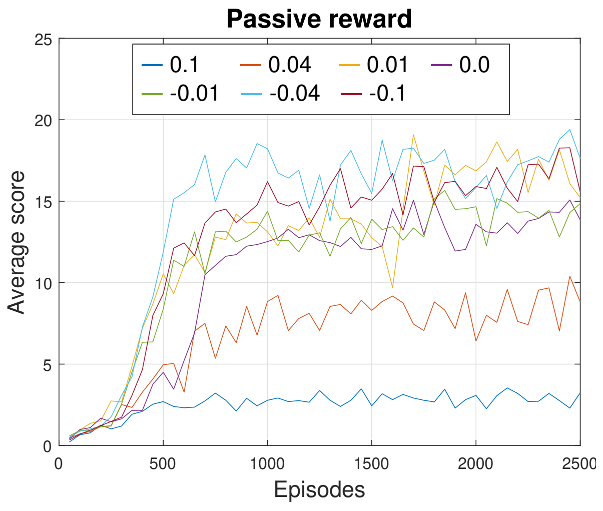

Figure 4.2: Average score from using different passive rewards.

In figure 4.2 we see how the blue 0.1 line gets stuck below 5 average score which indicates a looping behavior. The agent moves around the grid trying only to survive which results in no food eaten and a low score. Thus each episode ends due to maximum steps reached. The best performer was the default value of -0.04 indicated by the cyan colored line at the top, followed closely by -0.1. What really stands out in figure 4.2 is that a too large positive reward yields a poor result while a too large negative punishment does not affect as much. For example by comparing -0.1 and 0.1 curves.

The next parameter of interest is how changing the food rewards affects the learning rate and score. By varying this parameter we get the results in figure 4.3. Here we see that by using either +1 or +5 we get similar results of good quality. However when the food reward is too high, +10, it performs much worse. By using algorithm 2 and visualizing each episode we see that the agent gets stuck in a loop for this value.

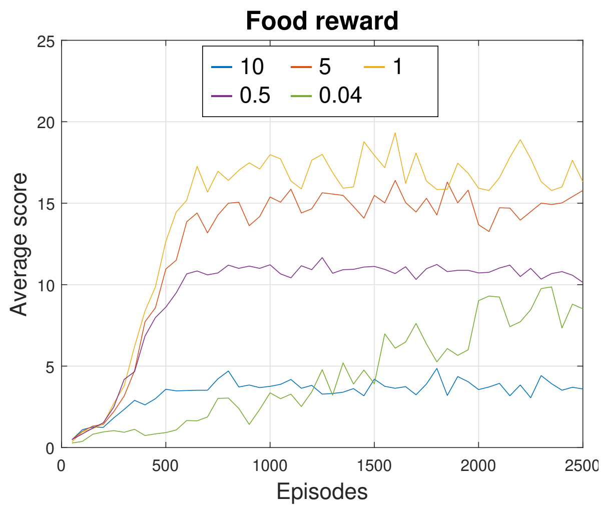

Figure 4.3: Average score from testing different rewards when the agent eats a food.

Lastly regarding environment parameters is the death punishment. In contrast to the food reward, the death punishment does not have as big of an impact which is shown in Fig 4.4. Every tested value except -0.04 have very similar results where the -0.04 got stuck in a loop. The passive parameter has its default value of -0.04 which means the agent is penalized with the same amount when either dying or just moving forward. Otherwise the overall best performer here was -0.5 but with regards to random errors any value of -0.5, -1, -5 and -10 will suffice.

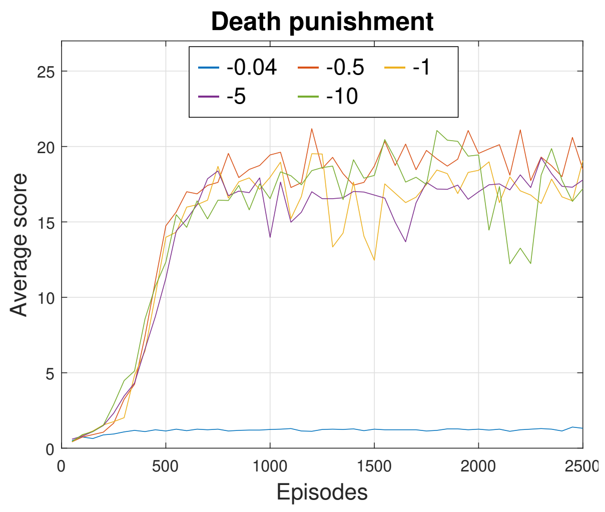

Figure 4.4: Average score when testing different values for death punishment.

