Pavol Jozef Šafárik University

Faculty of Science

REINFORCEMENT LEARNING ALGORITHMS IN THE COMPUTER GAME FLAPPY BIRD

MASTER'S THESIS

Field of Study:	**Informatics**
Institute:	**Institute of Computer Science**
Supervisor:	**doc. RNDr. Gabriela Andrejková, CSc.**
Co-supervisor:	**RNDr. Ľubomír Antoni, PhD.**

Košice 2018 **Bc. Martin Glova**

Acknowledgments

I wish to thank professor Gabriela Andrejkova, for her expert review, willingness, advices and support to write this thesis. I wish to thank my family and friends for their love, support and interest in the topic which boost my creativity and performance.

3608044270054170O

Univerzita P. J. Šafárika v Košiciach
Prírodovedecká fakulta

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Martin Glova
Študijný program:	Informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	Diplomová práca
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Reinforcement learning algorithms in the computer game Flappy Bird

Názov SK: Algoritmy postupného zlepšovania učenia sa v počítačovej hre "Flappy bird"

Cieľ: 1. Describe known results of Flappy Bird's reinforcement learning algorithms.
2. Explore the known learning reinforcement algorithms that are applicable to the Flappy Bird game. Design additional algorithms for this game, concentrate on the usability of neural networks.
3. Implement the proposed algorithms and compare them with the well-known Flappy Bird learning reinforcement learning algorithms so far, given the quality of learning and the time complexity of learning.

Literatúra: [1] N. Appiah, S. Vare: "Playing FlappyBird with Deep Reinforcement Learning". In: Technical Report (2016). Dostupné: http://cs229.stanford.edu/proj2015/362_report.pdf
[2] K. Chen: "Deep Reinforcement Learning for Flappy Bird". In: Technical Report (2015). Dostupné: http://cs231n.stanford.edu/reports/2016/pdfs/111_Report.pdf
[3] G. A. Rummery, M. Niranjan: "On-Line Q-Learning using connectionist systems". In: CUED/F-INFENG/TR 166 (1994).
[4] M. H. Hassoun: Fundamentals of artificial neural networks. MIT Press, Cambridge, 1995.

Vedúci:	doc. RNDr. Gabriela Andrejková, CSc.
Konzultant:	RNDr. Ľubomír Antoni, PhD.
Oponent:	RNDr. František Galčík, PhD.
Ústav :	ÚINF - Ústav informatiky
Riaditeľ ústavu:	prof. RNDr. Viliam Geffert, DrSc.

Spôsob sprístupnenia elektronickej verzie práce: bez obmedzenia

[illegible]

Dátum schválenia: 22.06.2018

Univerzita Pavla Jozefa Šafárika v Košiciach
Prírodovedecká fakulta
Ústav informatiky

Abstrakt

Odbor strojového učenia a umelej inteligencie sa stal v súčastnosti veľmi atraktívnym práve vďaka svojej schopnosti - niekedy viac a niekedy menej - poskytnúť takmer optimálne riešenie na problémy, kde by hľadanie optimálneho riešenia trvalo príliš dlho. Aj keď riešení na tieto ťažké problémy sa črtá viacero (napríklad pomocou kvantových počítačov), umelá inteligencia vytvorená pomocou metód strojového učenia je jedna z mála, ktorá sa už aj dnes ukazuje ako funkčná a reálne použiteľná.

V tejto práci sa venujeme algoritmom posilneného učenia, ktoré predstavujú v obore strojového učenia zvláštnu kategóriu práve vďaka svojmu jedinečnému prístupu učiť sa na základe metódy pokus-omyl. Na počítačovej hre Flappy Bird testujeme algoritmus Q-learning, kde prezentujeme pozitívne výsledky založené na vhodnom definovaní stavového priestoru, politiky výberu akcie, atď. Do algoritmu prinášame nový prvok - výber akcie s maximálnou odmenou v najbližších k krokoch, ktorý výrazne znižuje čas učenia a aj vylepšuje priemerné a najvyššie skóre. Najlepšie výsledky však dosahuje tento algoritmus v kombinácii s modifikovaným algoritmom Deep Q-learning, kde dávame zvláštny dôraz na použitie spojenia posilneného učenia s využitím neurónových sietí so zameraním na využitie doprednej siete namiesto pôvodne navrhovanej konvolučnej.

Kľúčové slová: *strojové učenie, umelá inteligencia, posilnené učenie, Q-learning, Deep Q-learning*

Abstract

Nowadays, machine learning and artificial intelligence have become very attractive because of its ability, sometimes more and sometimes less, to provide an almost optimal solution to problems where searching for the optimal solution would take too long. Although more solutions to these difficult problems are researched (for example by using quantum computers), artificial intelligence, as a part of machine learning, is one of the few which works and is applicable today.

In this thesis, we focus on reinforcement learning algorithms, that represents a specific category in the field of machine learning precisely because of its unique approach based on a trial-error basis. We implement and test Q-learning on the game Flappy Bird where significant results are achieved by appropriate setting of state space, act policy, etc. We introduce a new approach - maximizing k-future rewards policy which decreases learning time and increases maximal and average score significantly. The best results are achieved by using the algorithm combined with modified Deep Q-learning. We place particular emphasis on the use of neural networks with reinforcement learning focusing on the use of a feedforward neural network instead of the originally proposed convolutional.

Keywords: *machine learning, artificial intelligence, reinforcement learning, Q-learning, Deep Q-learning*

Contents

List of Figures

List of Tables

Introduction

Reinforcement learning is a learning that determines what action to choose in a state derived from an environment where the goal is to maximize the agent's reward. Algorithms that implement such approach have to pick actions by the previous experience and should act the way that maximizes its future reward. We can also say that a single action is not so important but the policy which is the sequence of correct actions to reach a goal. So the goal of reinforcement learning algorithms is to build a policy by learning from past good action sequences [5]. For this class of algorithm, two basic things are significant: **trial and error method** and **possible delayed reward** (you should not only act greedy).

Unlike the other two categories of machine learning, supervised learning, where the training data are required or unsupervised learning, where data that are later classified are required, reinforcement learning only needs to define a few basic attributes, such as state space, action space, rewards, etc. and needs to receive information from an environment in the form of reward or penalty. In designing algorithms we focus highly on the applicability of **neural networks** in reinforcement learning as far as it appears to be a method with potential to achieve better results. The main reason that neural networks have been introduced with reinforcement learning is that representation learning with deep learning enables automatic feature engineering and end-to-end learning through updating weight of the neural network so that reliance on domain knowledge is significantly reduced or even removed [6].

In this thesis, we try to create artificial intelligence that could learn to play computer games (or solve similarly defined problems). Games provide excellent testbeds for artificial intelligence algorithms [6]. We try the implemented algorithms in the game Flappy Bird. This relatively simple game can show advantages and disadvantages of many learning algorithms and compare them together. Introduces algorithms could be later used in more real-world applications such as a natural language processing [7, 8, 9, 10], computer vision [11, 12, 13], business management (ads, recommendation, customer management, marketing, etc.) [14, 15, 16, 17, 18], robot navigating [6] in an environment or drones navigation, etc. An autonomous drone has similar problems as are described in this thesis. Especially when we try to cope with indoor navigation with lots of obstacles or navigation in caves, etc. Lot more real-world examples of reinforcement learning applications can be found here [19].

In the thesis, we compare different reinforcement learning algorithms in order to achieve the highest score in the game usually in relation to the number of played games. The artificial intelligence created by such algorithms can even have a much higher score than a human in Flappy Bird. The algorithms' variations are virtually infinitely many even though the number of algorithms themselves is, of course, finite. But each algorithm has some parameters and setting these parameters to different values could significantly change results. Therefore, there is a relatively large scope for testing and analyzing these algorithms. Especially, we take a closer look at possibilities of applying different types of neural networks for reinforcement learning algorithms. The main idea of the concept of using neural networks in reinforcement learning is not so new. One of the first sources of such approach was available in 1993 [20]. However, not so long time ago, in 2013, this approach was tested by using convolutional neural networks for playing computer games with significant results and achieve a great popularity [1].

Chapter 1 contains a comparison and analysis of related works. Chapter 2 contains a more detailed description of reinforcement learning along with several specific algorithms that are later implemented and tested in the thesis. Chapter 3 describes the game Flappy Bird and the implementation of the game which is used in the thesis and some basic attributes for the game are defined here. In Chapter 4, there are the results and observations of the tested algorithms in the game Flappy Bird.

Chapter 1

Related Works

There are more articles or projects concerning searching for an efficient algorithm to play the game Flappy Bird (or other similar games). In this chapter, we review some of the most interesting and describe methods that attempt to solve the problem.

1.1 Replay Memory

The author of the article [21] introduce experience replay, learning action models for planning, and teaching to speed up learning time comparing to adaptive heuristic critic learning architecture [22, 23] and Q-learning [24].

The author introduces **experience replay** as a technique to obtain and use environment outputs more effectively. The article states how training examples are important and that training of a set of training data affects the whole network. Only some of the experiences need to be stored and replayed, but also not as many times as possible. This approach could over-train the network and lead to worse results. For example if in a state s the agent receive 80% of the time a great penalty and 20% of the time no penalty for an action a and if the agent is trained repeatedly by the experiences with no penalty, it is easy to see that this approach is very wrong and leads to not optimal policy.

Another advantage of experience replay is that if an input pattern has not been presented for quite a while, the network could forget what it has learned for that pattern. This problem is called the **re-learning problem**. If you train a model with a lot more patterns and do not repeat them evenly, the network forgets not repeated patterns and only adapts to the frequent ones [20].

The **learning action model** is a way how to simulate real-world hypothetical situations to generate a new state and reward. It is a function $S \times A \to S \times R$, where S is state space, A is the set of actions and R is the set of possible rewards [20]. It is not useful for our case because the environment in the game Flappy Bird is generated randomly and there is no straightforward application of the game in the real world.

The last approach introduced by [21] is to add an expert which generates learning data to teach the agent. The taught lessons are stored similar way as it is for the experience replay and are replayed. As far as we can sometimes consider some of the experiences as optimal (or not optimal) we can train with them more times or each time. This approach we do not consider as far as we want to use pure reinforcement learning with no expert training data.

1.2 Reinforcement Learning in Games

In the following subsections, we describe the main ideas of the methods applied to create an artificial intelligence for playing different games.

1.2.1 Backgammon

In [25], authors use a neural network that is able to teach itself to play **Backgammon** only by playing against itself and learning from the results, based on the TD(λ) reinforcement learning algorithm [26]. The performance of the bot is extremely close to the world's best human players.

1.2.2 Game of Go

Despite the diversity of games as Chess, Checkers, Othello, Backgammon and Scrabble, many of the best playing programs share a simple idea: linear evaluation of many simple features, trained by temporal difference learning, and combined with a suitable search algorithm. The authors of [27] try to return to the strategy in the ancient oriental game of **Go**.

Later on, deep neural networks trained by a combination of supervised learning from human expert games, and reinforcement learning from games of self-play have been introduced and it was the first time that a computer program has defeated a human professional player in the full-sized game of Go [28].

1.2.3 Texas Hold'em

The **DeepStack** algorithm defeated with statistical significance professional poker players in heads-up no-limit **Texas hold'em**. A common feature of games like poker is that players have not perfect information. Poker is a typical game of imperfect information and a longstanding challenge problem in artificial intelligence [29].

1.2.4 Doom

Doom is a First-Person Shooter game in which the player fights against other computer controlled agents or human players in a 3D environment. In [30] authors train artificial intelligence agent using a framework that is based on an asynchronous advantage actor-critic method with convolutional neural networks [31]. This model uses only the recent 4 frames and game variables to predict the next action.

1.2.5 Atari

One of the first successful applications of **neural networks** in reinforcement learning to play computer games was the article by **DeepMind Technologies** [1]. The authors test their approach on seven **Atari 2600** computer games and later on more Atari 2600 games [32]. Figure 1.1 provides sample screenshots from five of the Atari games. One can see that the games have a simple graphics similar to the game Flappy Bird (check Chapter 3, Figure 3.4). For most of the games, they achieved human-level performance or even better [32].

The method which is very similar is described in [33]. The main problem of non-linear approximator as a neural network is that a weight change induced by an update in a certain part of the state space might influence the values in arbitrary other regions. A solution to that problem is an approach similar to experience replay described in the previous section.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data. That is why authors use raw pixels of the game screen as an input for their method to make the method as much general as possible [1]. This way they created a method which performance does not heavily rely on the quality of the feature representation. A new application of the method for a game could be done only by defining the set of actions and by extracting raw pixels of the game. As far as in such images there is lots of noise without information value, we try to use a similar method but by extracting some features.

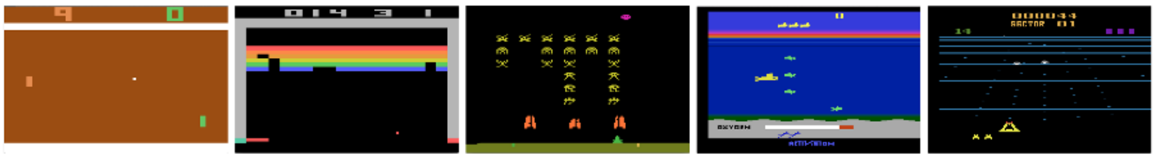


Figure 1.1: Screenshots from five Atari 2600 Games (left to right): Pong, Breakout, Space Invaders, Seaquest, Beam Rider [1].

In computer games like Atari, it is impossible to fully understand the current situation from only the current screen. It is the same in the game Flappy Bird. For instance, when we have only one game screen we do not know what is the speed of the bird. That is why state space is defined as a set of sequences of game screens with performed actions in [1].

The main idea of the **Deep Q-learning** algorithm is that neural network is used as a non-linear function approximator instead linear function approximator ($Q(s, a, \theta) \approx Q^*(s, a)$, where s is a sequence of actions and observations, a is an action and θ stand for the weights of the neural network).

In 2016 a new algorithm called Double Q-learning was introduced with the idea of a tabular setting that can be generalized to work with large-scale function approximation and leads to much better performance on several Atari games [34].

1.2.6 Flappy Bird

Following the pattern of use the deep reinforcement learning to play seven Atari 2600 computer games [1] authors of the article [35] use very similar approach to play the game **Flappy Bird**. A game emulator written in Python is used for testing the algorithm [36]. However, this is a fairly different emulator than the one we use, so results could be affected by this fact. The average score is very small and the best score is also only a few tens. In any case, usage of the deep reinforcement learning is an interesting idea to learn how to play Flappy Bird and it is very interesting to see comparing results of such approach with results that use other algorithms [35].

Another approach to learning how to play Flappy Bird with deep reinforcement learning was done a year before the previously described article. Several levels of the game are used in the article (levels differ in the size of the gap between the top and the bottom pipe) and learning has better results as it is in the previous case. However, it is to be said that another implementation of the game is used too. The states are defined as the game screen (raw pixels information). It is also interesting to note that the authors have defined up to three types of rewards:

1. a reward for "survival",
2. a reward for passing the pipe,
3. a penalty for hitting the pipe or the ground [37].

The Q-learning algorithm is implemented in the next article. State space is defined very similarly as it is in our case, but information whether the bird is alive or not is added. The authors do not use rounding function as it is used in our case so they are generating a too large state space. Learning rate is also defined another way and is changed depending on the state and the chosen action [38].

Two projects which are also inspirations for our approach are available on GitHub. However, the repositories are concerning more about the implementation of algorithms than about more extensive analyzes. First of all the project [39] defines states similarly to those described in our work and the Q-learning algorithm is used. Improving the approach by defining the rounding values has the base in the project [40]. These two projects are also the implementation basis for this work (they also use the same Flappy Bird emulator written in Python).

