Fachhochschule
Salzburg University
of Applied Sciences

Contextual Reinforcement Learning in Computer Games

BACHELOR THESIS 2

Student	Daniel Satanik, 1110601028
Supervisor	Wolfgang Litzlbauer, MSc

Salzburg, 25 August 2014

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst, und keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe. Weiters versichere ich hiermit, dass ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

Datum Unterschrift

Kurzfassung

Vor- und Zuname:	Daniel SATANIK
Institution:	FH Salzburg
Studiengang:	Bachelor MultiMediaTechnology
Titel der Bachelorarbeit:	Contextual Reinforcement Learning in Computer Games
Begutachter:	Wolfgang Litzlbauer, MSc

Viele wissenschaftliche Artikel handeln von der Effizienz von maschinellen Lernalgorithmen ohne dabei zu behandeln, wie diese Effizienz erreicht wurde. Komplexere Algorithmen sind schwer zu parametrisieren und bei falscher Konfiguration könnte der Lernprozess in die Länge gezogen werden oder sogar zu falschen Ergebnissen führen.
Es ist ebenfalls wichtig zu entscheiden ob man von Grund auf ein Verhalten erlernen lässt oder auf bestehenden Verhaltensregeln aufbaut. Bei Lernvorgängen, die nicht auf vorgefertigten Regeln basieren, kann das Verhalten stärker optimiert werden. Das Verbessern von bestehendem Verhalten braucht meist aber viel weniger Zeit und es können bestimmte Charakteristiken beibehalten werden.
Diese Bachelor Arbeit befasst sich mit dem Thema des Bestärkenden Lernens, einer Methode aus der Familie der Optimierungs-Algorithmen, bei denen in bestimmten Zuständen bestimmte Aktionen ausgeführt werden. Im Lernprozess werden diese Aktionen aufgrund von Erfahrung, durch geeignetere ersetzt. Dabei erhält der Agent stets eine Rückmeldung von der Welt, wie optimal das Ausführen der Aktion zur Erreichung eines bestimmten Ziels war.
In dieser Arbeit werden daher Künstliche Intelligenz, Bestärkendes Lernen und der spezielle Algorithmus Speedy Q-Learning vorgestellt. Des Weiteren wird auf die Wichtigkeit von sorgfältigem Parametrisieren der Algorithmen hingewiesen. Zum Schluss wird ein Experiment vorgestellt, bei dem verschiedene Implementierungen des vorher erwähnten Algorithmus Speedy Q-Learning verglichen werden, indem eine existierende Künstliche Intelligenz des Spiels Blobby Volley 2 verwendet und angepasst wird.

Schlagwörter: Artificial Intelligence, Contextual Reinforcement Learning, Q-Learning, Speedy Q-Learning, Balance, Blobby Volley 2

Abstract

Many scientific papers place special emphasis on the efficiency of Machine Learning Algorithms instead of pointing out how efficiency is achieved. More complex algorithms are harder to parameterize and balance. By inappropriate configuration the learning process may be prolonged in terms of time or yield wrong results.
Moreover, it is important to decide if pure learning or contextual learning is needed. While pure learning may result in more highly optimized behavior, contextual learning takes considerably less time to optimize, by building on a sufficient minimal set of behavioral rules and can keep some interesting or even intended characteristics of its predecessor.
This bachelor thesis discusses Contextual Reinforcement Learning as a family of optimization algorithms, using state-action mappings for evolution and changing the actions taken in specific states based on experience by considering feedback from the world.
Therefore Artificial Intelligence, Reinforcement Learning and the specific algorithm Speedy Q-Learning are explained. In addition, the importance of carefully balancing learning algorithms is emphasized. Afterwards an experiment is performed comparing different implementations of aforementioned Speedy Q-Learning by adapting an existing Artificial Intelligence of the game Blobby Volley 2.

Keywords: Artificial Intelligence, Contextual Reinforcement Learning, Q-Learning, Speedy Q-Learning, Balance, Blobby Volley 2

Table of Contents

Eidesstattliche Erklärung i

Kurzfassung ii

Abstract iii

List of Abbreviations 1

1 Introduction 2
- 1.1 Relevance 2
- 1.2 Motivation 2
- 1.3 Research Questions and Hypotheses 3
- 1.4 Structure of the Thesis 3
  - 1.4.1 Theoretical Part 3
  - 1.4.2 Practical Part 3

2 Part 1 - Theory 4
- 2.1 Artificial Intelligence 4
  - 2.1.1 Characteristics of Game AI 4
  - 2.1.2 Model of Game AI 5
- 2.2 Reinforcement Learning 7
  - 2.2.1 Markov Decision Processes 7
  - 2.2.2 Contextual Reinforcement Learning 8
  - 2.2.3 Q-Learning 8
  - 2.2.4 Speedy Q-Learning 10
- 2.3 Balancing of Algorithms 11
- 2.4 State-of-the-art Reinforcement Learning 12
  - 2.4.1 Efficient Use of Reinforcement Learning in a Computer Game 12
  - 2.4.2 Flappy Bird RL 12
  - 2.4.3 Multi-agent reinforcement learning 13

3 Part 2 - Experiment 16
- 3.1 Blobby Volley 2 16
- 3.2 Blobby Volley 2 Agent - Reduced 17
- 3.3 Tested Agents 19
  - 3.3.1 Pure Random SQL 19
  - 3.3.2 Deferred Random SQL 21
  - 3.3.3 Standard SQL 21
- 3.4 Test Setup 23
- 3.5 Results 24
- 3.6 Discussion 27

4 Conclusion 28

References **29**

List of Figures **30**

List of Tables **30**

List of Equations **30**

List of Algorithms **30**

List of Abbreviations

AI Artificial Intelligence . . . 2
ANN Artificial Neural-Network . . . 2
CRL Contextual Reinforcement Learning . . . 8
EA Evolutionary Algorithm . . . 4
MDP Markov Decision Process . . . 7
NPC Non-Playing Character . . . 5
QL Q-Learning . . . 2
RLA Reinforcement Learning Algorithm . . . 2
RL Reinforcement Learning . . . 2
SCA Soft Computing Algorithm . . . 11
SQL Speedy Q-Learning . . . 3

1 Introduction

1.1 Relevance

Many computer games nowadays, use heavy Artificial Intelligence (AI) to make them more challenging. The decision making of agents in such games is often manually constructed and is repeatedly tailored after many user tests, to avoid both boredom and frustration and provide adequate challenge.
An alternative approach to a manual setup is to use Machine Learning Algorithms. Those are usually integrated in an abstract implementation of decision making, like a state machine. The different techniques can then adapt the states and connections (i.e. actions) to change the behavior of the agent.
Many scientific papers concentrate only on the efficiency of Machine Learning Algorithms but forget to mention how the algorithmic parameters are determined. The algorithms themselves are very powerful and produce good results, but when parameterized improperly the learning process may be prolonged or even lead to unintended and unexpected behavior (i.e. overtrained).
Moreover, Machine Learning is always an option to optimize applications. There is, however, a difference between learning completely new and learning to improve existing behavior. Most papers focus on the former, which takes considerably more time but may result in the most optimized behavior, as the agent can learn the best method to address the problem. In contrast, using existing code with some flaws, to improve, could also result in a highly optimized application but uses far less time and lets the improved application retain some interesting or even intended characteristics of its predecessor.

1.2 Motivation

To emphasize the importance of careful balancing process for complex algorithms and present the difference between pure learning and contextual learning, this bachelor thesis will discuss Contextual Reinforcement Learning (RL) as a family of optimization algorithms, using state-action mapping for evolution. The algorithms will find the most appropriate actions for states based on experience. After the action has been performed a reward is received. When resulting in good feedback the action gets a higher probability of being chosen from within the corresponding state by future decisions. Otherwise the probability drops.
States and actions depend highly on the application's characteristics. Finding the best abstraction of situations and keeping actions simple enough is the art of Reinforcement Learning Algorithm (RLA). Not only states and actions influence the effectiveness and efficiency of the learning-rate, different algorithms differentiate vastly in complexity. While Q-Learning (QL) depends on just two parameters, other algorithms like Evolutionary Algorithms and Artificial Neural-Networks (ANNs), try to be very flexible to approximate natural processes like evolution and neural systems. Hence, they have many parameters.