4.1.3 Learning rate and Gamma

Now we will present the results from varying the hyperparameters which are directly connected to the Q-learning update rule (equation 2.1). First we begin by varying the learning rate which is shown in figure 4.5. The figure shows a similar rate of convergence for the values 0.1 and 0.01. The agent using learning rate of 0.001 has a slower convergence and the agent using learning rate of 0.0001 has achieved an average score of 1 after 2500 episodes. The green line (α=0.5) initially has a similar convergence rate to the purple (α=0.1) and yellow (α=0.01) lines, but the graph is more choppy/rough and varies between an average score of 10 and 20 later on. We see that a learning rate between 0.1 and 0.01 seems to be the most optimal and therefore try to find a value in between.

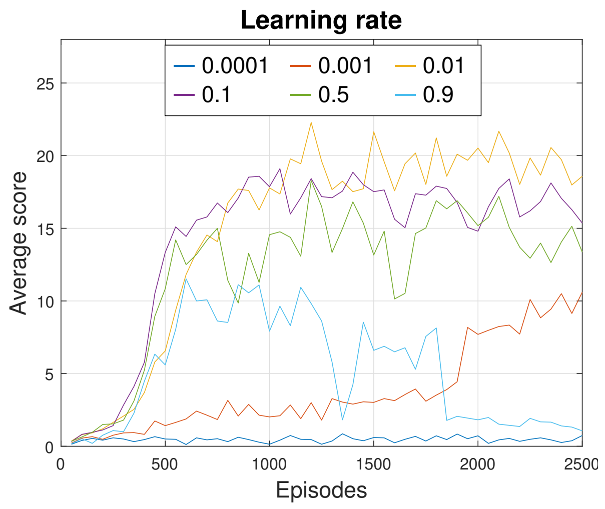

Figure 4.5: Average score from using different learning rates.

From figure 4.5 it became clear that a potential optimal value lies around 0.1 and 0.01. By investigating further and running new tests on 10 values between 0.01 and 0.1 we get the results in figure 4.6 consisting of 10 lines. We find that most lines have roughly the same convergence but a learning rate of 0.05 seems to fit the best for this algorithm.

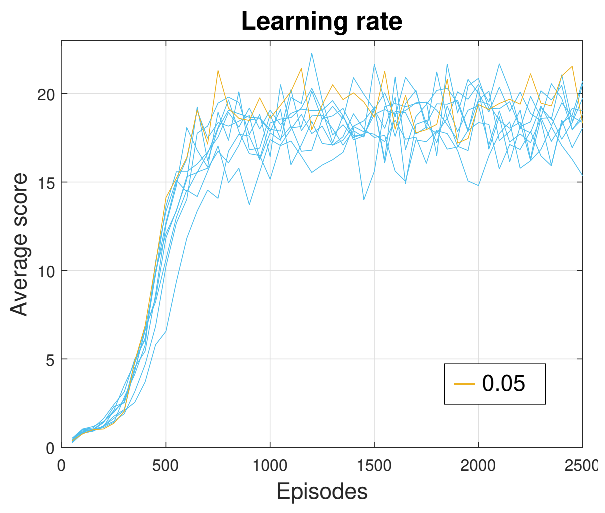

Figure 4.6: Average score by varying the learning rate for each value from 0,01, 0,02 to 0,1. Total of 10 lines where the 0.05 is highlighted.

The last hyperparameter within the update rule (equation 2.1) we investigate is gamma, also known as discount rate, which results are shown in figure 4.7. For all tested values of gamma the graph shows a similar rate of convergence, except for the light blue (γ=0.9999) line which converges to a lower value of approximately 14.