Very nice results are presented in the article [41]. The bird's velocity is also added to define state space (we also use this approach). Also, **Support Vector Machine (SVM)** are used in this work, but they require training data that must be manually produced.

Comparing other methods such as **heuristic methods** are available in the Ph. D. thesis [42]. At the end of the thesis, it is also interesting to note that the game Flappy Bird, as a game with relatively simple rules, can serve as a good domain to test the effectiveness and features of different learning algorithms.

As the last one to mention in an author who only published his results online in a GitHub repository. He is promising that he has created an immoral Flappy Bird bot. He also published quite long and detailed description with source codes which has also been interesting to read, because he is implementing an algorithm very similar to Deep Q-learning. However, he uses a different simulator which is also simpler because the background is removed and the convolutional neural network is used [43] (so there is less noise in such screenshots of the game).

Chapter 2

Reinforcement Learning (RL)

As it is mentioned in the introduction, reinforcement learning is a learning which chooses an action - a change from one state to another - according to a reward. Reinforcement learning is a field of machine learning for which is significant that **no learning data are required**. The artificial intelligence created by reinforcement learning algorithms is trained according to the **feedback in the form of a reward** (usually it is a real value). The algorithm should in the next same situation (or a very similar situation) act at least slightly better or the same as before (it should not act worse).

To describe process of reinforcement learning, let us have an infinite discrete **time line** which values are natural numbers $T = \mathbb{N} = \{0, 1, 2, 3, \dots\}$. A higher number represents passing time in the reinforcement learning environment which theoretically could last infinitely (in the case of a game it could mean that the game has no end but finishes by performing a wrong action - an example of such game is Flappy Bird). In the environment, we have an **agent** which performs actions according to the policy of an algorithm.

In the following let us have the finite **set of states** or **state space** denoted as S and the finite **set of actions** or **action space** denoted as A. Action space A is usually strictly defined for the same game but state space could be defined differently. State space should be defined in a way that we can uniquely map any information from the environment to one state from the defined set S.

So the agent is in a state from S denoted as $s_t \in S$ in each time unit $t \in T$ and depending on the policy of the learning algorithm it decides which action to take next. The whole process of reinforcement learning is shown in Figure 2.2 and repeats after each change of t. The agent performs actions in the environment in time $t \in T$

according to the last state s_t and the last reward r_t. After the action is performed it receives a new state s_{t+1} and a new reward r_{t+1} and continues with the same steps in the new time unit $t = t + 1$. It is also possible that $s_t = s_{t+1} = s_{t+2} = ... = s_k$ for $k \in T$, so the state is the same for more time units in a row. The rewards are usually defined as real numbers with some logical restrictions (ex. it makes no sense to give a positive reward for wrong actions).

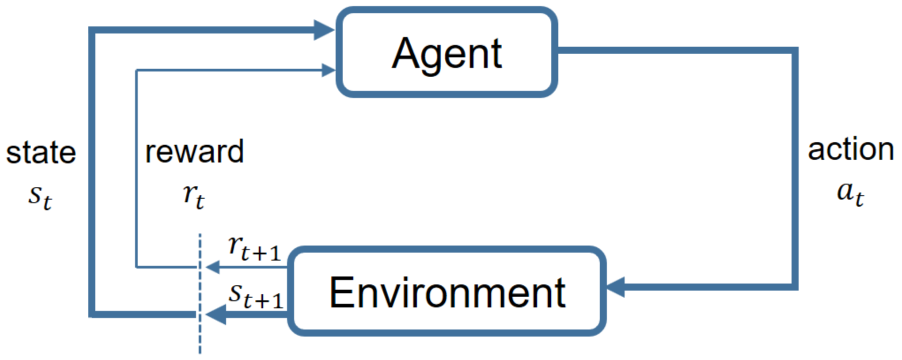


Figure 2.2: The change of states and choosing of the action in the reinforcement learning depending on the feedback from the environment [2].

To pick the best possible reinforcement learning algorithm for a problem there are many possibilities. First, all the algorithms offer own **policy** how the new action is chosen. Also, we can define state space, rewards, etc. differently. By testing different algorithms, trying the different parameters of the algorithms, redefining state space, etc. we can achieve different results and our goal is to find as good learning algorithm as possible.

2.1 Algorithms

The **reinforcement learning algorithms** used in this thesis are described in the following subsections. The algorithms as Q-learning or SARSA are the traditional learning algorithms in the reinforcement learning field. Another described algorithm is Deep Q-leaning introduced in the year 2013 by DeepMind Technologies company [1]. Deep Q-learning uses neural networks which is more usual for supervised learning.

2.1.1 Q-learning

Q-learning is an algorithm which uses **Q-values**. These values are stored as discrete function $Q : S \times A \rightarrow \mathbb{R}$ so the input for the function Q is a couple $\langle s, a \rangle$, where $s \in S$ (S is state space) a $a \in A$ (A is action space). As the output of the function Q we expect a future possible reward - the Q-value. As far as the sets S and A are finite the Q function is a discrete function (for example we can represent the Q-values as a table as it is shown in an example in Table 2.1).

	a_1	a_2
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$
s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$
s_4	$Q(s_4, a_1)$	$Q(s_4, a_2)$
$\vdots$	$\vdots$	$\vdots$

Table 2.1: An example of the function Q with two actions stored in a table.

The pseudocode of the Q-learning algorithm is in Algorithm 1. The algorithm requires defining state space S, action space A and definition of the parameters learning rate η and discount factor γ. The parameter **learning rate** $\eta \in [0, 1]$ means how much of the old Q-value we want to take into account. For example, if $\eta = 0.8$ then we use 20% of the old value we are updating and 80% of the new future and actual reward. The parameter **discount factor** $\gamma \in [0, 1]$ means how much of the next possible reward we want to take into account. The key part of the algorithm is Equation 2.1 which shows updating of the function Q which is at the beginning initialized by random values [44]. If all the actions from A are chosen in all the states infinitely many times, so if the algorithm is executed infinitely many times and parameter η and γ are set properly, then Q-values converge to the optimal values with the probability 1 [45, 46].

Algorithm 1: The pseudocode of the algorithm Q-learning.

```
Initialize all $Q(s, a)$ arbitrarily.
for all episodes do
    Initialize $s_t \in S$, where $t = 0 \in T$.
    while $s_t$ is not the terminal state do
        Choose $a \in A$ using policy derived from $Q$ for the state $s_t$ (e.g. the
         $\epsilon$-greedy policy).
        Take action $a$, observe $r$ and a new state $s_{t+1}$.
        Update $Q(s_t, a)$ by using equation

            $Q(s_t, a) = Q(s_t, a) + \eta(r + \gamma \max_{a' \in A} Q(s_{t+1}, a') - Q(s_t, a)).$     (2.1)

         Update $t = t + 1$.
    end
end
```

2.1.2 SARSA

The algorithm **SARSA** is very similar to the algorithm Q-learning described in Subsection 2.1.1. Actually, in the beginning, it was called **modified Q-learning** by its inventors Rummery and Niranjan (1994) [2]. The pseudocode of the algorithm SARSA is very similar to Algorithm 1, the only difference is that there is not used Equation 2.1 as the update equation, but the following equation

$$Q(s_t, a) = Q(s_t, a) + \eta(r + \gamma Q(s_{t+1}, a') - Q(s_t, a)), \tag{2.2}$$

where a' is chosen by the same policy as a is chosen in the state s_t (for example by the ϵ-greedy policy). So we do not maximize future possible reward in the equation but always use the same policy to chose the best action [47, 2]. That is why we also say that SARSA is **on-policy** algorithm and Q-learning is **off-policy** algorithm.

2.2 Deep RL

We obtain deep reinforcement learning (deep RL) methods when we use deep neural networks to approximate discrete components of the RL (for example in Q-learning the Q function) [6]. In the following subsections, Deep Q-learning is introduced together with neural network optimizers which are later used to update weights of neural network.

2.2.1 Deep Q-learning

Instead of using the discrete Q-function as described in Section 2.1.1, we can also use a neural network with the same input and output as before. Using neural networks is a well-known approach in the field of machine learning. The basics of neural networks are available in many sources such as the book [48]. Some of these terms such as activation function, connections' weights, etc. are also used in this thesis.

The input is defined as a couple $\langle s, a \rangle$, where $s \in S$ (an element of state space) and $a \in A$ (an element of action space) and the output is defined as a real value (a future possible reward). Advantage of such approach is that such network is trained as a whole system. For example let us have a great occurrence of the state $s_i = \langle s_{i1}, \dots, s_{in} \rangle \in S$ so our algorithm is well trained for the state s_i. Then let us also have a very poor occurrence of the state $s_j = \langle s_{j1}, \dots, s_{jn} \rangle \in S$, $s_j \neq s_i$, but $s_j = \langle s_{i1} + \epsilon_1, \dots, s_{in} + \epsilon_n \rangle$ and ϵ_k is very close to zero for all $k \in \{1, \dots, n\}$. In other words states s_i and s_j are very similar but no the same. For the previous approach by using Algorithm 1 with Equation 2.1 or 2.2 the agent should act very well for the state s_i but not for the state s_j, which is almost the same state, as far as training is realized by the discrete Q function. A comparison of such functions is in Appendix B and one can see that the function represented by a neural network created by Deep Q-learning is more smooth and act for similar states similarly unlike the discrete Q function created by the Q-learning algorithm.

In Algorithm 2 we use a neural network to represent our Q function. This causes that a well-trained network for the state s_i is also well trained for the similar state s_j or any other similar states. It is not so straightforward to modify the previous algorithm only by replacing the discrete function with a neural network. Neural networks need training data. The algorithm introduced by DeepMind Technologies company is described in Algorithm 2 and is highly based on the one from [33] which uses multi-layer perceptrons. At first, we need to initialize a replay memory (similar principle as it is "experience replay" in [21]). Such memory is used to store training data. The first training data can be created in some reasonable way, for example as a random playing of the game. We should balance between failure data and successful data if we want to learn faster. Picking an action in the algorithm is done the same way as it is for Q-learning or SARSA. Only updates are done by performing a gradient step and changing weights of the network. In the algorithm $\gamma \in [0, 1]$ is discount factor and has the same meaning as before in the SARSA and Q-learning algorithms. Originally algorithm was introduced with the usage of a convolutional neural network [1], but we try to use a feedforward neural network with hidden layers which is described later.

Algorithm 2: The pseudocode of the algorithm Deep Q-learning.

```
Initialize replay memory $D$ to capacity $n$.
Initialize the function $Q$ with random weights.
for all episodes do
    Initialize $s_t \in S$, where $t = 0 \in T$.
    while $s_t$ is not the terminal state do
        Choose $a \in A$ using policy derived from $Q$ for the state $s_t$ (e.g. the
          $\epsilon$-greedy policy).
        Take action $a$, observe $r$ and a new state $s_{t+1}$.
        Store $\langle s_t, a, r, s_{t+1} \rangle$ in $D$.
        Sample a minibatch of transitions $\langle s_?, a_?, r_?, s'_? \rangle$ from $D$.
        if $s'_?$ is the terminal state then
            Set $y = r_?$.
        else
            Set $y = r_? + \gamma \max_{a' \in A} Q(s'_?, a', \theta)$.
        end
        Perform a gradient step on $(y - Q(s_?, a_?, \theta))^2$.
        Update $t = t + 1$.
    end
end
```

2.2.2 Neural Network Optimizers

In this thesis, we decided to use feedforward neural network instead convolutional neural network when we implement modified Deep Q-learning algorithm. There are still a few questions to answer when implementing a neural network. How input and output of the network are defined is mentioned in the later sections. Among all of these questions like setting the number of hidden layers and neurons in them, choosing activation function or setting other parameters like learning rate, etc., one very important question to answer is which optimizer to use. In this subsection, we discuss what are the options and why we choose the Adam optimizer [3].

Gradient descent is a popular algorithm for maximization or minimization with respect to parameters of differentiable function. **Adam** optimizer is an algorithm which combines and generalizes **AdaGrad** [49] and **RMSProp** [50] algorithms. The main idea behind Adam algorithm is to compute gradient using mini-batch in time stamp t like $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$, where f objective function with parameters θ. Then update biased first $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ and second raw moment estimate $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$, with initialization $m_0 \leftarrow 0$, $v_0 \leftarrow 0$ and parameters β_1 defaultly set to 0.9 and β_2 defaultly set to 0.999. Then we can compute bias corrector like $\widehat{m}_t \leftarrow m_t/(1-\beta_1^t)$ and $\widehat{v}_t \leftarrow v_t/(1-\beta_2^t)$. After all of this update is done like this $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ with two parameters α - learning rate - with default value 0.001 and ϵ defaultly set to 10^{-8} [3].

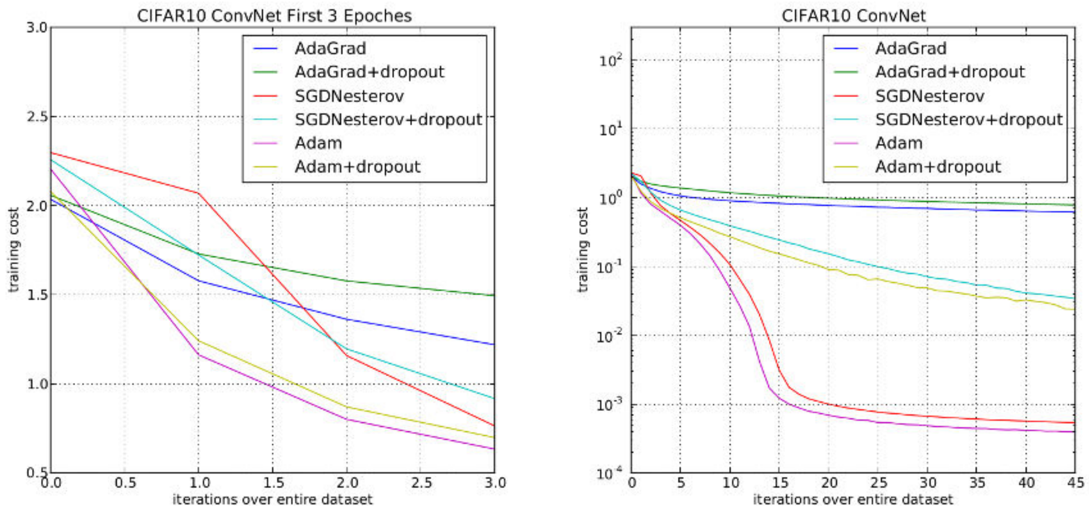


Figure 2.3: Convolutional neural networks training cost. Training cost for the first three epochs (left). Training cost over 45 epochs (right) [3].

Parameters β_1 and β_2 are used to temporally smooth out the stochastic gradient samples obtained during the stochastic gradient descent [51]. Figure 2.3 shows a comparison of the three algorithms AdaGrad, RMSProp and Adam with and without dropout on the CIFAR-10 dataset.

Chapter 3

The Game

Flappy Bird is a game in which the player controls the bird's movement through the gaps between pairs of pipes. Figure 3.4 shows **the configuration of the game**. The player can see up to two of the following pipes. The bird moves forward (along the horizontal axis) with a constant velocity, but his movement along the vertical axis is more complicated and is possible to influence it by the player's control. Figure 3.5 shows that the horizontal velocity cannot be affected by the player and is always 4 forward. In the figure, $\rightarrow 4$ means that we increase the horizontal position by 4 forward in each step.