1.3 Research Questions and Hypotheses

- What are state-of-the-art RLAs in Computer Games?
    1. In modern games only pure RLAs are used based on a discretized state space derived from the environment.
    2. In modern games RLAs are used based on the context, meaning existing agents.
- What is the performance change (difference in scores) in Blobby Volley 2, applying Speedy Q-Learning (SQL), in the context of an already established agent, which is shipped with the game, compared to the original agent and one using pure SQL?
    1. The performance is increasing regardless of the type of SQL.
    2. The performance is increasing using contextual SQL.
    3. The performance is increasing using pure SQL.

1.4 Structure of the Thesis

1.4.1 Theoretical Part

The theoretical part of the bachelor thesis covers all theoretical concepts necessary to understand how the algorithm SQL works. Therefore AI-Systems are first presented, followed by RL and Standard QL. Thereafter the importance of careful balancing process is discussed.

1.4.2 Practical Part

For the practical part the game Blobby Volley 2 will be the basis for an experiment. Blobby Volley 2 consists of different AI- and Rule-Set-Scripts, which can be freely adapted. For the experiment the agent *Reduced* is chosen as the base for the contextual SQL approach and the competitor of all the tested scripts. The experiment shall answer the research question by either of the hypotheses proposed. The results of the experiment shows then which are good parameters for specific contexts for the SQL algorithm.

2 Part 1 - Theory

This section concentrates on familiarizing the reader with the theory of RL and one algorithm which is then used in the experiment. To understand how all the parts of the game fit together, first fundamental terms, which are used throughout the whole thesis are described. Starting with the game engine part in which aforementioned RLAs are used and following all relevant architecture models from abstract to a specific algorithm.

2.1 Artificial Intelligence

An game engine commonly consists of modules. The reason for this is simple: modules have clear interfaces and can be replaced easily by providing interfaces with the same signature. Hence, updates and upgrades do not affect other parts of the engine.

AI is no exception in that. One model for Game AI is presented in section 2.1.2. But before that an important question has to be answered: *What is Game AI and what are its characteristics?*
The next section is going to answer exactly that question.

2.1.1 Characteristics of Game AI

AI is hard to define. Nonetheless there is a quote by Millington and Funge (2009) getting to the heart of it, with "[...]making computers able to perform the thinking tasks that humans and animals are capable of.[...]" and further "As games developers, we are primarily interested in only the engineering side: building algorithms that make game characters appear human or animal-like.".

To conclude, AI is an abstraction of natural processes. It should feel real and challenging like living creatures or even humans. There are even algorithms largely approximating biological processes like ANN or Evolutionary Algorithm (EA). But both come with limitations, as there are not enough resources to let them run at the same level nature itself does. The human nervous system is made up of approximately 10^{11} neurons. Humans can learn in seconds if the task is simple or in hours if it is a little bit harder. ANNs need so much computational resources, that with only a hundred neurons it needs days or even weeks to learn the most simple tasks. And species are made up of thousands and millions of individuals with even more genes. An EA cannot keep up with these numbers.

This does not mean that those algorithms should not be used, if anything, they can be used effectively and efficiently - under prerequisite that they are properly configured. By using the appropriate parameters and insert the appropriate algorithm-snippets in the variable sections, either of them will yield optimal performance to solve the given problem.
More about the configuration of algorithms is discussed in section 2.3.

Each of the two previously mentioned algorithms, have different characteristics. There are two types of algorithm characteristics, *mutable* and *immutable* ones. This does not refer to the code, but to the behavior on the surface.
Immutable algorithms are, like texts engraved in stone. Each and every behavior is repeated on the same conditions. Every time the AI finds itself in the same situation, it

will perform the same action.
Mutable algorithms, however, adapt their behavior to some conditions to improve their effectiveness or efficiency.

2.1.2 Model of Game AI

As already mentioned, this section gives a short overview of a specific model of Game AI of the book Artificial Intelligence for Games, Second Edition by Millington and Funge (2009, p.34). The structure displayed in figure 1 is sometimes referred to as Core AI:

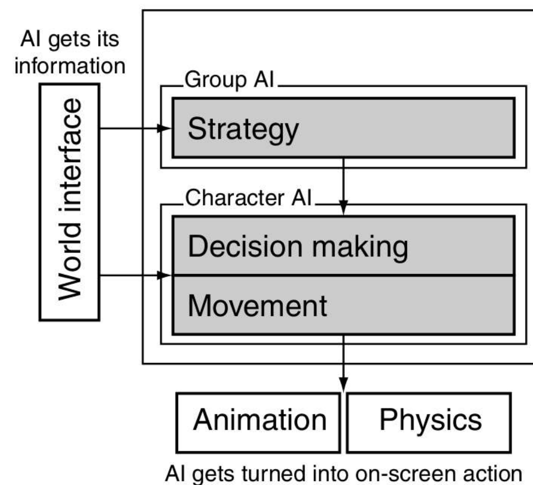

Figure 1: Components of the core AI. Adapted from Millington and Funge (2009)

This model describes an interchangeable module providing two interfaces to communicate with the world, the *world interface*, also called *perception* and an *action interface*. The former is used, to send information of the world's current state, to the AI. The latter tells the world in return, how specific elements have to behave depending on the decisions made. In this model the AI is divided in three sub-modules:

Strategy This part is also called *Group AI* and as the name indicates it usually controls several game elements at once. This is not limited to Non-Playing Characters (NPCs) or agents, but can be any resource of the world which is accessible and alterable, to guarantee challenge for the player. An AI using this part is called *top-down* as it behaves like god looking down on a world and manipulating it.

Decision Making It is part of a *Character AI*. Its role is to make decisions based on different situations for a single entity, sometimes called an NPC and an agent respectively.

Movement This is also part of the *Character AI*. Here are, more or less complex motion behaviors defined like *shoot, evade, hide etc.*

Either *strategy* or *decision making* is mandatory to make decisions. It is no problem, however, to omit one of both. An example for a strategy-only AI is that of Chess and for

a decision making approach would be a single-player fighting game, like Blobby Volley. A game using both could be a strategy game in which the individuals get commands from a strategic entity, but have also implemented individual preferences and limited perception for those.
To make it more specific, a general could give several soldiers commands over radio to march to a specific location, but as soon as one of them spots an enemy, this single one will attack.

Regardless of the type of decision making, any of those three have to be processed before the *movement* takes action. As mentioned before, the movement has implemented different motion behaviors, which can support each other. The movement module therefore receives geometric information from itself and the world, processes the behaviors and calculates geometric data like new positions, velocity or directions. If the game provides a physics engine, these are usually passed as requests there because the AI only decides on "what should be done". The physics engine then has some effect on the commands, to ensure same conditions for every individual. For example if the ground is tilting, all elements regardless of their intentions/actions should slide or fall.

2.2 Reinforcement Learning

Before RL is explained, a small example shall illustrate the process: *An NPC or agent in an ego-shooter has states depending on the values: life, ammunition, position and distance to the next enemy. If the agent is hidden at the moment and an enemy enters his zone of perception, the agent has different options, like attack, flee, stay hidden, use an item etc. If it decides on attacking and loses its life in the process, the agent will get a bad reward for choosing attack in this situation. Next time it finds itself in the same situation it will try another action. To prevent the agent from always performing the same action, because it gets high rewards at the moment (local maxima), the agent will sometimes choose random or suboptimal actions.*

In section 2.1.1 *Characteristics of Game AI* was mentioned that algorithms can have mutable characteristics. These algorithms are called *Machine Learning Algorithms*. According to Sutton and Barto (1998), the AI learns to map actions to situations, by trying them and receiving a numerical reward from the world, indicating its quality, i.e. how good it was to choose this action in the specific situation. By maximizing the reward received in every situation, the AI's decision making converges to be near optimal.