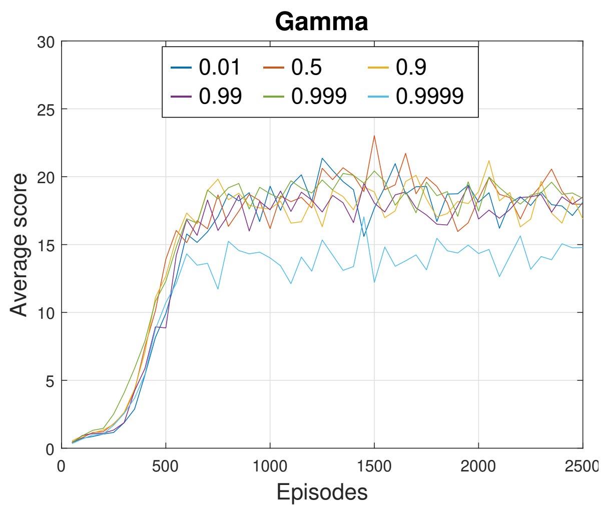

Figure 4.7: Average score from using different values for gamma (discount rate).

4.1.4 Exploration and exploitation

Lastly we investigate how the balance between exploration and exploitation affects the score and learning rate results. Firstly we begin by varying the minimum values for epsilon which results are shown in figure 4.8. Recall from section 3.2 that this is a probability value where 1.0 is treated as 100 % chance for the agent to select a random action while 0.0 is 0 % chance. When the agent does not select a random action it will follow equation 2.2. From the graph we see the fastest rate of convergence for the lowest values, namely the purple (ϵ_{min}=0.001), the green (ϵ_{min}=0.0001) and the cyan (ϵ_{min}=0) lines, which both converge to an average score of approximately 17. While the cyan line seems to achieve a higher score than all other lines, the average score measured after running algorithm 2 for benchmarking results in a lower score than for both the purple and green lines. The yellow line (ϵ_{min}=0.01) converges to an average score of 14, while the lines with the higher values, namely the orange line (ϵ_{min}=0.2) and blue line (ϵ_{min}=0.5) do not converge at all. The key part of this graph is a higher rate of action randomness leads to a lower average score.

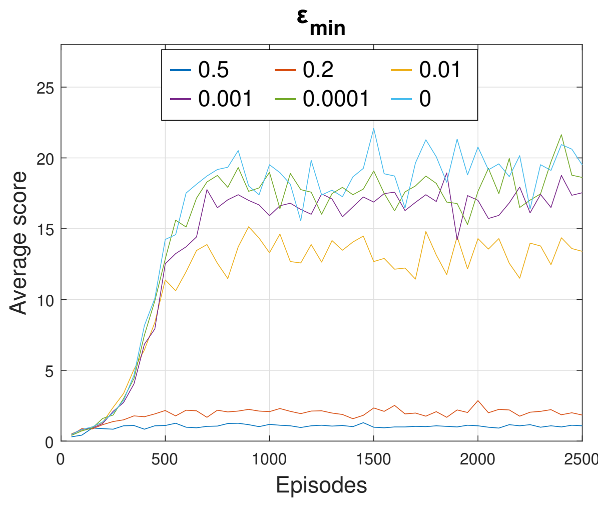

Figure 4.8: Average score by using different minimum values for epsilon.

The other parameter which affects the balance between exploration and exploitation is decay rate. The results for varying decay rate are shown in figure 4.9 and it shows the fastest rate of convergence for the orange line (d=0.1). The green line (d=0.0005) and purple line (d=0.0001) with low decay rates converge significantly slower than the rest of the lines. The blue line (d=0.2) converges slower than both the orange line (d=0.1) and the yellow line (d=0.01) meaning that a higher decay rate than d=0.1 is not preferable.