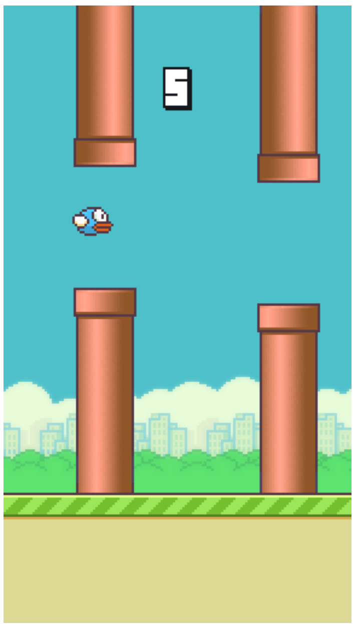


Figure 3.4: An example of the game configuration.

If the bird is not controlled by the player, then its **vertical velocity** (hereinafter referred to as only **velocity**) decreases in each step by 1 downwards until his velocity

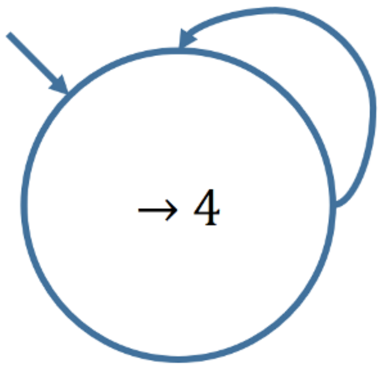


Figure 3.5: The change of the horizontal velocity.

is 10 downwards. In the beginning, its velocity is 9 upwards and in each next step, according to mentioned, it is decreased by 1 downwards until it reaches velocity 10 downwards. The only way how its velocity could be changed is to let the bird flap, so the player calls the flap action to set its upwards velocity. It is possible to call the flap action at any time. After flapping, the bird's velocity is immediately 9 upwards and in every next step, it is decreased by 1 downwards if the player does not call the flap action again. The velocity upwards decreases to 0 and continue increases by 1 until it is 10 downwards again. Figure 3.6 shows how the velocity of the bird is changed. In the figure orange arrows represent the flap action, green arrows represent not flap action and the red arrow points to the starting state. In the figure "s n" stands for vertical change of the bird's position upwards by n if $s = \uparrow$, downwards by n if $s = \downarrow$ or no change if $s = \epsilon$ (empty string) for all $s \in \{\uparrow, \downarrow, \epsilon\}$ and $n \in \{0, \ldots, 10\}$.

The game randomly generates a pair of vertical pipes with a constant width (the width is set to 52), a constant size of the gap between them (the size is set to 100) and a constant distance between every two consecutive pairs of pipes (the size is set to 92). Restrictions for the minimal lengths of pipes are set to 80 for the top pipes and 82 for the bottom pipes. So the maximal length between the smallest top pipe and the next smallest bottom pipe is 242 (and vice versa). All these constants are also shown in Figure 3.7.

By every bird's pass through the pair of pipes player's score increases by 1 (the score is 0 at the beginning). The goal of the game is to pass as many pairs of pipes as possible, to maximize the player's score, without touching the ground or any of the pipes (if any of this happen, the immediately game finishes).

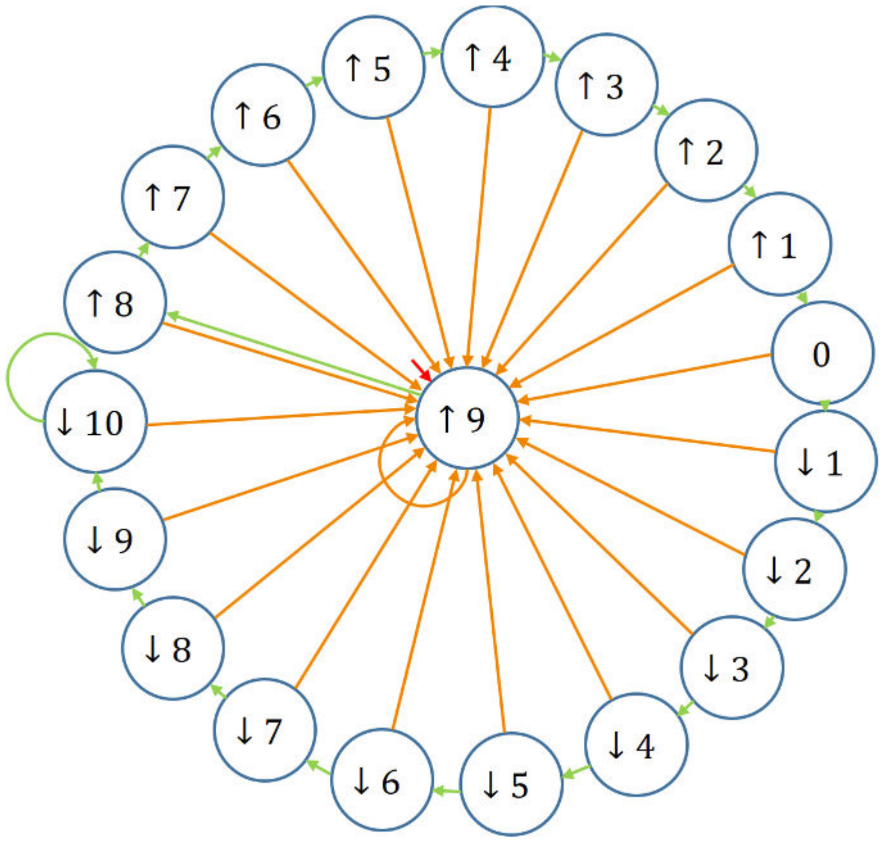


Figure 3.6: The change of the vertical velocity.

The first version of the game Flappy Bird was invented by **Dong Nguyen** in 2013 and was published by the **dotGears** company [52]. The game support was stopped only a year ago and the game was removed from official sources [53].

An open-source **Python** implementation of the game is used in this thesis [54]. An advantage of such approach is that image processing of the game screen is not necessary to get the configuration of the game because all necessary information is sent to learning algorithms by the variables of the game.

In the game Flappy Bird, we can consider one unit of time, as defined in Chapter 2, the time until one game configuration is redrawn on the game screen to another. As the agent, we consider the bird. Action space A we define as a two-element set containing the flap and not to flap action (these are the only actions we can perform in the game in each time unit). Stace space S could be defined in many ways. The ways how it is done could significantly change the results of tested algorithms. It could affect the quality of the learning or affect the learning time. Different ways of defining state space are described in Section 3.1.

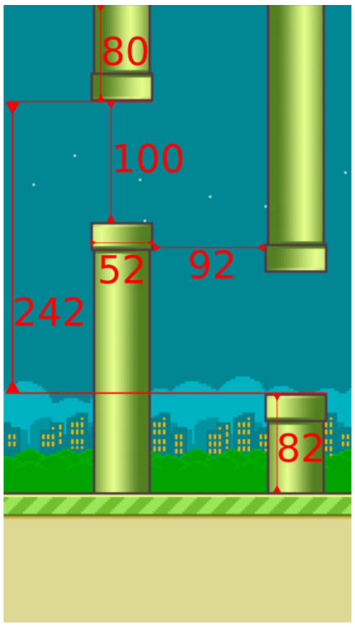


Figure 3.7: The constant sizes of the game configurations.

3.1 State space

State space could be defined differently. For the game Flappy Bird, we use a few different ways of defining state space. All of them are based on the same principle - depending on the position of the agent and the nearest pipes' position.

All the pipes are equally spaced but their height is generated randomly. We can define the state as a couple $\langle x, y \rangle$. The variable x represents the distance of the leftmost pixels of the bird to the rightmost pixels of the pipe nearest to the bird. The variable y represents the distance of the bottommost pixels of the bird the topmost pixels of the bottom pipe nearest to the bird. Such approach is shown in Figure 3.8. It could generate at most $|X \times Y|$ states if X represents all possible values for the variable x and Y represents all possible values for the variable y. Because the cardinality of such set could be very high, we sometimes use **rounding** to the nearest multiples of 5, 10, 15, etc., depending on how much we want to reduce cardinality of state space.

Definition 3.1 Let us have $x \in \mathbb{R}$ and $r \in \mathbb{N}^+$. The function $rnd : \mathbb{R} \times \mathbb{N}^+ \to \mathbb{Z}$ is called the ***rounding function*** and

$$rnd(x, r) = r * \lfloor x/r \rceil.$$

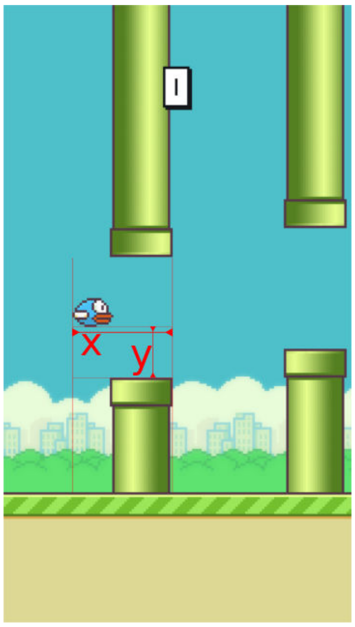


Figure 3.8: Vertical and horizontal distance from the bird to the next pipe.

For example let us have configurations $\langle 98, 50\rangle$ and $\langle 102, 48\rangle$. And also let us have rounding set to 5. Our approach described above would map both configurations to the same state $\langle 100, 50\rangle$ because of rounding to the nearest multiple of 5.

We also know that the situation could be very different when we have the state $\langle 40, 3\rangle$ and the bird's velocity is 9 downwards or we have the same state and the velocity is 5 upwards. In the first case if we do not perform the flap action in the next time unit the state would be $\langle 36, -6\rangle$ which means the death of the bird because it already hit the bottom pipe. However, if the velocity is 5 upwards the next state, without performing the flap action, would be $\langle 36, 8\rangle$ which is alright. That is why we also add velocity to define state space.

Now we can define the states conversion function as $X \times Y \times R \times V \to S$, where $X \subseteq \mathbb{R}$ represents all the different distances of the leftmost pixels of the bird to the rightmost pixels of the pipe nearest to the bird, $Y \subseteq \mathbb{R}$ represents all the different distances of the bottommost pixels of the bird the the topmost pixels of the bottom pipe nearest to the bird, $R \subseteq \mathbb{N}^+$ represents all the rounding values and $V = \{v : v \in [-9, 10] \cap \mathbb{Z}\}$ is the bird's velocity where the negative values mean upwards direction and the positive values mean downwards direction. Then $S = \{\langle x, y, v\rangle : x \in X_S, y \in Y_S, v \in V\}$ is the defined state space, where $X_S = X \cap M_r \cup \{rnd(\min(X), r), rnd(\max(X), r)\}$ and $Y_S = Y \cap M_r \cup \{rnd(\min(Y), r), rnd(\max(Y), r)\}$, where M_r is the set of all multiples of $r \in R$ and rnd is the rounding function defined in Definition 3.1.

Note that state space could be defined arbitrarily. The described approach is only the one of many. For the sake of generality we expand the **states conversion function** (hereinafter ***scf***) for n next pipes and also define roundings for both vertical and horizontal distances. We can really do it as far as the distance between two consecutive pairs of pipes is constant (as described in Chapter 3).

Definition 3.2 Let $n \in \mathbb{N}^+$ be the number of the next pipes we consider in state space, $X \subseteq \mathbb{R}$ be the set of all the different distances of the leftmost pixels of the bird to the rightmost pixels of the pipe nearest to the bird, $Y^n \subseteq \mathbb{R}^n$ be the set of all the different n-tuples where ith element of each n-tuple in the set represents distance of the bottommost pixels of the bird to the topmost pixels of the bottom ith next pipe for $i \in \{1, ..., n\}$, $R_X, R_Y \subseteq \mathbb{N}^+$ be the sets of all the rounding values for the horizontal/vertical distances, $V \subseteq \mathbb{Z}$ is the bird's velocity where the negative values mean upwards direction and the positive values mean downwards direction and $S = \{\langle x, y, v\rangle : x \in X_S, y \in (Y_S)^n, v \in V\}$ is **state space**, where $X_S = X \cap M_{r_X} \cup \{rnd(\min(X), r_X), rnd(\max(X), r_X)\}$ and $Y_S = Y \cap M_{r_Y} \cup \{rnd(\min(Y), r_Y), rnd(\max(Y), r_Y)\}$, where M_{r_Z} is the set of all multiples of $r_Z \in R_Z$ for $Z \in \{X, Y\}$ and rnd is the rounding function defined in Definition 3.1. The function $scf : X \times Y^n \times R_X \times R_Y \times V \to S$ is called the ***states conversion function*** and if $x \in X$, $y = \langle y_1, \dots, y_n\rangle \in Y^n$, $r_X \in R_X$, $r_Y \in R_Y$, $v \in V$ and rnd is the rounding function defined in Definition 3.1, then

$$scf(x, y, r_X, r_Y, v) = \langle rnd(x, r_X), \langle rnd(y_1, r_Y), \dots, rnd(y_n, r_Y)\rangle, v\rangle.$$

In our case V is stable defined as $V = \{v : v \in [-9, 10] \cap \mathbb{Z}\}$.

For $n = 1$ we have the same function as is described above. The greater the n value, the greater the cardinality of state space but also better accuracy of the game configuration conversion. Hence, let us describe how cardinality of state space increase with greater n. State space is the set of triples $\langle x, y, v\rangle$, where $x \in X_S$, $y \in (Y_S)^n$ and $v \in V$. So the cardinality of state space is $|X_S| * |(Y_S)^n| * |V| = |X_S| * |Y_S|^n * |V|$.

For example let us have $X = [0, 8]$, $Y = [-3, 3]$, $V = \{-1, 0, 1\}$, $n = 1$, $r_X = 5$ and $r_Y = 5$. By using the defined states conversion function for all values from X, Y and V we see how state space is defined. First we see that

$$\begin{aligned}
X_S &= X \cap M_{r_X} \cup \{rnd(\min(X), r_X), rnd(\max(X), r_X)\}\\
&= \{0, 5\} \cup \{5 * \lfloor 0/5 \rceil, 5 * \lfloor 8/5 \rceil\}\\
&= \{0, 5\} \cup \{0, 10\}\\
&= \{0, 5, 10\}
\end{aligned}$$

and

$$\begin{aligned}
Y_S &= Y \cap M_{r_Y} \cup \{rnd(\min(Y), r_Y), rnd(\max(Y), r_Y)\}\\
&= \{0\} \cup \{5 * \lfloor -3/5 \rceil, 5 * \lfloor 3/5 \rceil\}\\
&= \{0\} \cup \{-5, 5\}\\
&= \{-5, 0, 5\}.
\end{aligned}$$

So state space is in this case defined as

$$\begin{aligned}
S &= \{\langle x, y, v\rangle : x \in X_S, y \in (Y_S)^n, v \in V\}\\
&= \{\langle x, y, v\rangle : x \in \{0, 5, 10\}, y \in \{-5, 0, 5\}, v \in \{-1, 0, 1\}\}\\
&= \{\langle 0, -5, -1\rangle, \langle 0, -5, 0\rangle, \langle 0, -5, 1\rangle, \langle 0, 0, -1\rangle, \langle 0, 0, 0\rangle, \langle 0, 0, 1\rangle,\\
&\quad \langle 0, 5, -1\rangle, \langle 0, 5, 0\rangle, \langle 0, 5, 1\rangle, \langle 5, -5, -1\rangle, \langle 5, -5, 0\rangle, \langle 5, -5, 1\rangle, \qquad (3.1)\\
&\quad \langle 5, 0, -1\rangle, \langle 5, 0, 0\rangle, \langle 5, 0, 1\rangle, \langle 5, 5, -1\rangle, \langle 5, 5, 0\rangle, \langle 5, 5, 1\rangle,\\
&\quad \langle 10, -5, -1\rangle, \langle 10, -5, 0\rangle, \langle 10, -5, 1\rangle, \langle 10, 0, -1\rangle, \langle 10, 0, 0\rangle,\\
&\quad \langle 10, 0, 1\rangle, \langle 10, 5, -1\rangle, \langle 10, 5, 0\rangle, \langle 10, 5, 1\rangle\}.
\end{aligned}$$

We can easy see that for any input from defined X, Y and V the function scf yields a triple from S defined in Equation 3.1 (ex. $scf(1, -2, 5, 5, 0) = \langle 0, 0, 0\rangle \in S$, $scf(2.49, 2.51, 5, 5, -1) = \langle 0, 5, -1\rangle \in S$, etc.).