2.2.1 Markov Decision Processes

This and the following sections until section 2.2.3 *Q-Learning* point out the basics of RL, taken from Szepesvári (2010).
To understand RL some definitions have to be made first. The expected value $\mathbb{E}$, is the weighted average of a sequence of random values, obtained by a long probabilistic experiment. The supremum operator of a function $f : \mathcal{X} \rightarrow \mathbb{R}$ gives the least upper bound function value and shall be termed $\sup_{x\in\mathcal{X}} |f(x)|$. The diametrically opposed infimum operator of a function $f : \mathcal{X} \rightarrow \mathbb{R}$ shall be denoted as $inf_{x\in\mathcal{X}}|f(x)|$. Underlying an RLA is a problem called Markov Decision Process (MDP), describing an environment in which decisions have to be made on partly random circumstances. The MDP is a triplet $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$. Whereas $\mathcal{X}$ defines the finite set of states and $\mathcal{A}$ is the finite set of actions available. The state transition probability kernel $\mathcal{P}(x, a, x')$ is providing a probability of translating to state x' by choosing action a in state x. The transition probability kernel $\mathcal{P}_0(\{x'\} \times \mathbb{R}|x, a)$ gives the probability that the next state and reward are received when choosing action a in state x. For clarity equation (1) shows the relation between the two kernels.

$$\mathcal{P}(x, a, x') = \mathcal{P}_0(\{x'\} \times \mathbb{R}|x, a) \quad (1)$$

The solution to an MDP is a policy. Such can be *deterministic* $\pi : \mathcal{X} \rightarrow \mathcal{A}$, which simply maps a state to an action or *stochastic* $\pi(a|x)$, which gives the probability of choosing action a in state x. A policy can even be *stationary* which means, that the probability distribution only depends on the previous state.
Depending on the reward drawn by $\mathcal{P}_0$, deterministic policies will change the action and stochastic ones the probability distribution.

2.2.2 Contextual Reinforcement Learning

RL can be used for any kind of environment. To clarify, *contextual* in this thesis, refers to an environment with an existing policy. The idea of Contextual Reinforcement Learning (CRL) is to use established AI-implementations, extend the *action space* and try to optimize the given policy by choosing the most proper action in this state.

In contrast to this, pure RL starts with a very large *state-action space* as the characteristic (i.e. conditionals) of the AI has to be developed by the RL itself. Most optimal would be a state for every possible situation and all actions which can be performed in those situations. It is clear that this would let the learning-rate grow exponentially depending on the number of parameter available and value-ranges of those.

Pure RL should deliver better performance of the AI when there is infinite time to learn and the algorithm is optimally parameterized, as the agent can learn real optimal behavior and is not limited by specifically implemented character. In most cases CRL could be sufficient as it yields faster results.

2.2.3 Q-Learning

QL, first introduced by Watkins (1989), is called the algorithm family solving the MDPs by finding the optimal policy π^*. A policy is called optimal when the received reward is always the maximum. Trying to find the optimal policy directly is hard, as there are too many to try them all, depending on the state-action space.

Finding an optimal *action-value function*, however, is very easy. Figure 2 shows an example state-action space and how the optimal policy is achieved.

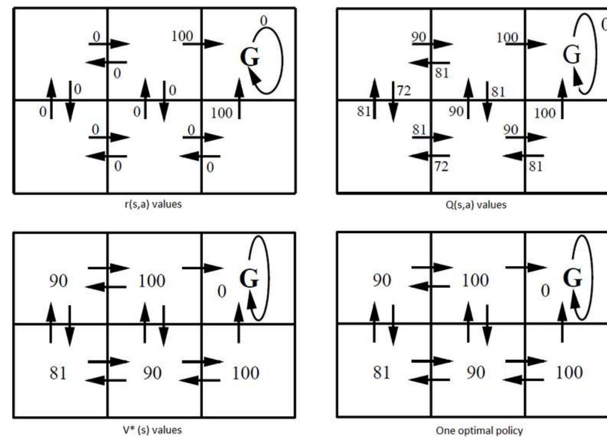

Figure 2: Q-, V- and reward-values for RL problem. Taken from CSE Wiki (2014)

The image in the top-left corner shows the immediate rewards received when the agent lands in a certain state performing the specific action. Only the final state G, gives a reward of 100 all the others return 0. Every action of every state has a Q-value which will converge when using QL explained below which are shown in the top-right corner. As also explained below the V-value is the max Q-value of all values of a state and therefore representing the state and is shown in the bottom-left corner. An optimal policy π^* is

then based on either Q- or V-values, displayed in the bottom-right corner. For a better understanding, QL is described in more detail.

The action-value or Q-value gives the expected value $\mathbb{E}$ of all possible sums of discounted rewards depending on the sequence of x and a chosen, on the condition that $x_0 = x$ and $a_0 = a$, whereas γ is the discount factor and $R(x, a)$ the delayed future reward of performing a in x. Hence, when the rewards get higher the value gets higher. The function is shown in equation (2).

$$Q^{\pi}(x,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1}(x,a) \middle| x_0 = x, a_0 = a\right] \tag{2}$$

This is the action-value function following π. The optimal action-value function is the sum of the discounted distributed future values and is shown in equation (3).

$$Q^*(x,a) = R_{t+1}(x,a) + \gamma \sum_{x \in \mathcal{X}} \mathcal{P}(x,a,x') \sup_{a' \in \mathcal{A}} (Q^*(x',a')) \tag{3}$$

The optimal action-value function has to be executed simultaneously for all (x, a, x') to make it converge. By making an update every time the agent transitions from x to x' by using a, the previous and new values can be averaged using a exponentially decaying learning rate $\alpha_t = \frac{1}{t}$. The probability of this transition is $\mathcal{P}(x, a, x')$, which is why the update can be performed without using the state transition probability kernel. Convergence is guaranteed when the update is performed infinitely often in every state with every action. The learning operator shall be denoted by $v \overset{\alpha_t}{\leftarrow} w$ and can be rewritten as $v = (1 - \alpha_t)v + \alpha_t w$. Then the QL update rule can be described as in equation (4).

$$Q_{t+1}(x_t, a_t) \overset{\alpha_t}{\leftarrow} \left[R_{t+1}(x,a) + \gamma \sup_{a' \in \mathcal{A}} Q_t(x_{t+1}, a')\right] \tag{4}$$

Action-values show which action to take when in a specific state. Analogously the V-value or state-value shows which states should be preferred. The V-value is the maximum action-value of a state as shown in equation (5).

$$V_t(x_t) = \sup_{a' \in \mathcal{A}} Q_t(x_t, a') \tag{5}$$

The optimal policy π^*, independent of its properties deterministic or stochastic, is obtained by choosing actions greedily based on their Q-values (non-stationary) or V-values (stationary).

2.2.4 Speedy Q-Learning

SQL described in Reinforcement Learning with a Near Optimal Rate of Convergence by Gheshlaghi Azar et al. (2011) is an attempt to speed up QL by using three generations of the state-action space $\mathcal{X} \times \mathcal{A}$.

First k, the generation counter is initialized to 0. As well as the visit-counters of all state-action pairs of the current generation $N_0(x, a)$. Additionally the Q_{-1}-values of the non-existent previous generation are set to the current Q_0-values. This is shown in algorithm 1.

Algorithm 1 SQL initialization. Adapted from Gheshlaghi Azar et al. (2011)

$k = 0$
for all $(x, a) \in Z$ **do**
  $Q_{-1}(x, a) = Q_0(x, a)$
  $N_0(x, a) = 0$

When the agent acts the new action A_t is drawn from the policy distribution kernel π and the new state X_{t+1} is drawn from the transition distribution kernel $\mathcal{P}_0$. Thereafter SQL is initiated. Whereby t is the current algorithm iteration. Both the act-function and the SQL algorithm are shown in algorithm 2.