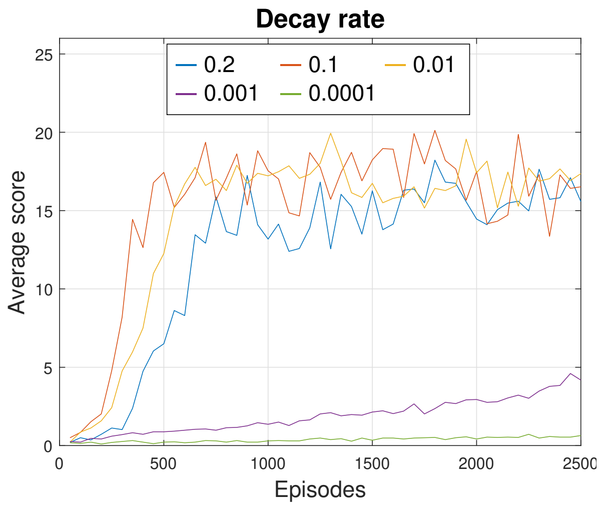

Figure 4.9: Average score from using different decay rates.

4.2 A better parameter setup

By using the results from section 4.1 we construct an optimal parameter setup with regards to fastest learning rate and highest achieved score. This resulted in the values shown in table 4.1. When running algorithm 1 for training with these parameters the graph in figure 4.10 is given. Already before 500 episodes the agent with the optimal parameter setup performs well. Also by running algorithm 2 for benchmarking the trained Q-table, an average score of 23.59 is achieved. The highest score recorded over the 100 games played was 52 which means the snake covers 34 % of the grid. We also trained the optimal parameter setup with only 500 episodes and ran it through the benchmark of 100 games. In that test the agent received an average score of 17.2 and high score of 44 (29 % cover). When training on even fewer episodes than 500 the result did vary a lot. The training could be either as successful as 2500 episodes or not at all and only reaching scores below 5. Instead, increasing the number of episodes (500 or above) gave a much more stable result each time.

The real time taken for running 2500 training episodes (roughly 225 000 Q-table updates) lies around 7 seconds on an Intel core i7 8550U processor at 1.80 Ghz.

Parameter	**Default value**	**Optimal value**
Grid size	(15,10)	(10,7)
Reward passive	-0.04	-0.04
Reward food	1	1
Reward death	-1	-0.5
Learning rate	0.1	0.05
Gamma	0.90	0.90
Min epsilon	0.0001	0.0001
Decay rate	0.01	0.1

Table 4.1: Comparison of the default parameters and the optimal setup.

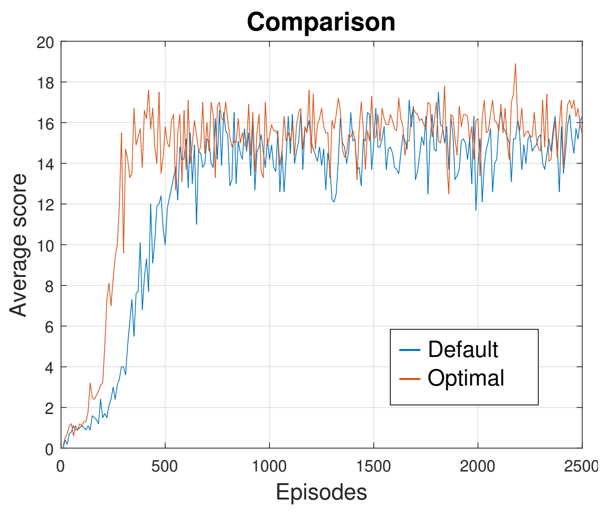

Figure 4.10: Average score for training when comparing the optimal parameter setup with the default. Note that the average score is calculated over the last 10 episodes instead of 50.

Chapter 5

Discussion

5.1 Grid size and observation space

By looking at figure 4.1 regarding grid size we see that training on a smaller grid results in a faster training compared to a large one. This can be explained by a larger size makes it more difficult for the agent to find food resulting in a higher chance of dying along the way towards the food. While training the agent we only need to explore a fixed amount of states-action combinations regardless of grid size and if a smaller grid size can cover those combinations faster, the better.