Chapter 4

Results

One of the most important goals of the thesis is to find an algorithm with appropriate setting of parameters to be optimized with respect to the highest or average score or learning time. This chapter presents what results have been achieved and brings new ideas in the Q-learning algorithm and uniquely present results of the combination of Q-learning and Deep Q-learning using feedforward neural network.

4.1 Simple Greedy Algorithm

The first tested algorithm is a greedy algorithm described in Algorithm 3. This algorithm uses no learning and only use a simple rule: flap anytime vertical distance of the bird to the next bottom pipe could be in the next state less than zero (so the agent could hit the pipe). The average score of the algorithm for 5,000 games is 143.4292 and with individual scores of games (the red dots) are shown in Figure 4.9. The algorithm is implemented mostly for comparison with other algorithms and future use as a part of other algorithms.

Algorithm 3: The pseudocode of the simple greedy algorithm, where y is the vertical distance of the bird to the next bottom pipe.

```
if $y < 10$ then
    return $FLAP$
else
    return $NOT\ FLAP$
end
```

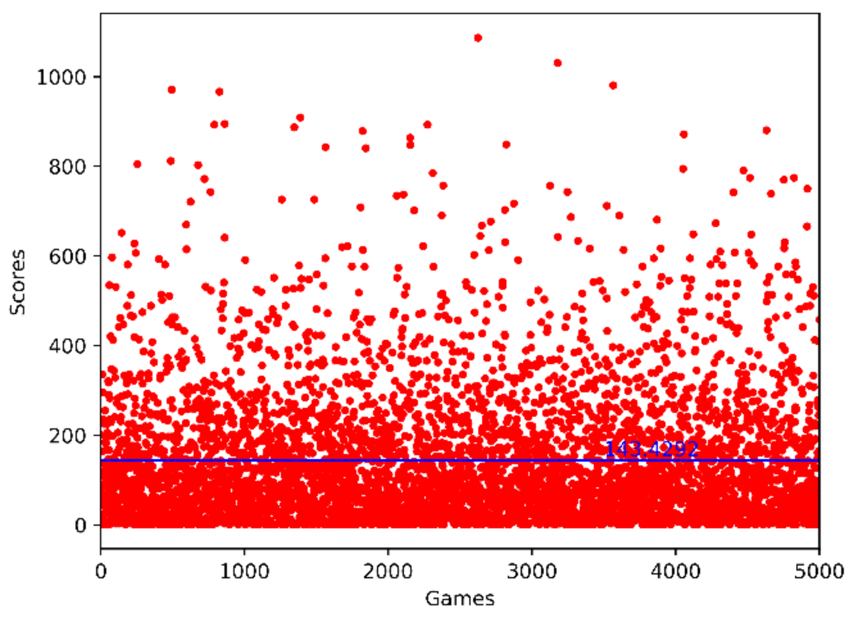


Figure 4.9: Scores of 5,000 games (the red dots) with the highlighted average score 143.4292 (the blue line) played by Algorithm 3.

4.2 Advanced Greedy Algorithm

Another greedy algorithm is presented in Algorithm 4. The algorithm uses no learning and flap only if it is necessary - so only if in the next twenty states the bird would hit the ground of the bottom pipe. Twenty is chosen because it is the minimal time until the bird is in the same vertical position as it was before it flaps the last time. Or in other words: twenty is the number of the states which are affected by flapping. The last restriction is that the bird would flap only if it does not cause hitting the pipe in any of the twenty next steps. The average score of the algorithm for 5,000 games is 144.3628 and with individual scores of games are shown in Figure 4.10. So the advanced greedy policy does not change the average score significantly but still is better than Simple Greedy Algorithm presented in the previous section. The pipes positions in played games are the same as it is in results for the simple greedy algorithm in Figure 4.9, so both algorithms are tested using the same data.

Algorithm 4: The pseudocode of the advanced greedy algorithm.

```
$have\_to\_flap \leftarrow$ False
for $state$ in $next$ $20$ $states$ do
    $x \leftarrow$ getX($state$)
    $y \leftarrow$ getYWithoutFlap($state$)
    if $y \leq GROUND\_BASE$ or ($x \leq PIPE\_WIDTH + BIRD\_WIDTH$
     and $y \leq 0$) then
        $have\_to\_flap \leftarrow$ True
    end
end
if $have\_to\_flap$ then
    for $state$ in $next$ $20$ $states$ do
        $x \leftarrow$ getX($state$)
        $y \leftarrow$ getYWithFlap($state$)
        if $x \leq PIPE\_WIDTH + BIRD\_WIDTH$ and
         $y \geq VERTICAL\_GAP\_SIZE - BIRD\_HEIGHT$ then
            return $NOT$ $FLAP$
        end
    end
    return $FLAP$
end
return $NOT$ $FLAP$
```

4.3 Q-learning

In the following subsections, we focus on searching for the optimal parameters for the Q-learning algorithm described in Algorithm 1.

4.3.1 ϵ-greedy Policy

The policy depends on the ϵ value or better say whether ϵ-greedy policy is used or not. If ϵ is not used then it means there is no significant difference between SARSA and Q-learning as far as during update of Q-value SARSA uses ϵ-greedy policy and Q-learning maximize the values (we are now comparing the update Equations 2.1 and 2.2). Setting ϵ to some non zero values is a mechanism to force the algorithms to be

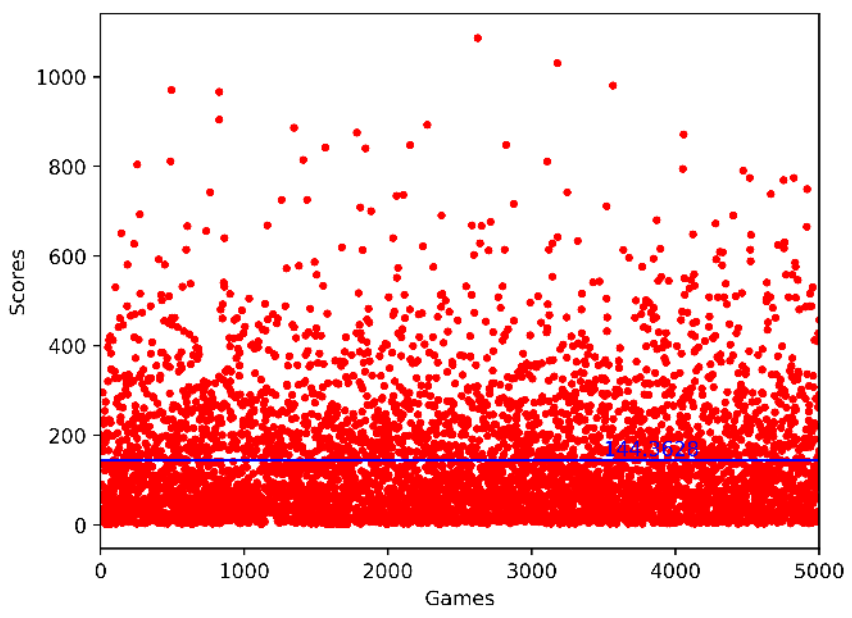


Figure 4.10: Scores of 5,000 games (the red dots) with the highlighted average score 144.3628 (the blue line) played by Algorithm 4.

able to converge to optimal Q-values. Otherwise, it would be impossible to prove such statement. This parameter ensures that we are also likely to choose other previously unselected actions to also test whether the future reward is not higher by choosing such action.

Figure 4.11 shows that there is not a big difference in using or not using ϵ-greedy policy. But using ϵ-greedy policy generates slightly better results. We performed experiments consisting of 50,000 iterations and averaged the results of 100 random simulations for the scores of the last 1,000 games in each iteration.

The value of ϵ is changed by the following equation

$$\epsilon_k = \frac{1}{k}, \tag{4.1}$$

where k is the number of iteration (the actual number of played games). All other parameters in the experiments of this subsection are set as follows: discount factor $\gamma = 1.0$, learning rate $\eta = 0.7$, the reward $r = 1$ for alive states and $r = -1000$ for the three last states and the rounding values are set as follows $r_X = r_Y = 5$ with $n = 1$ (the meaning of the parameters is described in Sections 2.1 and 3.1).

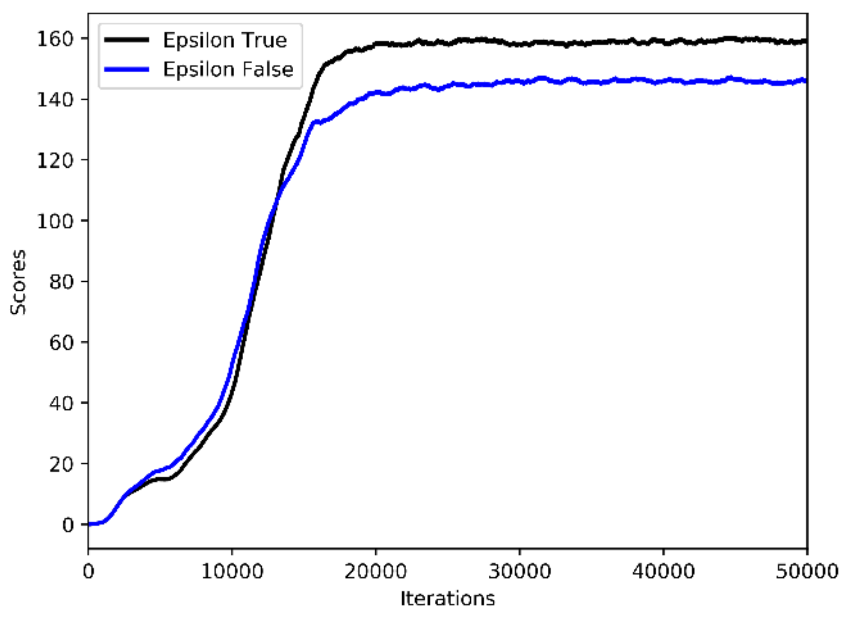


Figure 4.11: Using and not using ϵ-greedy policy.

4.3.2 Rewards

Rewards are signals from the environment which should lead the agent to act better. We define two rewards: rewards for keeping alive and penalization for death. It is obvious that setting these parameters should be done the way that reward should represent positive value and penalization negative. Figure 4.12 shows how changing the rewards affect results. From the figure, it is obvious that the most optimal setting is to set reward to a small positive value and penalization to a big negative value.

It also depends how many of the last states should be penalized in the game Flappy Bird. We expect that only the last state is not responsible for the death of the agent. Figure 4.13 shows that the optimal value for the number of the last penalized states is somewhere around 4 or 3. Values higher than 5 start decrease and values less or equal to 2 are not the optimal ones either.

All other parameters in the experiments of this subsection are set as follows: discount factor $\gamma = 1.0$, learning rate $\eta = 0.7$, the ϵ policy is set to true and the rounding values are set as follows $r_X = r_Y = 5$ with $n = 1$ (the meaning of the parameters is described in Sections 2.1 and 3.1). The colored trending lines represent average scores of the last 1,000 games with the values of reward and the number of the penalized states appertain to the values in the legend on the left side of the figures.

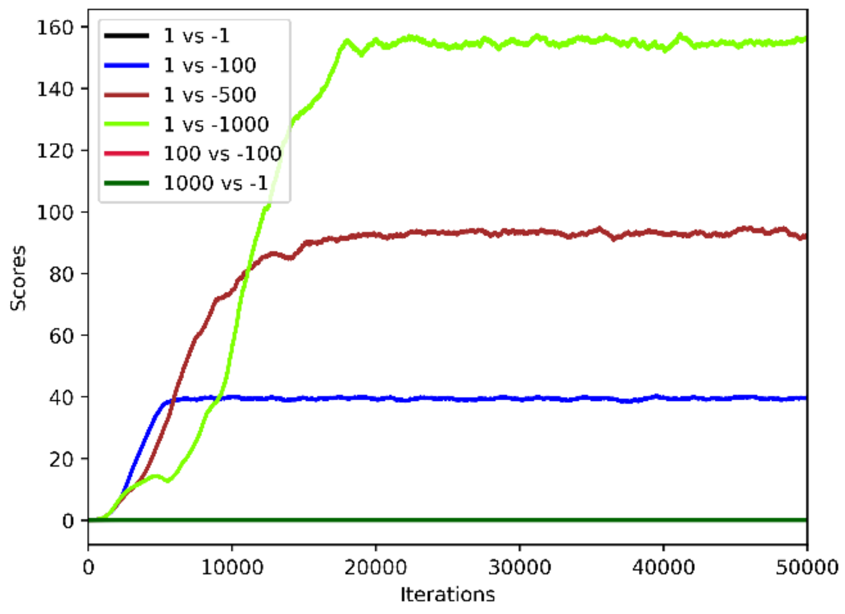


Figure 4.12: A change of rewards for 50,000 iterations and averaged the results of 25 random simulations.

4.3.3 Discount Factor

Discount factor is a parameter in Q-learning. It is denoted by variable γ (algorithms are described in Section 2.1).

Figure 4.14 shows how a change of this parameter influence learning by the algorithm Q-learning on the same 15,000 games. In Figure 4.15, on the other hand, it is shown that when we take 50,000 iterations and averaged the results of 100 random simulations, the results are slightly changed - the value 0.9 shows very unstable. We can see that algorithm efficiency decreases by decreasing the value of the parameter except for values close to 1. In fact, the parameter determines how much we take into account the future possible reward. By setting the parameter to a very low value we only take into account actual reward which is not alright because it could be that the bird is a few states before certain death but as far as it ignores future penalty for the death it only receives reward for keeping survive (even if the next state is the end of the game).

In Figure 4.14 we set the parameters as follows: learning rate $\eta = 0.8$, the reward $r = 1$ for alive states and $r = -1000$ for the three last states, the rounding values are set as follows $r_X = r_Y = 5$ with $n = 1$ and the ϵ policy is set to false (the meaning of the parameters is described in Sections 2.1 and 3.1). Results in Figure 4.15 has

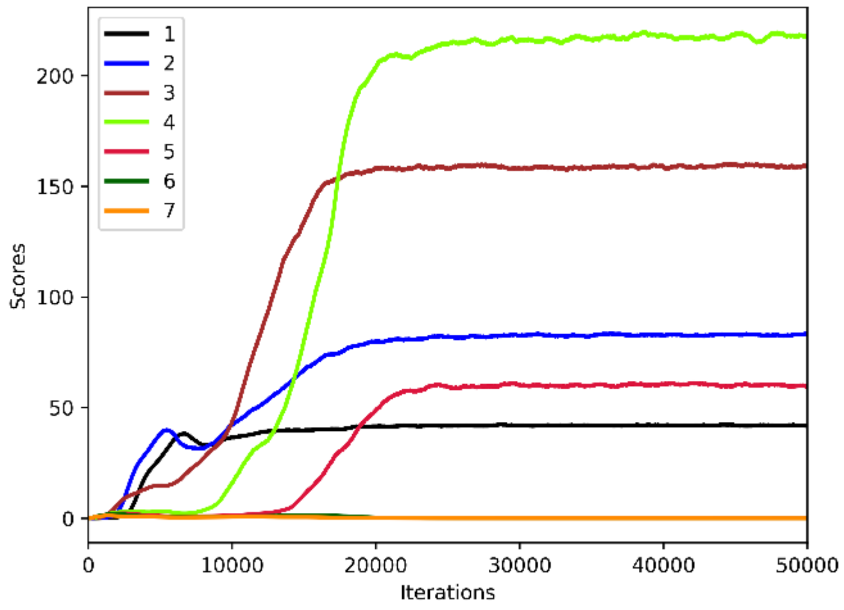


Figure 4.13: A change of the number of the last penalized states for 50,000 iterations and averaged the results of 100 random simulations.

the same setting except ϵ policy is set to true and learning rate $\eta = 0.7$ which is very similar value. The colored trending lines represent average scores of the last 1,000 games with discount factor appertain to the values in the legend on the left side of the figures.