Algorithm 2 Agent acts and initiates SQL. Adapted from Gheshlaghi Azar et al. (2011)

function ACT
  $A_t \sim \pi(\cdot|X_t)$
  $X_{t+1} \sim \mathcal{P}_0(\cdot|X_t, A_t)$
  SQL(X_t, A_t, X_{t+1})

function SQL(X_t, A_t, X_{t+1})
  $\eta_N = \frac{1}{N_k(X_t, A_t)+1}$
  $Q_{k-1}(X_t, A_t) \overset{\eta_N}{\leftarrow} [R(X_t, A_t) + \gamma \sup_a Q_{k-1}(y_k, a)]$
  $Q_k(X_t, A_t) \overset{\eta_N}{\leftarrow} [R(X_t, A_t) + \gamma \sup_a Q_k(y_k, a)]$
  $Q_{k+1}(X_t, A_t) \overset{\alpha_k}{\leftarrow} [kQ_k(X_t, A_t) - (k-1)Q_{k-1}(X_t, A_t)]$
  $N_k(X_t, A_t) + +$
  if $\min_{(x,a)\in Z} N_k(x, a) > 0$ **then**
    $k + +$
    $\alpha_k = \frac{\alpha_0}{k+1}$
    for all $(x, a) \in Z$ **do** $N_k(x, a) = 0$

Generation k only changes when every state-action pair (x, a) is visited at least once. Every time the same (x, a)-pair is visited, the learning rate η_N is decaying. This means that when the (x, a)-pair is visited the first time in this generation, it will adopt the whole new value. Next time it is visited it generates the mean between the old and the new value. As long as this pair is visited in that generation, the update will get weaker. Just as a new generation starts the learning rates are set back. When Q_{k-1} and Q_k have been updated, the update rule assigns the difference of those values, scaled by their generation to the next generation Q_{t+1}. The learning rate α_k is decaying analogously to η_N but on the condition of incrementing generation k.

2.3 Balancing of Algorithms

This section does not outline specific techniques to balance an algorithm or a discussion about balance of specific algorithms, but rather it notes the importance of careful balancing to receive effective and efficient results. As mentioned before algorithms usually come with some parameters or even replaceable code-snippets which make them more flexible but harder to control.
Maybe a good example are Soft Computing Algorithms (SCAs) described in Weicker (2007). The first one is the ANN. It consists of layers, nodes, weights and connections, but is fully customizable. There are special algorithms with a specific setup like Single-Layer Perceptrons or Multi-Layer Perceptrons but depending on the input vector, meaning all the input values, the output vector and the operation for the ANN to learn, there are many different configurations. Therefore the ANN is very complex, compared to other algorithms, but only defines the structure and says nothing about the learning process. How to adapt the net afterwards, i.e. Back-Propagation or EAs, makes it even more complex.
Another example would be to use pure EAs, consisting of a population of settings and the process of mating, recombination, mutation, fitness evaluation and natural selection. The processes are already designed and can be implemented. The population, however, with its individuals have to be created. For the values of the properties, random ones can be used, but which settings to include, how big to choose the population and which specific EA would fit best is still to decide.

Both algorithm families try to copy natural processes and both are very flexible.
All that holds true for Q-Learning as well. Aforementioned algorithm has only two variable parameters α_t and γ. For infinite time the former should go to 0 and the latter to 1 but not reach them. More important though is to choose the right states and available actions, as well as the reward received for landing in a state ($V(x)$) or using an action in a state ($Q(x, a)$).

Choosing either of those right can be very hard. For states it means making the right abstraction of the situation, for the agent to have few enough to accelerate learning and many enough to provide the right amount of sub-situations to try different actions. For the actions it means again to choose less for accelerated learning but also those which are appropriate for the situation.
Giving proper rewards is even harder. For a game which has to end fast and in a specific state, every state can give a little negative reward and a big one for the final state. As the agent tries to maximize rewards, it will try to end the game fast, to receive the minimum of negatives and the big positive reward. Other states like *dead*, may be avoided by giving high negative rewards when reaching those.

The more complex an algorithm the more difficult to balance as the effect of one parameter change becomes increasingly harder to predict. Experience gives hints in which direction and to which extent the parameters have to be changed or which sub-algorithm may be of best use. Still testing will be most sufficient when trying to get optimal results.

2.4 State-of-the-art Reinforcement Learning

2.4.1 Efficient Use of Reinforcement Learning in a Computer Game

In the work of Björnsson et al. (2004) two approaches to speed up QL were introduced. They presented a mobile game with a creature in need of help. Which is raised and formed by the player. It has a mental constitution and physical abilities and can be interacted with; for instance be fed, rewarded, punished and most important forced to learn physical and mental skills, i.e. sprinting and puzzles. There are events in which the player can train the creature and competitions to show off its skills. As every creature is different in its abilities, skills and constitution, the optimal policy for events and competitions can also be different.
Basically they used standard QL, but during the events the player could overwrite the creature's actions, which were remembered to reuse them.
In the first approach, they call imitation runs, they performed reruns in the background. In the second one, bonus reward points, the player's preferred actions add bonus rewards when used. Both approaches yield faster convergence when the player gives good advice, which can be seen in the left plot of figure 3.

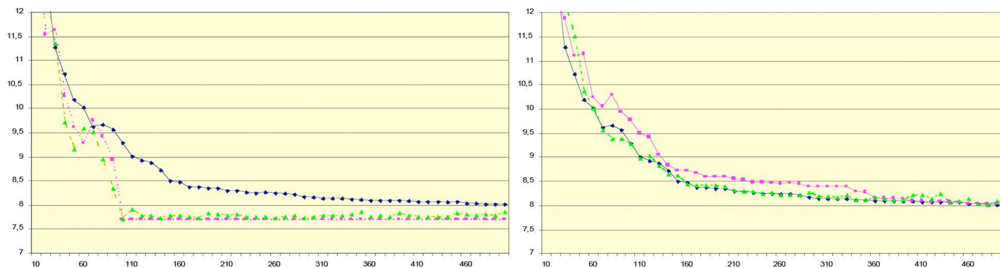

Figure 3: blue: standard, pink: imitation run, green: bonus points; left: good player advice, right: bad player advice. Adapted from Björnsson et al. (2004)

In the right plot the player deliberately chose bad actions, which caused slower performance but still both approaches recovered eventually.

2.4.2 Flappy Bird RL

Sarvagya Vaish (2014) presents on his website *Flappy Bird hack using Reinforcement Learning*. Flappy Bird is a game in which the player controls a little bird, in a graphical environment similar to Super Mario World. The bird is constantly falling because of the gravity and when it hits the floor or an object it dies and the game is over. To prevent it from constantly dropping, the player can tap the screen, making the bird perform a flap - as the game's title suggests - letting it jump a little bit upwards. The only objects in the world are green pipes on the floor and ceiling creating bottlenecks in different heights. Everytime a bottleneck has been passed by the player, she gains a point. Maximizing the score is the games main objective.

For the learning pure QL is used and the states are defined on the properties

- vertical distance from lower pipe,
- horizontal distance from next pair of pipes and
- life (Dead or Living).

Where figure 4 illustrates how the first two properties are calculated.

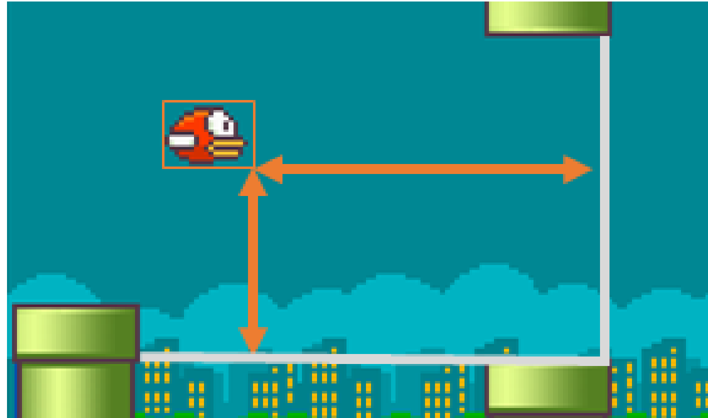

Figure 4: The state properties: vertical and horizontal distance of the next pipe. Adapted from Sarvagya Vaish (2014)

It was further defined the available actions in every state

- click and
- do nothing

and the delayed rewards after performing an action

- **+1** if Flappy Bird is still alive and
- **-1000** if Flappy Bird is dead.

With this configuration it took Flappy Bird 6-7 hours of training to be good enough to score approximately 150 points. It is further suggested to instantiate several birds learning on the same Q-values or to let the user give input to speed up convergence.