It can also be due to the observation space which has been purposely simplified for performance gains. This however comes with its limitations due to the small number of states as described in 3.1. The biggest limitation is the agent's inability to differentiate situations where a move causes it to trap itself. An example of a situation where this happens is depicted in fig. 5.1. As a human we can easily see that one move will result in certain death and the other will allow us to move towards the food. The agent can not see the difference between situations (1) and (2) because the observation state is [north: false, east: true, south: false, west: true, col: 1, row: 0] for both. This means the snake will, depending on the training, always choose one action over the other and either always succeed or always fail when a situation like this occurs. This type of situation is more likely to happen if the snake's body is longer, which is a lot more common in larger grid sizes. A consequence of this simplification is that our implementation performs, with regards to average percentage cover, better on small grid sizes and worse on larger ones.

On the contrary, our observation space simplification has made it possible to efficiently use a Q-table approach instead of an approximation based solution such as a neural network. The real training time for consistently getting good results is under 10 seconds on a modern laptop consumer processor which is very fast. Comparing this to Finnson and Molnö, which used a deep neural network combined with a much more complex observation space, we were able to train many times faster and get a similar average score on the same grid size. However the main difference is that theirs perform much better in the late game due to the ability to differentiate situations depicted in figure 5.1. Therefore they will have a higher high score.

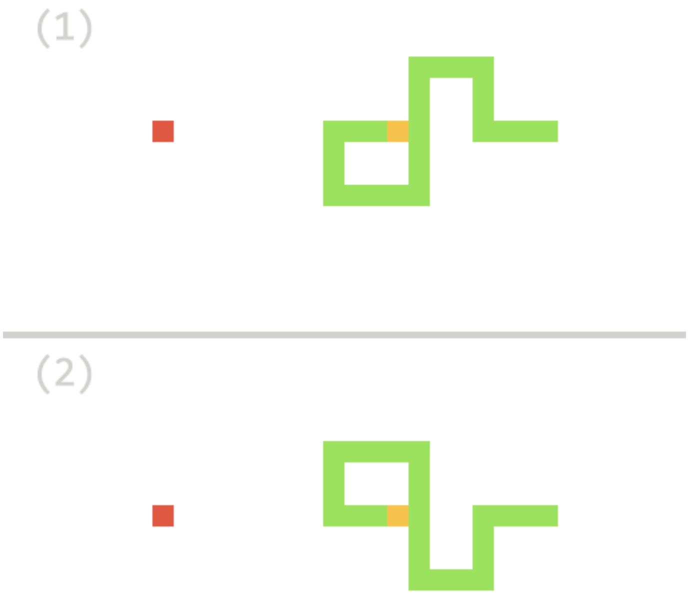

Figure 5.1: Illustration of two different situations (1) and (2), with the same observation state.

5.2 Exploration and exploitation

From the results in section 3 we find that ϵ_{min} and the decay rate (γ) makes the most difference in training time efficiency. The value of gamma and ϵ_{min} can affect how efficient the exploration is and can lead to the agent converging faster. From figure 4.8 it seems clear that a lower ϵ_{min} leads to a faster convergence and a higher average score. Epsilon is decreased after every episode according to the update rule (2) until ϵ_{min} is reached. This explains why the graphs appear similar until ϵ_{min} is reached. After enough exploration has occurred then it seems favorable to exploit the acquired knowledge as much as possible. Since training the learning algorithm with an ϵ_{min} value of 0 results in a lower average score after 2500 episodes, it might be favorable to always keep some amount of exploration during training. From figure 4.9 a decay rate of 0.1 gives the fastest convergence. It also shows that both a decay rate higher and lower than 0.1 gives a slower convergence meaning that a proper balance of exploration and exploitation is necessary to fast reach a good policy for the agent. Our results seem to show that the exploration strategy and a good balance (the so called tradeoff) between exploration and exploitation is the key to faster training times.