4.3.4 Learning Rate

Figure 4.16 shows the difference in the influence of the parameter **learning rate** on the Q-learning algorithm efficiency. The results are not as straightforward as it is in the previous section. If the value is set to 1.0, the algorithm does not take into account the original value which could be well trained from the previous iterations and therefore the algorithm works with the most recent results only. If the value is too small, learning is very slow because the new value has a minimal influence.

Setting the value to 1.0, 0.99999 and 0.00001 shows the worst results. The reason is described above - we take into account the current reward too much or do not take it into account at all. As we can see, values between 0.1 and 0.9 represent better results, but learning with very low learning rate slow down learning very much and more iterations are needed. The values around 0.8 look like the most optimal values for learning rate.

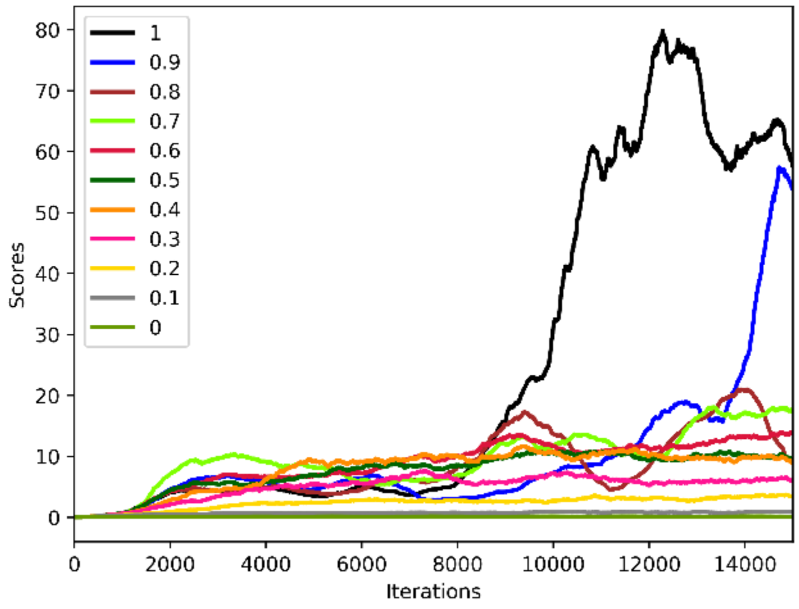


Figure 4.14: A change of discount factor on the same 15,000 games in 1 simulation.

In all of the simulations shown in Figure 4.16 discount factor is set to 1.0 as far as in the previous subsection it tends to be the optimal setting of this parameter, the reward r is set to 1 for alive states and to -1000 for the last three states, the rounding values are set as follows $r_X = r_Y = 5$ with $n = 1$ and the ϵ policy is set to true (the meaning of the parameters is described in Sections 2.1 and 3.1). The colored trending lines represent average scores of the last 1,000 games with learning rate appertain to the values in the legend on the left side of the figure.

4.3.5 Maximizing k-Future Rewards Policy

The **policy** for selecting actions is one of the key factors of the reinforcement learning algorithms. The classical description of the Q-learning algorithm and also SARSA only takes one next step into account. In Figure 4.18 one can see how taking more future steps into account could change learning time in term how many runs of the game are necessary to see improvement.

To understand the change of policy better, during picking the best action the algorithm tries to maximize not only the actual state values for all the actions but sum up Q-values of all possibilities in depth k and pick the best action with the greatest summed value. The complexity of picking a new action grows exponentially with respect to the number of future steps taken into account. If we take k future

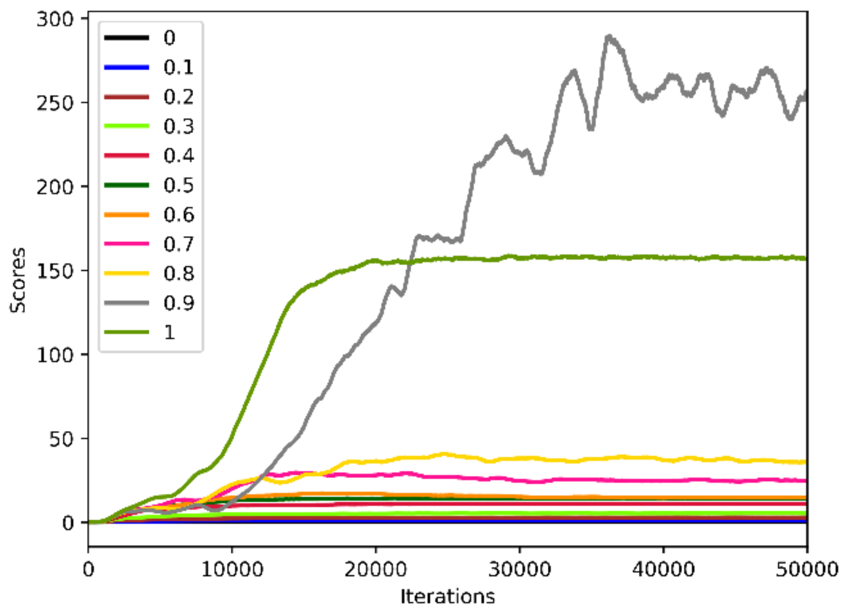


Figure 4.15: A change of discount factor for 50,000 iterations and averaged the results of 100 random simulations.

steps into account it is $\mathcal{O}(|A|^k)$, where A is the set of all possible actions - action space (in our case only a two elements set). Since the complexity of choosing the best action is very high the k values must be set "sensibly" to avoid slow agent responses. An example of such policy for depth $k = 3$ is shown in Figure 4.17. In the example we have these possibilities:

- $q_1 = Q(s, FLAP) + Q(s_1, FLAP) + Q(s_3, FLAP)$,
- $q_2 = Q(s, FLAP) + Q(s_1, FLAP) + Q(s_3, NOT\ FLAP)$,
- $q_3 = Q(s, FLAP) + Q(s_1, NOT\ FLAP) + Q(s_4, FLAP)$,
- $q_4 = Q(s, FLAP) + Q(s_1, NOT\ FLAP) + Q(s_4, NOT\ FLAP)$,
- $q_5 = Q(s, NOT\ FLAP) + Q(s_2, FLAP) + Q(s_5, FLAP)$,
- $q_6 = Q(s, NOT\ FLAP) + Q(s_2, FLAP) + Q(s_5, NOT\ FLAP)$,
- $q_7 = Q(s, NOT\ FLAP) + Q(s_2, NOT\ FLAP) + Q(s_6, FLAP)$,
- $q_8 = Q(s, NOT\ FLAP) + Q(s_2, NOT\ FLAP) + Q(s_6, NOT\ FLAP)$.

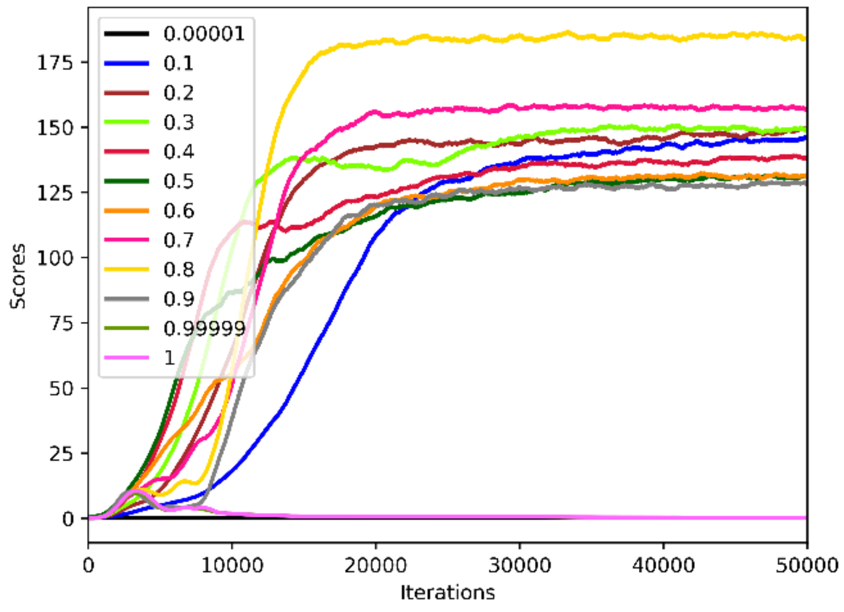


Figure 4.16: A change of learning rate for 50,000 iterations and averaged the results of 100 random simulations.

As far as $k = 3$ we have $2^k = 2^3 = 8$ possibilities. We pick the corresponding action in state s depending on $\max\{q_i : i \in \{1, \ldots, 8\}\}$.

Figure 4.18 shows that if we take only one next step into account, the algorithm Q-learning does not begin to improve in the first 1,000 iterations. However, if we increase k - the number of the next steps we take into account - we see that for $k = 4$ improvement can be seen in the first 1,000 iterations dramatically comparing to lover values.

In all of the simulations shown in this section, learning rate is set to 0.7, discount factor is set to 1.0, the reward $r = 1$ for alive states and $r = -1000$ for the three last states, the rounding values are set as follows $r_X = r_Y = 5$ with $n = 1$ with $n = 1$ and the ϵ policy is set to true (the meaning of the parameters is described in Sections 2.1 and 3.1). The colored trending lines represent average scores of the last 1,000 games with the parameter k appertains to the values in the legend on the left side of the figure.

4.3.6 Optimal Algorithm

Based on the results from the previous subsections we set the parameters to achieve as good results as possible with reasonable learning time. So we set the parameters

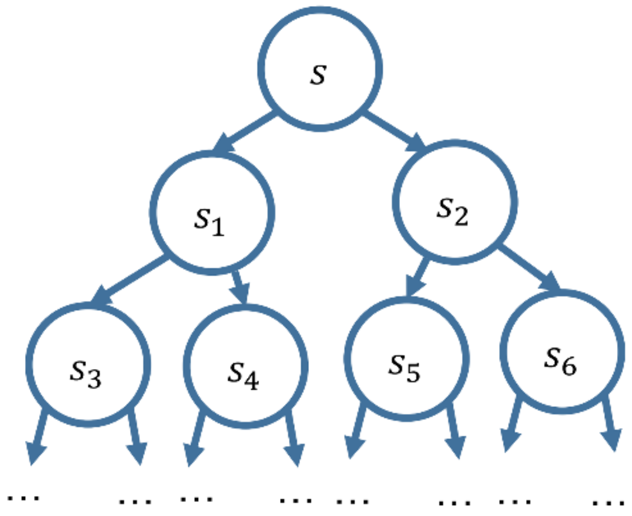


Figure 4.17: An example of picking an action for possibilities in depth 3.

for Q-learning as follows:

- learning rate $\eta = 0.8$,
- discount factor $\gamma = 1.0$,
- the ϵ policy is set to false,
- the rounding values described in the Section 3.1 are set to $r_X = r_Y = 5$ and $n = 1$,
- we take 4 next possible states into account - to make reasonable time complexity and algorithm efficiency (see Section 4.3.5), so $k = 4$,
- the reward $r = 1$ for alive states and $r = -1000$ for the three last states.

By setting the parameters as described above we can see results in Figure 4.19. The **best score** is **40,223** and the **average score** stabilised around the score **3,206.513** (if we consider the last 150 values in each iteration).

4.4 Deep Q-learning

To implement the algorithm described in Algorithm 2 we first need to decide what kind of neural network we want to use. Authors usually use convolutional neural networks when implementing the algorithm and use raw pixel information from the game screen to define state space [35, 1, 37]. To express velocity of the bird they also

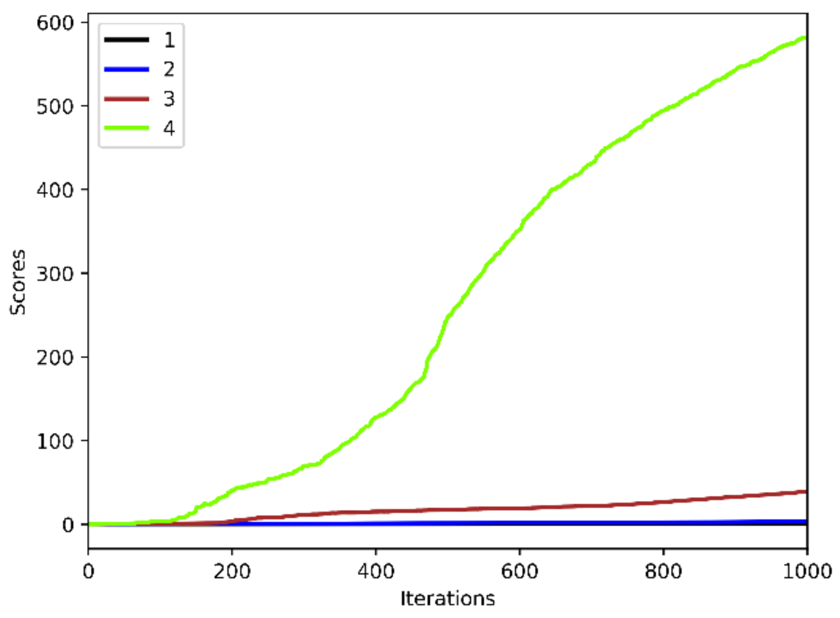


Figure 4.18: Changing the number of steps taken into account when selecting a new action (for 1,000 iterations and averaged the results of 25 random simulations).

sometimes need to use 3 or 4 consecutive game screens. The neural network we use is implemented by using the library **TensorFlow** [55].

In our case, we use extracted features from the game to define state space (with $n = 1$, as it is described in Section 3.1). In results provided in this section we use **feedforward neural network** with **2 hidden layers** with **600 and 200 neurons**. These values seemed to be the best in our case. We also tested more and fewer layers but the best results are achieved by using only two layers (we also tried 3 to 5 layers with more neurons, like 1000, or less, like 100). The input for the network is a triple consisting of three real values: horizontal and vertical distance of the bird to the next pipe and the velocity (as described in Section 3.1). The output of the network is an ordered pair of two real values from interval $[-1, 1]$ meaning future reward by choosing flap or not flap action. As the **activation function** three different activation function are tested: **sigmoid**, **hyperbolic tangent** (hereinafter **tanh**) and **rectified linear unit** (hereinafter **relu**). In our case we use **tanh**, but we also tested **relu** with unsatisfactory results. Figure 4.20 shows overview of these activation functions. The network is initialized with random weights and these are updated by using Adam algorithm (described in Subsection 2.2.2). Using weights with zero values at the beginning does not present good results. Learning rate is set to 0.01.