2.4.3 Multi-agent reinforcement learning

An adapted version of QL, called *minimax-Q* is used in the work of Littman (1994), to approach multi-agent environments. These are stochastic as choosing an action in a state depends not only on the current situation but also on the intention of other agents.
To address this problem, the agent does not only try to maximize the discounted rewards, instead it tries to maximize the discounted rewards of the worst situation possible, i.e. the situation, when the opponent tries to minimize the discounted rewards by choosing appropriate actions. The definition of the update rule has to be adapted by using the

finite set $\mathcal{O}$ as the available actions of the opponent and o is drawn from this set. The new algorithm is shown in equation (6).

$$Q_{t+1}(x_t, a_t, o_t) \overset{\alpha_t}{\leftarrow} \left[R_{t+1}(x, a, o) + \gamma \sup_{a' \in \mathcal{A}} \inf_{o' \in \mathcal{O}} Q_t(x_{t+1}, a', o') \right] \tag{6}$$

This algorithm was then used in an experiment, in which two agents play an abstraction of soccer, in a 4x5 grid shown in figure 5.

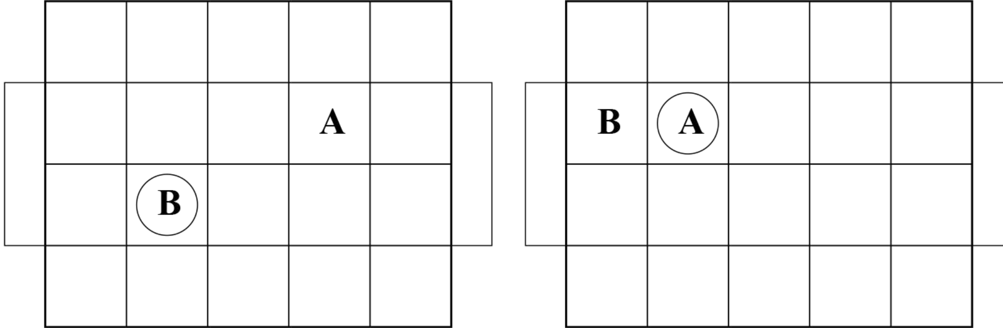

Figure 5: left: starting positions; right: defending situation. Adapted from Littman (1994).

The left grid shows the starting positions of **A** the agent and **B** the opponent. The circle is the ball which is given to anyone randomly when the game starts. Both agents can choose between the five actions go north, east, south, west or stay. When both have chosen their actions, they are performed in random order. By reaching a square with the opposing character, the ball is lost to the other one and the action is undone. A good strategy then is to position oneself in a square which the other one will want to get. When **A** reaches the left goal or **B** the right one, they score a point.
The right grid of figure 5 shows a situation in which **A**'s actions could be forfeited easily by the opponent. There is only a chance of succeeding if the agent chooses randomly between going south and staying.

The results of the performed tests are presented in table 1.

	MR		**MM**		**QR**		**QQ**	
	% won	games	% won	games	% won	games	% won	games
random	99.3	6500	99.3	7200	99.4	11300	99.5	8600
hand-built	48.1	4300	53.7	5300	26.1	14300	76.3	3300
MR	35.0	4300						
MM			37.5	4400				
QR					0.0	5500		
QQ							0.0	1200

Table 1: Test results of the minimax-Q algorithm. Adapted from Littman (1994)

There were two minimax-Q and two QL agents tested. One of each was trained by playing for one million steps against an opponent with a random picking policy (i.e.

MR, QR), while the others were trained against versions of themselves (i.e. MM, QQ). The learned values were fixed and all four then had to compete against an agent with a random picking policy, an agent with a hand-built policy and for the third test, against a version of themselves after training. In these tests they were playing for 100,000 steps. In the columns labeled *% won* the percentage of won games is stated, whereas in the columns labeled *games*, the number of games completed during the 100,000 steps is given. The results show that one million steps for training does not suffice for the minimax-Q to converge, as there is a difference in the number of games won, against the hand-built agent. Furthermore one can see that both QL agents have learned different policies, so that there was a big performance difference against the hand-built agent.
The third tests show that the minimax-Q agents would need more training as they should perform at least even against the strongest opponent and that the QL agents did not score a single point.

3 Part 2 - Experiment

3.1 Blobby Volley 2

Blobby Volley was first developed by Daniel Skoraszewsky and Silvio Mummert and released 2000. Blobby Volley 2 is now further developed by a free developer team and licensed under GPL for Linux, Mac OS X and Windows. This second version is implemented in *C++* and uses *Lua* for its AI scripts.

Figure 6: Screenshot of the game Blobby Volley 2

It is a 2D volleyball game, in which the players are symbolized as colored blobs. The game is designed for single- and multi-player. It is played one on one and usually has the following rules:

- The first to reach 15 points wins the game
- There has to be at least a 2-point difference to win the game
- A loss of ball is considered a win for the other party
- Touching the ball 4-times in a row is considered a loss of ball
- When the ball touches the ground it is a loss of ball
- To score a point, the player first has to catch the initiative (indicated by a small exlamation mark (!) besides the scores)

In additon to the AI scripts, there are also the files `rules.lua` and `config.xml`. The former can be used to adapt the rules mentioned in the list above. The configuration-file can be changed directly or in the game itself through the *Options* menu. The important settings, used later for the experiments are then presented in section 3.4.

3.2 Blobby Volley 2 Agent - Reduced

Reduced is one of the included AI scripts. It is the base for two of the implemented agents in section 3.3 and is furthermore the competitor to compare their performance changes.

Every AI Lua script for Blobby Volley contains at least three entry points:

OnServe(ballready) Gets called every frame as long as the ball is on the agent's field side, waiting to be kicked off by the agent.

OnOpponentServe() Gets called every frame as long as the ball is on the opponent's field side, waiting to be kicked off by the opponent.

OnGame() Gets called every frame after kick-off until one of both scores a point.

All those entry points are mandatory. To understand the character and the behavior of the agent, the state-action space, implemented throughout all three entry points, is illustrated in figure 7.

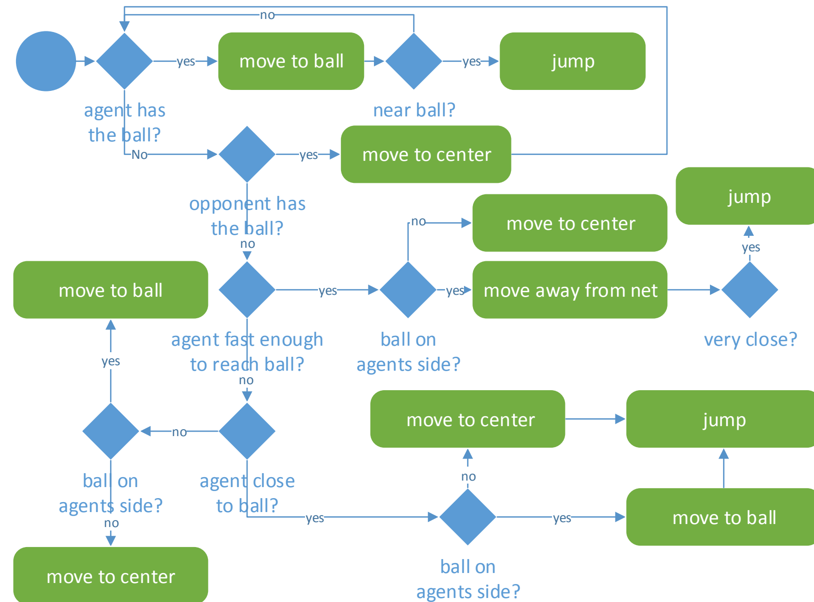

Figure 7: Behavior of Blobby Volley 2 agent Reduced.

When in the *OnServe()* function, the agent has to kick off the ball. As long as the agent has not reached the x-position of the ball it moves in that direction. As soon as it reaches it, it jumps and hits the ball.
After hitting the ball the agent uses the *OnGame()* function. When the agent theoretically is able to reach the ball with its next move and the ball is on its side of the field it tries to stay away of the net and jump to push it over the net. Should it be on the other side the agent moves to the center of its field.
If the agent thinks the ball is not reachable with its next move and the ball is on the

agents side it moves to the ball, otherwise to the middle of its side. Should the agent still be close to the ball it will jump regardless of the balls side.
When the agent loses the ball, the agent uses the *OnOpponentServe* function and has to wait until the opponent kicks off the ball by staying in the middle of its side.