It is also worth mentioning the really slow score increase shown by the purple line (d=0.001) in figure 4.9. By running with this value for decay rate and a lot more episodes (15 000) it actually converges to the same score as the rest, around 17. This was something we were used to seeing before testing different values for exploration/exploitation. Initially we had much worse default parameters compared to table 3.1 but after conducting an early test run of different values we quickly realized that the decay rate was way too low. It is therefore not necessary to perform random actions after a while, in our case just after a couple of 100 episodes. The main takeaway from this is that the agent should have at least explored all possible actions and states before stopping with randomness in order to have knowledge about what can be done in each state. In our case we have a small Q-table consisting of only 144 * 4 = 576 possible Q-table values which makes it quite small and fast to explore.

5.3 Gamma and learning rate

We see that changing the value of gamma has only a small effect on the convergence except for values very close to 1. A discount rate close to 0 means that the agent favors immediate reward, while a discount rate close to 1 means that the agent favors long-term reward. It is therefore interesting that this hyperparameter has such a small effect on how the q-table learning algorithm performs in regards to the average score.

In our implementation the agent learns best when the learning rate is approximately 0.05. The agent with learning rate 0.5 quickly gets a good average score, but then drops off compared to the agent with learning rate 0.05. The learning rate specifies how quickly an agent will change its current policy in response to the estimated error each time it is updated with a value close to 0 meaning that it changes it slowly and a value close to 1 means that it changes more quickly. Changing the policy quickly does however not appear to be favorable and this can be explained by the fact that the agent might make an incorrect choice and change its policy incorrectly and therefore achieve a lower average score. Or on the contrary, running the learning algorithm with a high learning rate might make it fit to the problem quicker, by which time the algorithm can terminate. A learning rate between 0.1 and 0.01 for our algorithm, however, seems to be the best for a faster and more stable convergence towards a high average score.

One limitation is that we measure performance based on the amount of episodes performed. The Q-table is updated in every timestep, that is after observing a state, making an action and getting a new state and reward the Q-table is updated. The length of an episode in timesteps can vary greatly and taking this into account may more accurately show the actual time it takes for a learning algorithm to converge.

Chapter 6

Conclusions

The balance between exploration/exploitation needs to be carefully tuned due to it can greatly reduce the amount of required training when using a Q-table approach for Q-learning. The environmental parameters such as food reward, passive reward/punishment and death punishment should all be relatively close to each other but do not affect the learning rate and average score as much as exploration/exploitation parameters does. It is unlikely that the optimal learning rate and gamma values found for Snake in this paper can be applied to all Q-learning problems and the rewards are definitely problem specific, but it has highlighted the importance of a good balance between exploration and exploitation. An unexpected conclusion from this paper is that a learning algorithm using a simplified observation space of a problem can achieve similar scores to that of a complex neural networks with access to the entire environment. It is therefore worth looking into a simplified observation space if one is willing to trade higher scores for faster training times.

6.1 Future work

In this paper, epsilon was the only parameter that decreased as the training progressed. It could be interesting to study how a decrease of other parameters, such as the learning rate, over time could affect the performance of a q-learning algorithm. It could also be interesting to research how other problems can be simplified similarly to Snake in this paper.

Bibliography

[1] Christopher Berner et al. “Dota 2 with Large Scale Deep Reinforcement Learning”. In: *CoRR* abs/1912.06680 (2019). arXiv: 1912.06680. URL: http://arxiv.org/abs/1912.06680.

[2] Wallenberg AI, Autonomous Systems and Software Program. *WASP 2030 – Paving the Way*. https://wasp-sweden.org/about-us/vision-and-strategy/. 2021.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: an introduction*. eng. Adaptive computation and machine learning. The MIT Press, 2018. ISBN: 0262039249.

[4] Michael Littman. *Markov Decision Processes*. eng. 2015.

[5] Martin L Puterman and Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. eng. Wiley-Interscience paperback series. Hoboken: WILEY, 2009. ISBN: 0471727822.

[6] F. S. Melo. “Convergence of Q-learning: a simple proof”. In: *Instituto Superior Tecnico, Institute for Systems and Robotics* (2019).

[7] Anton Finnson and Victor Molnö. “Djupinlärning på Snake”. In: TRITA-SCI-GRU 2019:249 (2019).

[8] *OpenAI gym*. 2019. URL: https://gym.openai.com/.

TRITA-EECS-EX-2022:481