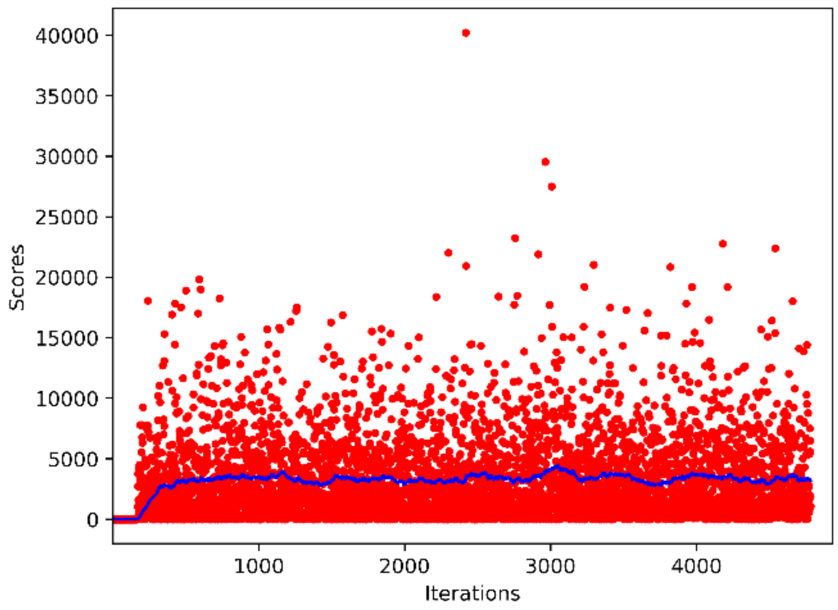


Figure 4.19: The best-achieved results using Q-learning (red dots represent score in individual games) with the average score 3,206.513 and the highest score 40,223.

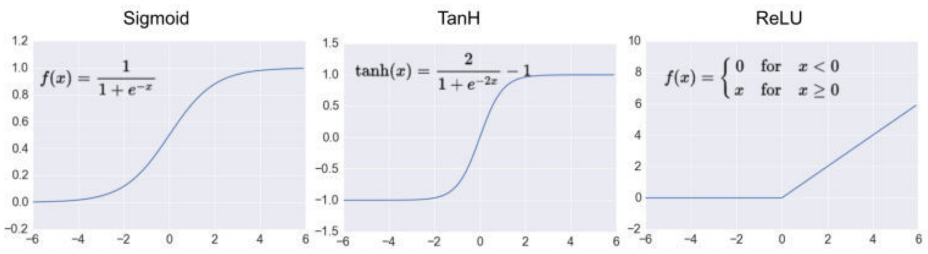


Figure 4.20: An overview of three activation functions: sigmoid, tanh and relu [4].

Figure 4.21 shows results achieved by using the Deep Q-learning algorithm and uses the Advanced Greedy Algorithm described in Algorithm 4 to obtain the first replay memory. We can see that the average score starts to increase more times but drop each time down again.

Figure 4.22 shows results achieved by using the Deep Q-learning algorithm and uses the best-trained Q-values presented in Figure 4.19 to train network at the beginning. This way we can obtain the **best score** which is extremely higher than the one achieved by using the Q-learning algorithm. It is **457,827**.

When we initialize the network by using Q-values generated by Q-learning we map for each state s decisions to 1, if the action would be taken or -1, if the action would not be taken by the Q-learning algorithm. So we have for each state $s \in S$ and both actions a_1 and a_2 training data in format $Q(s, a_1) = 1$ and $Q(s, a_2) = -1$ or

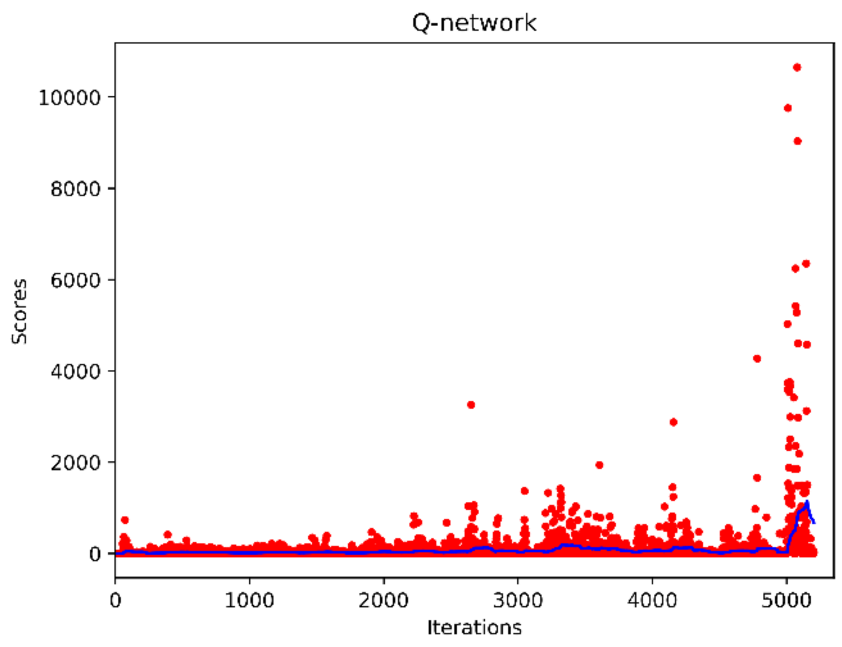


Figure 4.21: The best-achieved results by using the Deep Q-learning algorithm with replay memory obtained by using Algorithm 4.

$Q(s, a_1) = -1$ and $Q(s, a_2) = 1$. When training the model during initialization we use more optimizers like gradient descent, AdaGrad, RMSProp or Adam and train more separated networks. As far as Adam trains the best models we use them and decisions are made by all of them by summing predictions of all models and then picking the action to take by maximizing the values.

To update the network after initialization the replay data of the new game are stored as described in Deep Q-learning algorithm (Algorithm 2). Rewards are then generated similar way as during initialization - in state s the reward in 1, if the action was taken or -1, if the action was not taken. Only the last 30 states are ignored because we presume that some of them caused the death of the bird. Training is then done only by using a new data and discount factor is not used in this case.

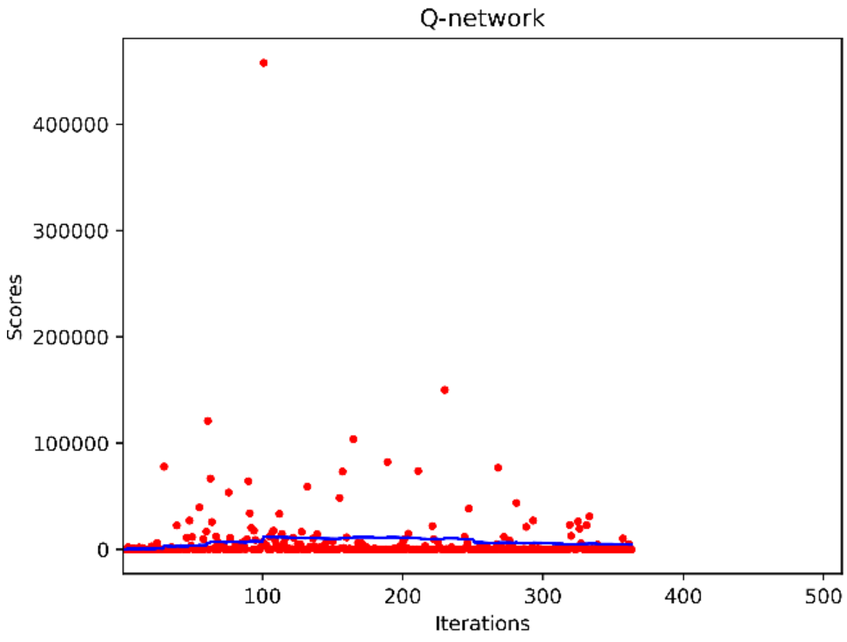


Figure 4.22: The best-achieved results by using the Deep Q-learning algorithm with replay memory obtained by using Algorithm 1 with the highest score 457,827.

Conclusion

The results described in the thesis show some properties of Q-learning and the role of neural networks in reinforcement learning through the Deep Q-learning algorithm. The experiments confirm some hypotheses about the setting parameters of the algorithms like learning rate, discount factor, rewards, the number of penalized states, etc. However, in order to achieve optimal results, Q-learning should run infinitely many times, which is practically impossible. Also, the pipes' positions are generated randomly and the algorithm must be run a sufficient number of times to be in all the states equally number of times. To reduce that fact we use the same generated pipes in games to make results more comparable or use multiple simulations with random pipes each time.

In Figure 4.23 one can see the comparison of two best-implemented instances of Q-learning and Deep Q-learning tested on the same 100 games where each game has at maximum 50,000 pipes generated (then the game stops). The **average score** is approximately **36,429.26 for Deep Q-learning** and **2,771.59 for Q-learning**. So Deep Q-learning plays significantly better then Q-learning. If we allow more pipes the difference could be even higher as far as the Deep Q-learning algorithm stops many times after passing all the pipes in an iteration. For more detailed comparison of the algorithm's instances check Appendix B where in Figures B.24-B.28 decisions of both algorithms are visualized for all vertical and horizontal distances and for all velocities.

The results show that we are able to have the best results with learning rate around 0.8 and discount factor around 1.0. This means that the importance of the old value is around 20% comparing to the importance of a new value which is 80% and importance of the future possible reward is very high during learning.

The introduced algorithms tested in the game Flappy Bird show that extracting features from images and using feedforward neural network instead convolutional can lead to significant results. The same approach has a potential in more real-world applications where raw pixels with lots of noise are used as an input.

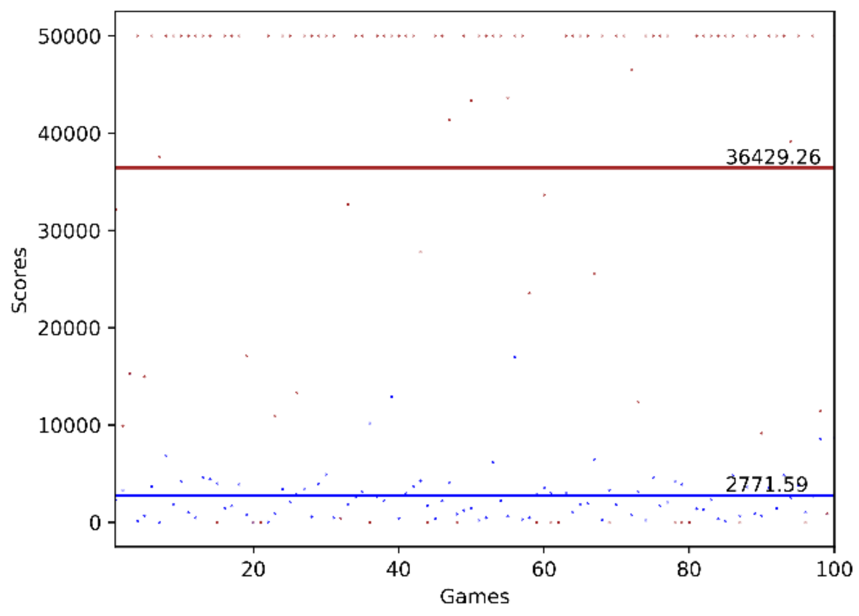


Figure 4.23: The comparison of results by using the Deep Q-learning algorithm (brown color) and Q-learning (blue color).

An interesting idea would also be to try dynamic changing of the parameters - as some articles state [38]. Testing results by using more consecutive pipes to define state space (using $n \geq 2$ as described in Section 3.1) or using completely different approach to define state space, could be done as a future work. For example, some articles define states also with an alive factor of the agent [38] or adding distance to the ground from the bird in the definition of the states. Some articles also define three types of reward [37]. For this particular game would be also interesting to do research whether there is a combination of pipes which is not possible to pass.

Many other algorithms that could be used to solve the problem (like QV(λ)-Learning, etc.) are described in many other articles as [56, 34]. For some of them, the apparent disadvantage is that it has too high time complexity since for some of the algorithms you need to update V-values (which are of the same size as all Q-values) in each update of a Q-value. So if we have more than 80,000 states then we must update at every step of each iteration all of them. But the authors promise a faster convergence to the optimal values. So the thesis has still a great potential for a future research.

Resumé

V tejto diplomovej práci sa venujeme algoritmom posilneného učenia a dané algoritmy a ich modifikácie testujeme v doméne hry Flappy Bird. Cieľom práce je preskúmať možnosti a už dosiahnuté výsledky v podobných doménach a problémoch a aplikovať dané metódy vo vlastnom prístupe, tieto postupy potom otestovať na doméne hry Flappy Bird a sledovať hlavné atribúty ako časová zložitosť, čas učenia, dosiahnuté najvyššie alebo priemerné skóre a pod. V práci tiež spomíname aj mnohé možné aplikácie takto navrhnutých algoritmov z referenciami na konkrétne zdroje.

V kapitole 1 sa venujeme aktuálnemu výskumu v posilnenom učení. V sekciách tejto kapitoly ukazujeme výhody a použitie tzv. replay memory a ďalej sa venujeme metódam posilneného učenia v počítačových hrách ako Backgammon, Go, Texas Hold'em, Doom, Atari a Flappy Bird. Zameriavame sa na rôzne metódy a základné myšlienky použité na tvorbu umelej inteligencie na hranie konkrétnych hier.

V kapitole 2 sa venujeme samotnému posilnenému učeniu. V úvode vysvetľujeme základné pojmy a neskôr sa zameriavame na konkrétne algoritmy, ktoré neskôr v práci aj implementujeme a testujeme. Pri algoritmoch, kde sú použité neurónové siete sa vo zvlášť podsekcii venujeme aj možnostiam ako aktualizovať váhy neurónovej siete.

V kapitole 3 sa už venujeme samotnej hre Flappy Bird. Prv popisujeme detailne ako sa hra hrá, ako funguje, ako sa menia a prekresľujú konfigurácie hry. Ďalej sa venujeme, pre lepšie pochopenie, aj konkrétnym konštantám a vzdialenostiam v hre. Po základnom oboznámení sa s hrou nasleduje sekcia s definovaním stavového priestoru práve pre hru Flappy Bird. Pojednáva sa o rôznych možnostiach definovania stavov a na konci sa definuje stavový priestor pre hru Flappy Bird v definícii 3.2 a je uvedený aj konkrétny príklad ako funguje prepočet konfigurácie hry na konkrétny stav zo stavového priestoru.

V kapitole 4 sa už venujeme konkrétnym výsledkom, ktoré boli dosiahnuté. Prv sa venujeme dvom pažravým algoritmom, ktoré sú použité na porovnanie, ale aj v iných algoritmoch a inicializáciu dát a pod. Ďalej sa venujeme Q-learning algoritmu a testujeme rôzne nastavenia politiky výberu akcie, učiaceho a diskontného faktora, odmeny a penalizácie a sekciu ukončujeme jednou z najlepších inštancií implementácie algoritmu. Tento algoritmus využívame v ďalšej sekcii danej kapitoly na inicializáciu neurónovej siete pre algoritmus Deep Q-learning. Táto kombinácia sa zdá byť unikátna spoločne s použitím doprednej siete namiesto konvolučnej (ako je to aj v článku, kde je predstavený Deep Q-learning [1]).

V závere práce pozitívne hodnotíme zmenu politiky výberu akcie, ktorá sa pozerá a maximalizuje hodnoty všetkých možností až do hĺbky k, použitie doprednej siete v algoritme Deep Q-learning namiesto konvolučnej a použitie Q-hodnôt naučených Q-learning algoritmom na inicializáciu neurónovej siete použitej v algoritme Deep Q-learning. Porovnanie najlepších algoritmov je možné pozrieť na obrázku 4.23. Ďalej v závere spomíname možnosti ďalšieho výskumu alebo nápadov, ktoré by mohli byť v ďalších prácach analyzované a otestované.

V prílohách uvádzame odkaz na verejný GitHub repozitár, kde sú zverejnené všetky zdrojové kódy súvisiace s prácou spoločne s komentármi (príloha A) a tiež aj vizualizácie rozhodnutí najlepších inštancií Q-learning a Deep Q-learning algoritmu (príloha B).