Several functions, provided by the game to access values of the current game state and to manipulate the agent's blob, which are frequently used in the tests, are listed in table 2.

function	**description**
posx()	the current x-position of the agents blob
posy()	the current y-position of the agents blob
oppx()	the current x-position of the opponent's blob
oppy()	the current y-position of the opponent's blob
ballx()	the current x-position of the ball
bally()	the current y-position of the ball
speedx()	the current speed of the ball in x-direction
speedy()	the current speed of the ball in y-direction
getScore()	the current score of the agent
getOppScore()	the current score of the opponent
jump()	the blob jumps in the direction it is moving
moveto(number)	the blob moves to number (x-position)

Table 2: Functions available in the AI-scripts of Blobby Volley 2

3.3 Tested Agents

In this section the three tested agents will be described. All of these use SQL differently. The naming is described below:

Pure Random SQL In this agent a completely new state-action space is defined, means it is using a *pure* implementation of SQL. Furthermore, the policy, chooses 20% of the actions *randomly*. On a specific condition described below the learning is initiated.

Deferred Random SQL The state space is the same as in the original *Reduced* agent, the action space, however, is extended by additional actions similar to the existing ones. The transitions the agent is performing are stored until the iteration occurs and the learning is then processed in a *deferred* manner.

Standard SQL This agent has the same extended state-action space as the *deferred* version. The policy, however, chooses in 20% of the time, actions which have never been chosen before. This agent initiates the learning as soon as every action has been visited.

Before the scripts are particularized, some technical limitations of the experiment are presented.
In Blobby Volley 2 only the Lua-interpreter and the Lua math library are added. Apart from that additional libraries like the Lua base library are missing and therefore only a few language features are supported. For the implementation of simple scripts this is not a problem, but for complex scripts the required code lines increase significantly as there are no metatables to implement class behavior and no functions to access tables like *pairs(table)*. Even *load(snippet)* is not available, used for importing external code, which means that code has to be included in every used script separately and if a common part is changed, it has to be changed in the other scripts as well. This can easily lead to inconsistency in the algorithms.
For data the Lua base library usually provides IO-functions for files and the console. The only output function implemented though, is a function called *debug(number)*. It is able to write a message to the console in which the application is launched, with the message *"Lua Debug: number"*, while *number* is an integer or float.
Hence, all of the scripts use arrays and no tables, the debug function to output data and do not share any code.

3.3.1 Pure Random SQL

The first script is a pure implementation of SQL. As a consequence a state-action space has been defined depending on the games properties and not an existing AI script. Such properties would be *posx()*, *speedx()* etc. As the state-space would get too large (many billion states), the properties are clustered and hashed together. The hash is a multi-base number whereas the bases are defined by the number of clusters per value. Table 3 shows the properties received, the boundaries, the mapping from the property to the number and an example.

property	boundaries	mapping	example
current loop	entry points	1, 2, 3, 4, 5, 6, 7	OnServe1 $\rightarrow$ 1
posx()	min 1, max 4	$floor$(posx()/89.375) + 1	90 $\rightarrow$ 2
posy()	min 1, max 5	$floor$(posy()/78.8) + 1	30 $\rightarrow$ 1
ballx()	min 1, max 10	$floor$(bally()/80) + 1	240 $\rightarrow$ 4
bally()	min 1, max 7	$floor$(bally()/85.714) + 1	0 $\rightarrow$ 1
speedxy()	min 1, max 8	$todeg$($norm$(speedxy()))%45°	(-2.1, 0.3) $\rightarrow$ 4

Table 3: Bases of the multibase hash for the states

There are three entry points but the *OnServe(ballready)* function's parameter divides this entry point in two different states. Moreover it is considered a separate state when one of the three loops OnServe1, OnServe2 and OnOpponentServe are visited first after a win or a loss, as this particular situation is when the learning is performed. The position cluster sizes are defined by the number of clusters fitting with a minimum size of $2.5 \times blobby\ radius$. This results in the numbers used in the table to map the values to the cluster indices. The speed vector of the ball is normalized and mapped to the degrees of the unit circle and clustered by 45°.
Doing that the state-space is reduced to 78.400 states. This is still many states and it is not required to visit every state-action pair as suggested in the SQL implementation described in section 2.2.4. Firstly because there is no guarantee that the agent will land in every state and secondly because it would take a considerable amount of time to make the test. Therefore iteration of the generations is performed every time the agent lands in one of the served entry points. The condition

$$\textbf{if } min_{(x,a)\in Z} N_k(x,a) > 0 \tag{7}$$

is changed to

$$\textbf{if } [1,2,3].contains(reverseHash(state)[1]). \tag{8}$$

$[1, 2, 3]$ are the entry points *OnServe(false)*, *OnServe(true)* and *OnOpponentServe()* and the index [1] for the $reverseHash$ function extracts the number of the first base, which is the base for the entry points.
Additionally the policy is drawing the optimal action with a probability of 80%. The resulting policy is shown in algorithm 3.

Algorithm 3 Suboptimal policy choosing random actions.

function π(state)
  if $\xi(0,1) \leq 0.8$ **then**
    $aIdxs = supIdxs_a Q_k$(state, a)
    return $\mathcal{A}_{\text{state}}[\xi(0, \text{sizeOf}(aIdxs), 1)]$
  return $\mathcal{A}_{\text{state}}[\xi(0, \text{sizeOf}(Q_k), 1)]$

$\xi(min, max, [stepsize])$ is the random function used to decide between selecting the optimal or a random action where $\mathcal{A}_{\text{state}}$ is the set of actions for the passed state and $supIdxs$ yields the indices of the elements with the highest value.

3.3.2 Deferred Random SQL

This script is using *Reduced* as the base, which means that the conditions and the type of action are not changed but every time *moveto(number)* would be called with the hard-coded value, *moveto(number)* is called with an x-position in the range $\{x \in \mathbb{N} \mid 32 \leq x \leq 356\}$ with a stepsize of 6 resulting in the set $\{32, 38, \ldots, 356\}$. There is a total of 54 x-positions in the interval, which are reachable by the blob as positions beyond would result in leaving the blob's field's side.
By using this method the agent is retaining its behavior and still trying to improve by choosing similar actions doing that it should receive better rewards.
Every time a routine is left because all actions have been executed for the met conditions, a specific state has been reached. In the context of the agent *Reduced* this means that there are 13 states. Originally every state consists of only one available action, in particular, one action executing all of those being chosen by the agent's conditions. For a better understanding the following code snippet shall illustrate it.

```
function OnServe(ballready)
    moveto(100);
    if ballready and abs(posx() - 100) < 5 then
        jump();
```

This is then rewritten as

```
function OnServe(ballready)
    if ballready and abs(posx() - 100) < 5 then
        action := π(1);
    else
        action := π(2);
    action();
```

while $\pi(1)$ usually chooses any allowed moveto(*number*) combined with jump() and $\pi(2)$ only executes an allowed moveto(*number*).
As mentioned before in standard SQL the action-values are adjusted continously, but adapted not until every state-action pair has been visited at least once. Like in the *pure* implementation of the previous section 3.3.1, the *iterating condition* is met when the agent lands in one of the states of the entry points *OnServe()* and *OnOpponentServe()* respectively. In contrast to the original implementation though, all state-action-state triplets (x, a, x') are added to a queue, which will only be processed when iterating condition is met, by applying the SQL algorithm to update the action-values $Q(x, a)$. Thereafter the iteration occurs and the game continues.

3.3.3 Standard SQL

This script again is using *Reduced* as the base and uses the same states as described in the previous section 3.3.2. The learning, however, is as in the standard implementation. Hence the *iterating condition* is met when all state-action pairs have been visited at least once. For clarity $infIdxs$ yields the indices of the elements with the lowest value. The updated policy is then

Algorithm 4 Suboptimal policy using not visited actions first.

```
function π(state)
    if ξ(0,1) ≤ 0.8 then
        aIdxs = supIdxs_a Q_k(state, a)
    else
        aIdxs = infIdxs_a N_k(state, a)
    return A_state[ξ(0, sizeOf(aIdxs), 1)]
```

whereas $N_k(x,a)$ shall denote the set of visits for each state-action pair in the current generation k. By following this policy 20% of the actions have never been visited before. When every action in every state has been visited the *iterating condition* has been met.