Bibliography

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 1998.

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

[4] KDnuggets. Activation functions. `https://www.kdnuggets.com/wp-content/uploads/activation.png`, 2017. Accessed: 2017-10-10.

[5] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[6] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[7] Steve Young, Milica Gašić, Blaise Thomson, and Jason D Williams. Pomdp-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5):1160–1179, 2013.

[8] Xuijun Li, Yun-Nung Chen, Lihong Li, and Jianfeng Gao. End-to-end task-completion neural dialogue systems. *arXiv preprint arXiv:1703.01008*, 2017.

[9] Bhuwan Dhingra, Lihong Li, Xiujun Li, Jianfeng Gao, Yun-Nung Chen, Faisal Ahmed, and Li Deng. End-to-end reinforcement learning of dialogue agents for information access. *arXiv preprint arXiv:1609.00777*, 2016.

[10] Pei-Hao Su, Milica Gasic, Nikola Mrksic, Lina Rojas-Barahona, Stefan Ultes, David Vandyke, Tsung-Hsien Wen, and Steve Young. On-line active reward

learning for policy optimisation in spoken dialogue systems. *arXiv preprint arXiv:1605.07669*, 2016.

[11] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in neural information processing systems*, pages 2204–2212, 2014.

[12] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[13] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[14] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[15] Georgios Theocharous, Philip S Thomas, and Mohammad Ghavamzadeh. Personalized ad recommendation systems for life-time value optimization with guarantees. In *IJCAI*, pages 1806–1812, 2015.

[16] Yue Deng, Feng Bao, Youyong Kong, Zhiquan Ren, and Qionghai Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems*, 28(3):653–664, 2017.

[17] Yair Goldberg and Michael R Kosorok. Q-learning with censored data. *Annals of statistics*, 40(1):529, 2012.

[18] Zheng Wen, Daniel O'Neill, and Hamid Maei. Optimal demand response using device-based reinforcement learning. *IEEE Transactions on Smart Grid*, 6(5):2312–2324, 2015.

[19] Mark Hammond. Deep reinforcement learning in the enterprise: Bridging the gap from games to industry. Artificial Intelligence Conference, San Francisco, 2017.

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

[21] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[22] Richard Stuart Sutton. Temporal credit assignment in reinforcement learning. 1984.

[23] Andrew Gehret Barto, Richard S Sutton, and Christopher JCH Watkins. Learning and sequential decision making. 1989.

[24] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, 1989.

[25] Gerald Tesauro. Td-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer, 1995.

[26] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[27] David Silver, Richard S Sutton, and Martin Müller. Reinforcement learning of local shape in the game of go. In *IJCAI*, volume 7, pages 1053–1058, 2007.

[28] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[29] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[30] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. 2016.

[31] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland,

Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[33] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[34] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[35] Naveen Appiah and Sagar Vare. Playing flappybird with deep reinforcement learning. 2016.

[36] Timo Wilken. Flappy bird (using pygame). https://github.com/TimoWilken/flappy-bird-pygame, 2014.

[37] Kevin Chen. Deep reinforcement learning for flappy bird. 2015.

[38] Moritz Ebeling-Rump, Manfred Kao, and Zachary Hervieux-Moore. Applying q-learning to flappy bird. *Department Of Mathematics And Statistics, Queen's University*, 2016.

[39] Sarvagya Vaish. Flappy bird rl. https://sarvagyavaish.github.io/FlappyBirdRL/, 2014.

[40] Cihan Ceyhan. Flappy bird bot using reinforcement learning. https://github.com/chncyhn/flappybird-qlearning-bot, 2017.

[41] Yi Shu, Ludong Sun, Miao Yan, and Zhijie Zhu. Obstacles avoidance with machine learning control methods in flappy birds setting. *Univ. of Stanford, CS229 Machine Learning Final Projects Stanford University*, 2014.

[42] Matthew Piper. *How to Beat Flappy Bird: A Mixed-Integer Model Predictive Control Approach.* PhD thesis, The University of Texas at San Antonio, 2017.

[43] Ihor Menshykov. Immortal flappy bird. https://github.com/ibmua/immortal-flappy, 2017.

[44] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[45] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[46] Francisco S Melo. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4, 2001.

[47] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.

[48] Mohamad H Hassoun. *Fundamentals of artificial neural networks*. MIT press, 1995.

[49] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[50] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Technical Report*, 2012.

[51] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[52] Mike Bertha. Everything you need to know about your new favorite cell phone game, 'flappy bird'. http://www.philly.com/philly/blogs/trending/Flappy-Bird-app-game-iPhone-Android-obsessed-cheats-impossible-Ironpants.html, 2014. Accessed: 2017-10-10.

[53] Steve Kovach. Flappy bird is officially dead. http://www.businessinsider.com/flappy-bird-pulled-from-app-stores-2014-2, 2014. Accessed: 2017-10-10.

[54] Sourabh Verma. Flappy bird. https://github.com/sourabhv/FlapPyBird, 2017.

[55] TensorFlow Library. https://www.tensorflow.org/.

[56] Marco A Wiering. Qv (lambda)-learning: A new on-policy reinforcement learning algrithm. In *Proceedings of the 7th European Workshop on Reinforcement Learning*, pages 17–18, 2005.

Appendices

Appendix A contains link to the source codes used in this thesis and Appendix B contains visualizations of the learning functions.

Appendix A

Source Codes

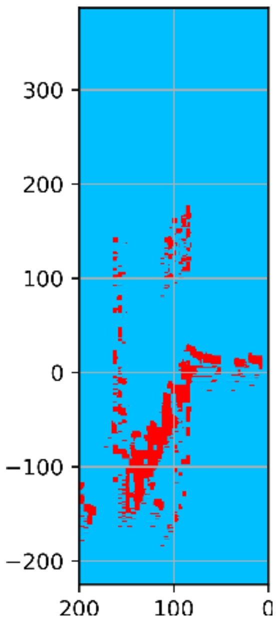
All the source codes generated to create this thesis are available in the public GitHub repository `https://github.com/martinglova/FlappyBirdBotsAI`. The code is commented and contains the `README.md` file with a description.

Appendix B

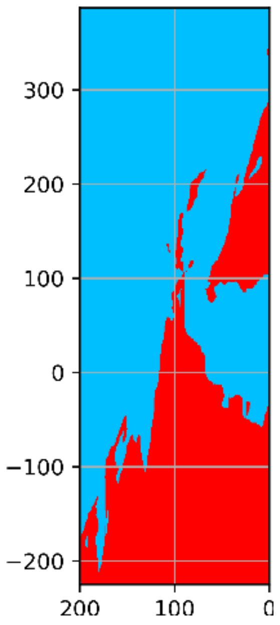
Visualizations of Learning Functions

Visualizations in this appendix show how the Q-learning and Deep Q-learning bots make decisions in states from state space $\{\langle x, y, v\rangle : x \in [0, 200], y \in [-225, 387], v \in [-9, 10]\}$ as defined in Section 3.1. In all the figures in this appendix Q-learning is denoted only as QL and Deep Q-learning as DQL. Subcaption format is as folows "a x A", where $a \in \{\uparrow, \downarrow, \epsilon\}$ where $\uparrow$ means velocity upwards, $\downarrow$ means velocity downwards and ϵ means empty string, $x \in [0, 10]$ meaning the speed of the bird (upwards or downwards) and $A \in \{\mathrm{QL}, \mathrm{DQL}\}$ denotes used algorithm to create the visualization.

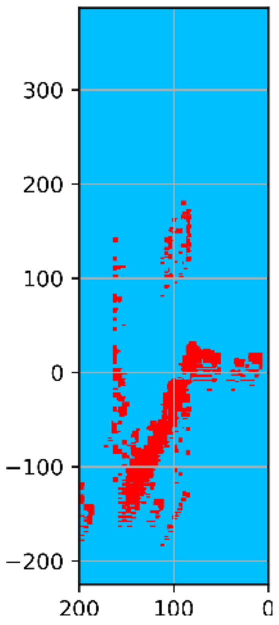
The red color in visualizations means that the bird would flap and the blue color means that the bird would not flap in the state.



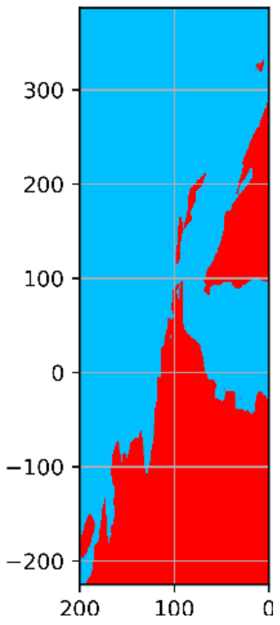
(a) ↑ 9 QL.



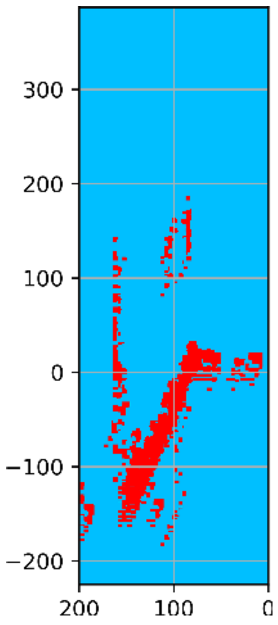
(b) ↑ 9 DQL.



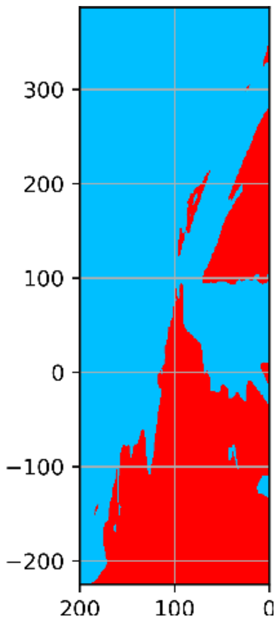
(c) ↑ 8 QL.



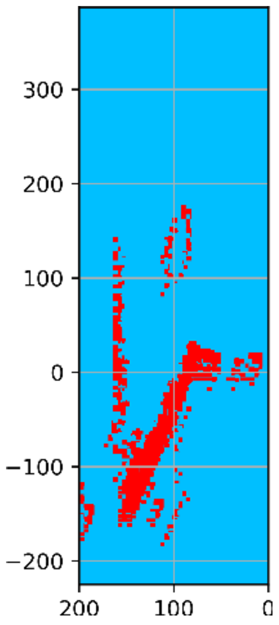
(d) ↑ 8 DQL.



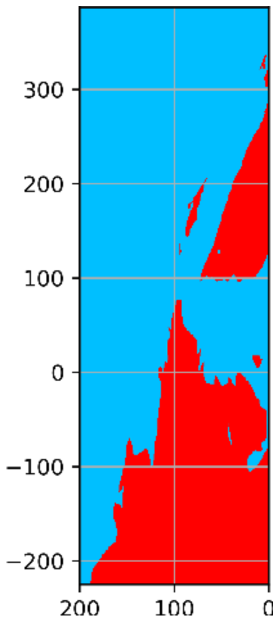
(e) ↑ 7 QL.



(f) ↑ 7 DQL.

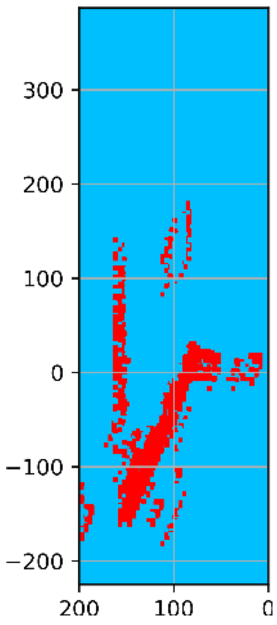


(g) ↑ 6 QL.

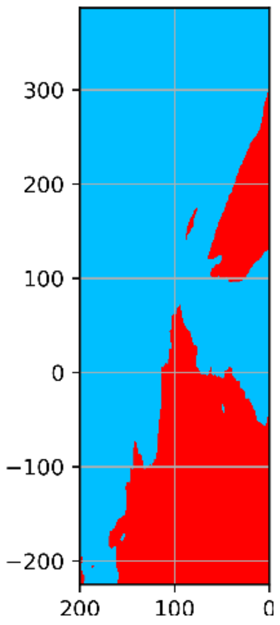


(h) ↑ 6 DQL.

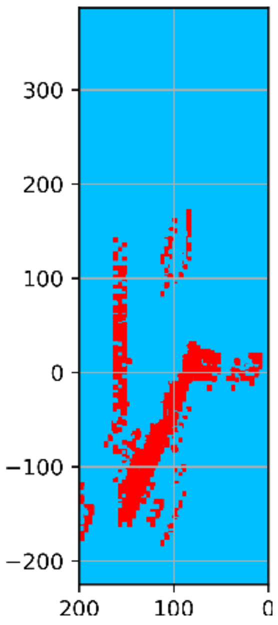
Figure B.24: Visualization of the learning functions for velocities ↑ 9, ↑ 8, ↑ 7 and ↑ 6 for QL and DQL.



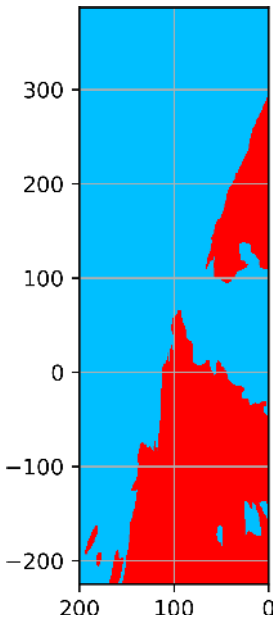
(a) ↑ 5 QL.



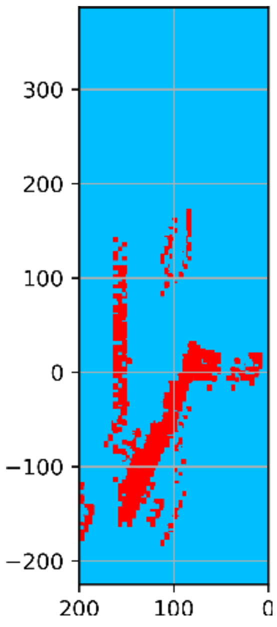
(b) ↑ 5 DQL.



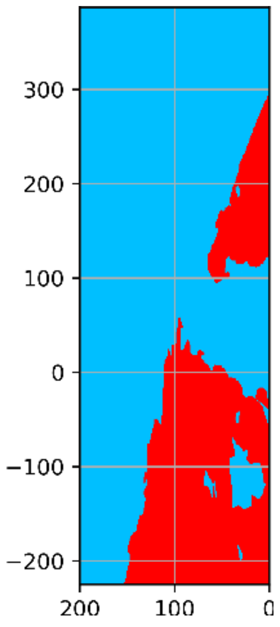
(c) ↑ 4 QL.



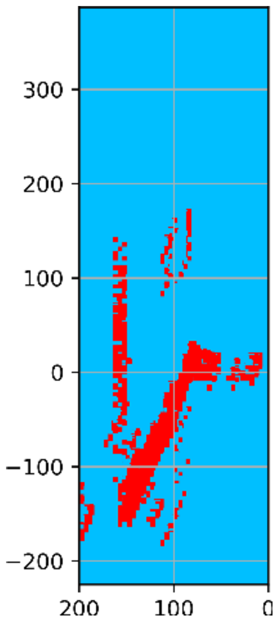
(d) ↑ 4 DQL.



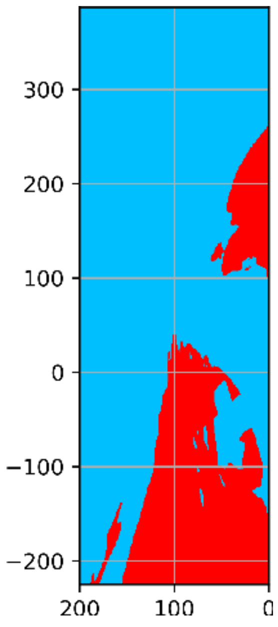
(e) ↑ 3 QL.



(f) ↑ 3 DQL.

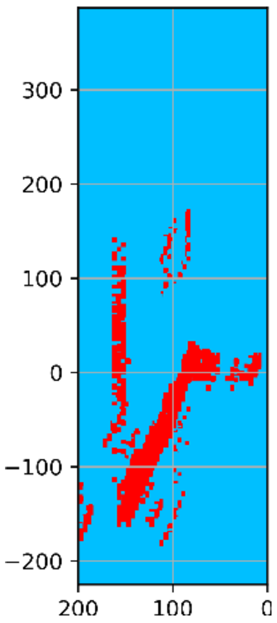


(g) ↑ 2 QL.

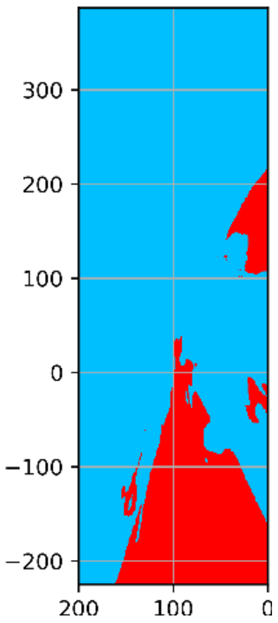


(h) ↑ 2 DQL.

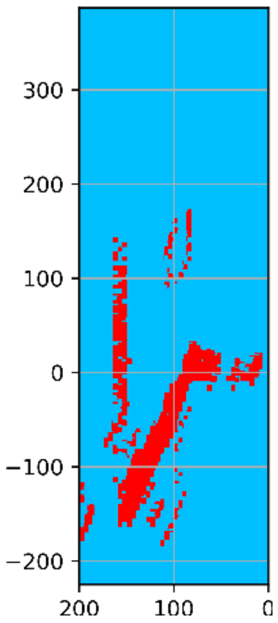
Figure B.25: Visualization of the learning functions for velocities ↑ 5, ↑ 4, ↑ 3 and ↑ 2 for QL and DQL.



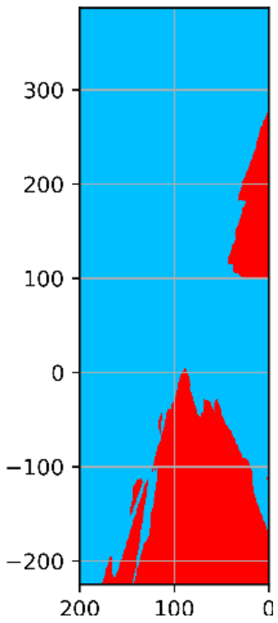
(a) ↑ 1 QL.



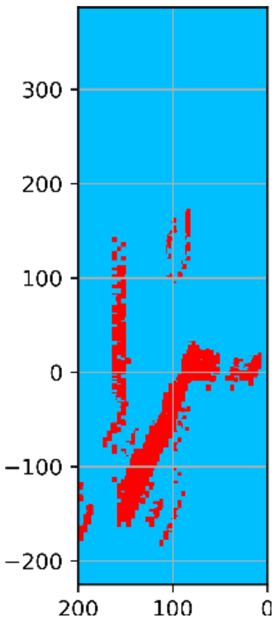
(b) ↑ 1 DQL.



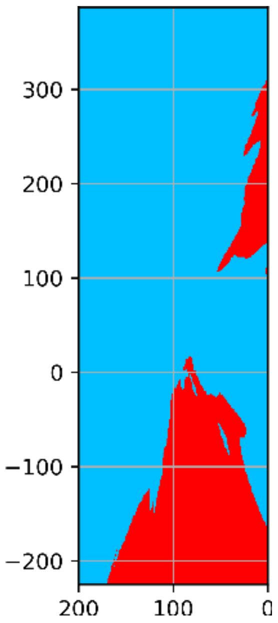
(c) 0 QL.



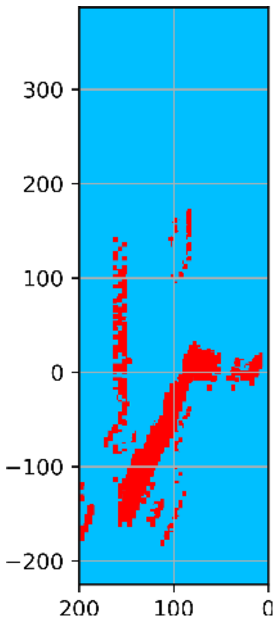
(d) 0 DQL.



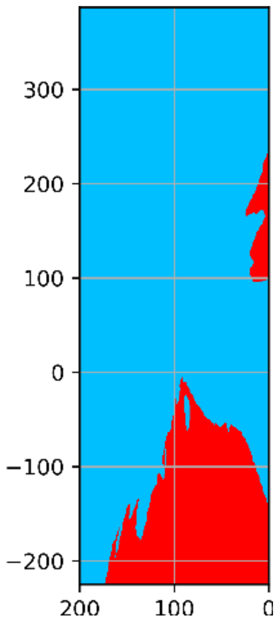
(e) ↓ 1 QL.



(f) ↓ 1 DQL.

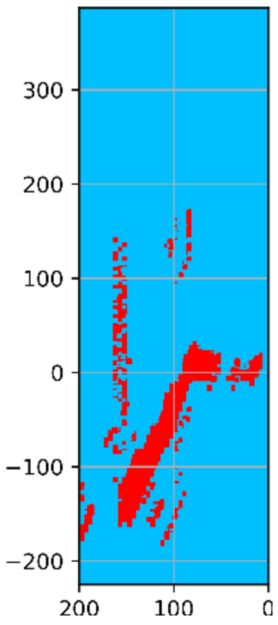


(g) ↓ 2 QL.

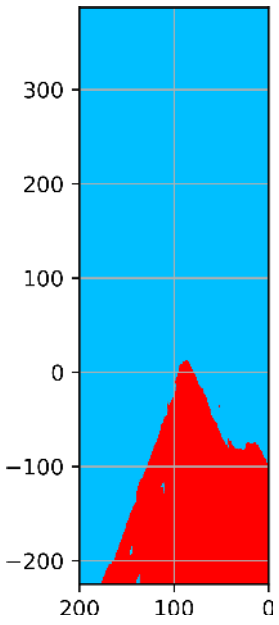


(h) ↓ 2 DQL.

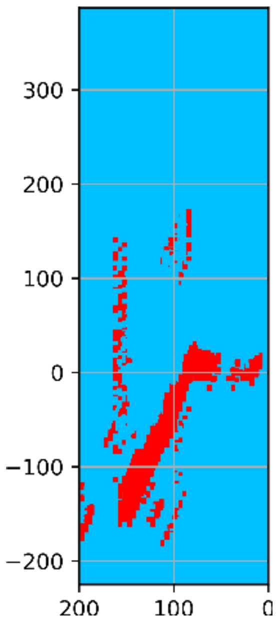
Figure B.26: Visualization of the learning functions for velocities ↑ 1, 0, ↓ 1 and ↓ 2 for QL and DQL.



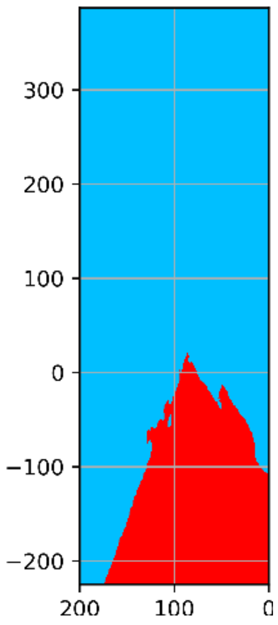
(a) ↓ 3 QL.



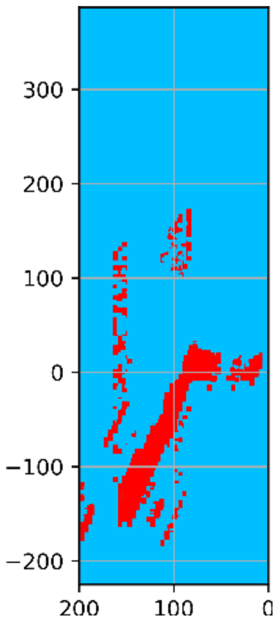
(b) ↓ 3 DQL.



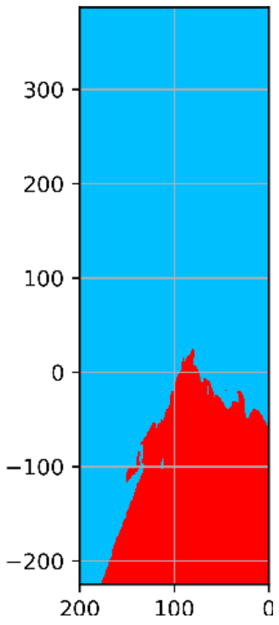
(c) ↓ 4 QL.



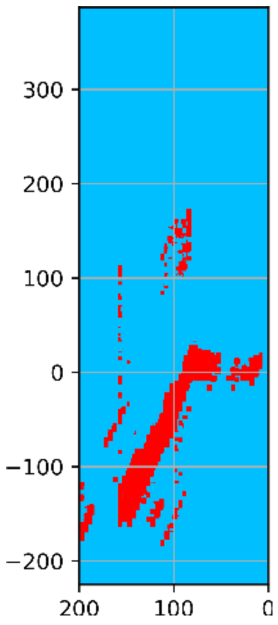
(d) ↓ 4 DQL.



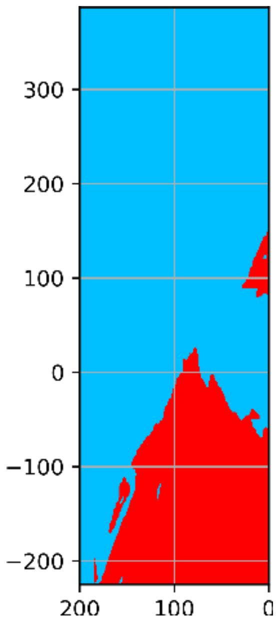
(e) ↓ 5 QL.



(f) ↓ 5 DQL.

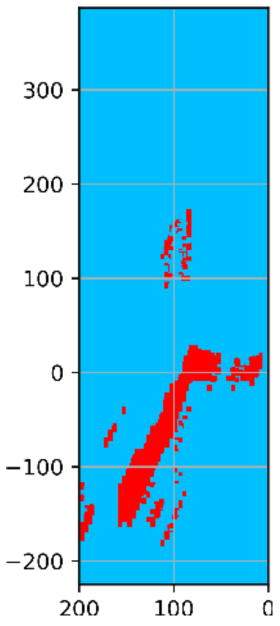


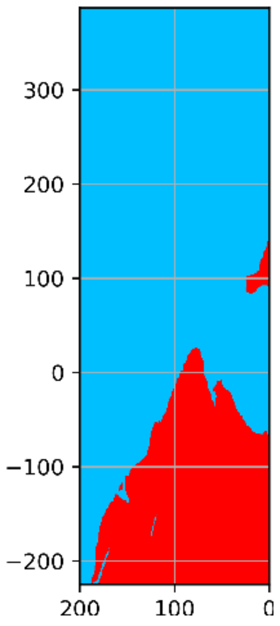
(g) ↓ 6 QL.

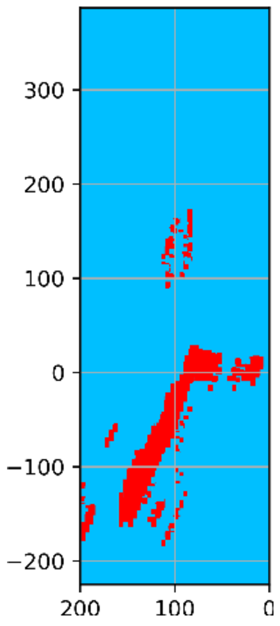


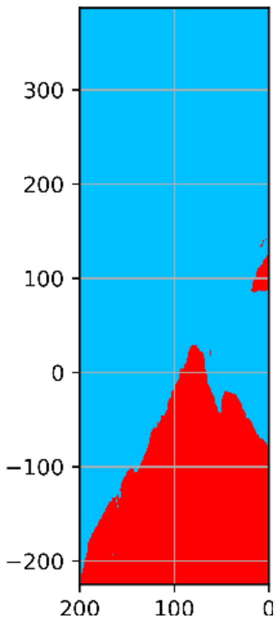
(h) ↓ 6 DQL.

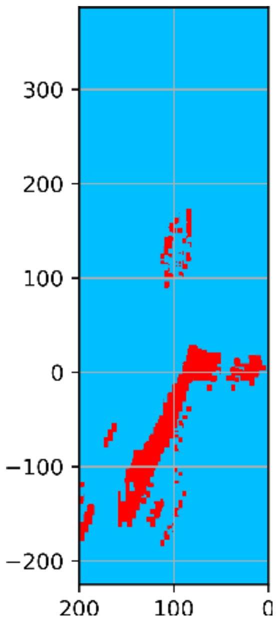
Figure B.27: Visualization of the learning functions for velocities ↓ 3, ↓ 4, ↓ 5 and ↓ 6 for QL and DQL.

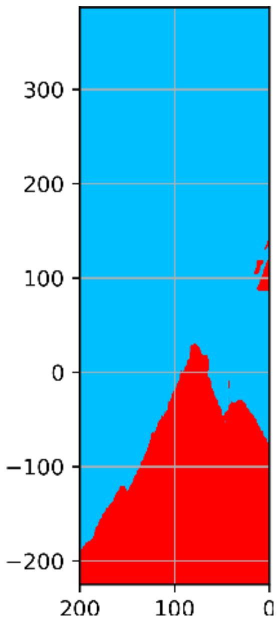

(a) ↓ 7 QL.

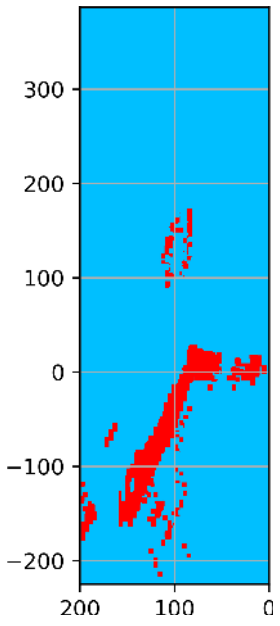

(b) ↓ 7 DQL.


(c) ↓ 8 QL.


(d) ↓ 8 DQL.


(e) ↓ 9 QL.


(f) ↓ 9 DQL.


(g) ↓ 10 QL.

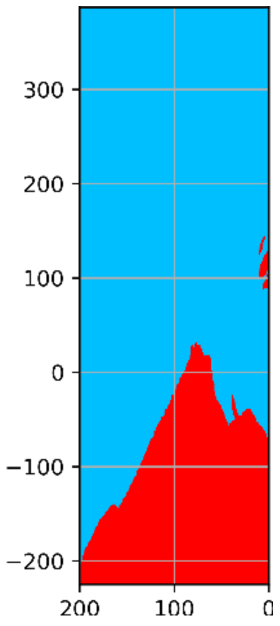

(h) ↓ 10 DQL.

Figure B.28: Visualization of the learning functions for velocities ↓ 7, ↓ 8, ↓ 9 and ↓ 10 for QL and DQL.