3.4 Test Setup

All test-scripts are written in *Moonscript* which is according to MoonScript (2014), a dynamic scripting language compiling to Lua.
The tests were conducted on a notebook running the Ubuntu derivate *Elementary OS* and were performed automatically. For that a *Makefile* and several *Python*- and *Shell*-scripts were created. The scripts are:

script	**functionality**
`common.py`	natural sorting, parsing text → numbers, replacing text in a file
`replace.py`	wraps the replace functionality of `common.py`
`directory.py`	lists and creates directories at once by following a pattern
`parser.py`	parses results and writes them to files
`misc.py`	creates directories, result-files and reinsert learned information
`echo`	outputs correctly colored strings
`test`	conducts a single test
`Makefile`	starts continuous tests

Table 4: Python, shell-scripts and Makefile for the experiment

Those are available in the git repository `http://git.satanik.at/bsc-thesis-2/` to reproduce the tests.

As the Lua-scripts cannot use IO-functions the game was started in the terminal and output was written to a file called *results.txt*. After a game was over and the application was closed, the file's contents were parsed into `CSV`-format. For continuous learning, the latest information of all state-action pairs is written to a `moon`-file and reinserted into a specially marked part of the test-scripts to overwrite the default values. Additionally every time one of the agents scored a point, both scores and the game time were written to `results.txt` as well.
As the game does not accept any command-line arguments for control, *xdotools*[1] was used to press the important buttons of the game's interface to start and stop the game. The settings used in the `config.xml` are presented in table 5.

option	**expected values**	**set to**
fullscreen	true or false	false
left_script_name	any lua-AI script	*tested script*
right_script_name	any lua-AI script	`reduced.lua`
left_player_human	true or false	false
right_player_human	true or false	false
game_fps	any positive integer	1000
left_script_strength	[best] 0...25 [worst]	13
right_script_strength	[best] 0...25 [worst]	13

Table 5: Blobby Volley 2 settings for the experiment

1. http://www.semicomplete.com/projects/xdotool/

3.5 Results

As mentioned before in section 2.3, it is unclear how different parameters affect the learning in different contexts. Hence, the standard implementation of SQL was tested first with different parameters. After each test one parameter was slightly adjusted to observe the learning behavior and the difference in the results. The tests performed before the actual parameters had been found are displayed in table 6.

attempt	Q_0	γ	ξ_π	R_0	R_{win}	R_{loose}	learn	test
01	0	0.7	0.8	-0.1	100	-100	11:20	06:20
02	0	0.7	0.8	-0.1	10	-10	11:20	12:20
03	0	0.7	0.8	-0.1	1	-1	07:20	10:20
04	0	0.7	0.8	-0.01	10	-10	20:18	08:20
05	0	0.7	0.8	-1	10	-10	14:20	14:20
06	0	0.9	0.8	-1	10	-10	10:20	17:20
07	0	0.5	0.8	-1	10	-10	07:20	10:20
08	0	0.9	0.8	-0.1	10	-10	04:20	09:20
09	0	0.5	0.8	-0.1	10	-10	07:20	17:20
10	0	0.9	0.8	-0.01	10	-10	15:20	10:20
11	0	0.5	0.8	-0.01	10	-10	15:20	08:20
12	0	0.7	0.5	-1	10	-10	20:06	09:20
13	0	0.7	0.9	-1	10	-10	20:22	17:20
14	0	0.3	0.9	-1	10	-10	20:16	04:20
15	0	0.5	0.9	-1	10	-10	22:20	10:20
16	0	0.9	0.9	-1	10	-10	20:17	10:20
17	0	0.7	0.8	-1	10	-10	074:100	03:20
18	0	0.9	0.8	-1	10	-10	056:100	10:20
19	0	0.5	0.8	-0.1	10	-10	057:100	04:20
20	0	0.7	0.9	-1	10	-10	062:100	09:20
21	0	0.5	0.9	-1	10	-10	056:100	04:20
22	0	0.9	0.9	-1	10	-10	069:100	09:20
23	0	0.9	0.9	-1	10	-10	0481:1000	13:20
24	0	0.7	0.8	-1	10	-10	0569:1000	15:20
25	0	0.5	0.8	-0.1	10	-10	0598:1000	8:20
26	0	0.3	0.8	-0.1	10	-10	0518:1000	12:20

Table 6: Blobby Volley 2 settings for the balancing of the algorithm

The used parameters are the starting Q-value Q_0, the discount rate γ, the random threshold ξ_π in policy π, the reward for landing in most of the states R_0, the rewards for landing in *OnServe()* R_{win} or *OnOpponentServer()* R_{loose} after scoring or losing a point, the result of the training *learn* and the result of the trained agent *test* respectively.
In the first 16 tests the training stopped when one of both agents scores 20 points. After that six tests were performed, in which they needed to score 100 points. And then another four tests were conducted with a winning score of 1000.
When observing the agents one can see, that the learning and trained agent uses many

suboptimal actions, which is reflected in the results. In every pretest performed the trained agent lost.

The settings used in the final test, worked on the standard implementation of SQL and are shown in table 7. In all the tests before, the winning and losing rewards were of the same extent; using a low but positive reward for winning and a high negative reward for losing, as used in section 2.4.2 forced the agent to reach the winning state but to avoid the losing state at all costs.

parameter	**value**
Q_0	0
γ	0.7
ξ_π	0.8
R_0	-0.1
R_{win}	1
R_{lose}	-100

Table 7: Blobby Volley 2 settings for the experiment

All three agents have been trained until one character has won by reaching a score of 100 points. Thereafter ten tests per agent have been performed with the given settings.

Figure 8 shows the score differences of the pure implementation of SQL. On the x-axis the game time is plotted while the y-axis shows the score difference by subtracting the points of the original agent from the points of the tested agent. The lines show that this agent had never won a point and had not been able to withstand the original agent for very long.

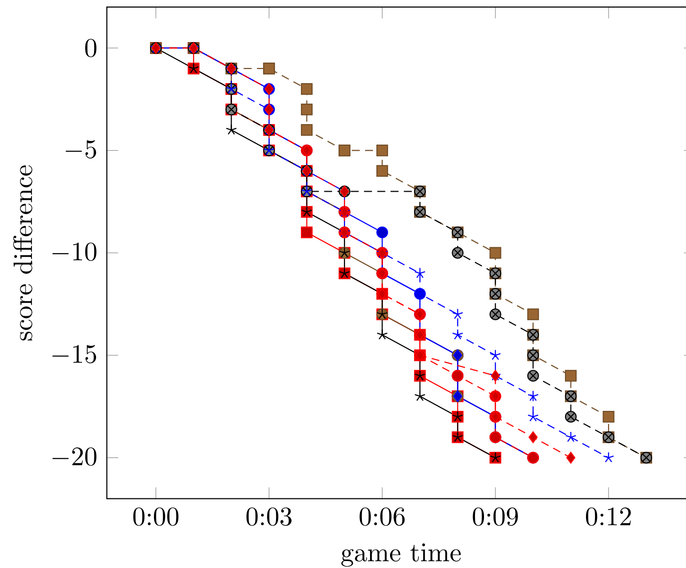

Figure 8: Score difference *Pure SQL* and *Original* agent.

The next figure 9 is analogous to the previous and shows the results of the tests performed with the agent using the deferred version of SQL. This agent had won most of the tests and the point differences are more positive than negative. The games also had taken longer to finish compared to the ones of the pure implementation. Figure 10 displays

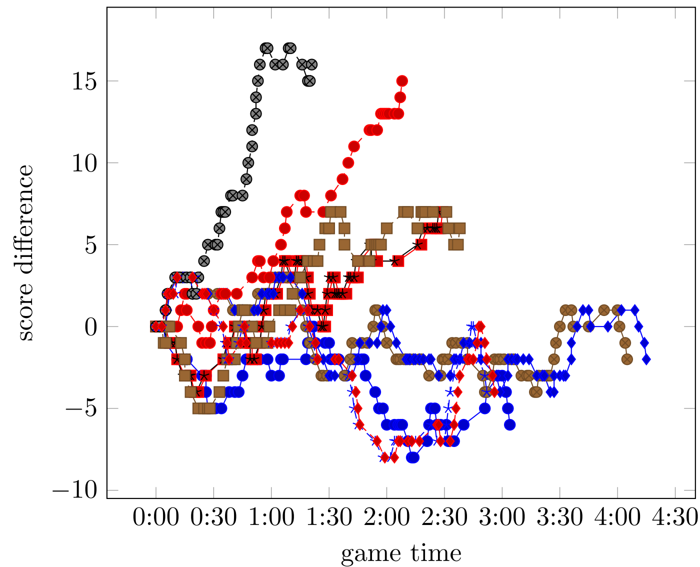

Figure 9: Score difference *Deferred SQL* and *Original* agent.

the results of the standard implementation of SQL. The score differences are primarily positive and most of the time bigger than ten points. The tested agent had won all of the games and they took longer than the ones of the pure implementation as well.

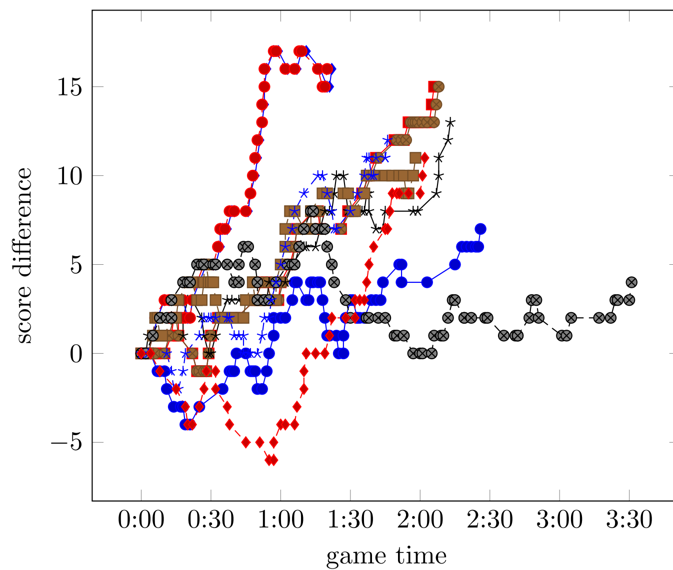

Figure 10: Score difference *Standard SQL* and *Original* agent.

3.6 Discussion

Section 3.5 shows that for most of the parameter settings, the agent was constantly performing worse. Performing more tests with each configuration would yield more statistically relevant information on how to set the configuration optimally.
Furthermore, both contextual approaches work better than the pure implementation with the chosen test setup, which was expected as the pure implementation's state-space is big and needs many iterations to converge. Hence, it makes many wrong moves. Longer training would most likely improve the agents performance as one could watch the agent learning to move near the ball more times while processing the training.
Another approach would be to use the minimax-Q algorithm presented in section 2.4.3. This can be achieved by replacing the sup operator with the $\sup\inf$ operator, iterating over the anticipated actions of the opponent, as in the script there is no way to get the used action.

All three agents were trained using the original *Reduced* as their competitor. Therefore, all of them strove to optimize their behavior only to beat this one opponent. Making more training iterations with different opponents could be the next step to further improve the agents.
Using several instances to train parallel or giving user input to advice the agents, as suggested in sections sections 2.4.1 and 2.4.2, would not be applicable in this application because therefore the application itself would need to be remodeled to allow instantiate several agents and allow user input when agents are playing against each other.

4 Conclusion

Contextual Reinforcement Learning is a way of improving existing agents by extending its state-action space but using the current policy as the base for the training. This retains the agents habits and can yield good performance improvements in decent time. In the scientific sector Q-Learning, and adaptions like Speedy Q-Learning and minimax-Q, are the most used form of Reinforcement Learning. Most of the papers though, concentrate on defining a state-action space for a pure implementation, instead of using established ones. In the economical sector this could make a big difference, preferring mainstreamed Artificial Intelligence with desired behavior, over a strictly optimized one, both in terms of user experience and extended learning-time.
Yet, it is important to properly parameterize the used algorithm otherwise the learning process may be prolonged in terms of time or it could lead to unintended behavior.

The conducted experiment, in which three different agents have been tested against the same opponent, shows that Contextual Reinforcement Learning yields adequate results with few iterations. Combining different Reinforcement Learning Algorithms and using additional methods like user input for advice or parallel learning of several instances updating the same values, could improve the learning process and make Contextual Reinforcement Learning preferable.

References

Björnsson, Yngvi, Vignir Hafsteinsson, Ársæll Jóhannsson, and Einar Jónsson. 2004. "Efficient Use of Reinforcement Learning in a Computer Game." In *Computer Games: Artificial Intellignece, Design and Education (CGAIDE'04),* 379–383.

CSE Wiki. 2014. "Q- and V-values of a reinforcment learning problem." August. `http://cse-wiki.unl.edu/wiki/index.php?title=File:Ex_All.png&oldid=13752`.

Gheshlaghi Azar, Mohammad, Rémi Munos, Mohammad Ghavamzadeh, and Hilbert Kappen. 2011. "Reinforcement Learning with a Near Optimal Rate of Convergence" (October). `http://hal.inria.fr/inria-00636615`.

Littman, Michael L. 1994. "Markov games as a framework for multi-agent reinforcement learning." In *Proceedings of the 11th International Conference on Machine Learning (ML-94),* 157–163. Morgan Kaufmann.

Millington, Ian, and John Funge. 2009. *Artificial Intelligence for Games, Second Edition.* 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

MoonScript. 2014. "A programmer friendly language that compiles to Lua." August. `http://moonscript.org/`.

Sarvagya Vaish. 2014. "Flappy Bird RL." August. `http://sarvagyavaish.github.io/FlappyBirdRL/`.

Sutton, Richard S., and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning.* Cambridge, MA, USA: MIT Press.

Szepesvári, Csaba. 2010. *Algorithms for Reinforcement Learning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Watkins, Christopher John Cornish Hellaby. 1989. "Learning from Delayed Rewards." PhD diss., King's College. `http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf`.

Weicker, Karsten. 2007. *Evolutionäre Algorithmen (2. Auflage).* Stuttgart: Teubner.

List of Figures

1 Components of the core AI. Adapted from Millington and Funge (2009) 5
2 Q-, V- and reward-values for RL problem. Taken from CSE Wiki (2014) 8
3 blue: standard, pink: imitation run, green: bonus points; left: good player advice, right: bad player advice. Adapted from Björnsson et al. (2004) . 12
4 The state properties: vertical and horizontal distance of the next pipe. Adapted from Sarvagya Vaish (2014) 13
5 left: starting positions; right: defending situation. Adapted from Littman (1994). 14
6 Screenshot of the game Blobby Volley 2 16
7 Behavior of Blobby Volley 2 agent Reduced. 17
8 Score difference *Pure SQL* and *Original* agent. 25
9 Score difference *Deferred SQL* and *Original* agent. 26
10 Score difference *Standard SQL* and *Original* agent. 26

List of Tables

1 Test results of the minimax-Q algorithm. Adapted from Littman (1994) 14
2 Functions available in the AI-scripts of Blobby Volley 2 18
3 Bases of the multibase hash for the states 20
4 Python, shell-scripts and Makefile for the experiment 23
5 Blobby Volley 2 settings for the experiment 23
6 Blobby Volley 2 settings for the balancing of the algorithm 24
7 Blobby Volley 2 settings for the experiment 25

List of Equations

1 Relation between the state transition probability kernel and the transition probability kernel 7
2 Q-function following π 9
3 Optimal Q-function 9
4 Q-Learning update rule 9
5 V-function 9
6 minimax-Q update rule 14

List of Algorithms

1 SQL initialization. Adapted from Gheshlaghi Azar et al. (2011) 10
2 Agent acts and initiates SQL. Adapted from Gheshlaghi Azar et al. (2011) 10
3 Suboptimal policy choosing random actions. 20
4 Suboptimal policy using not visited actions first. 22