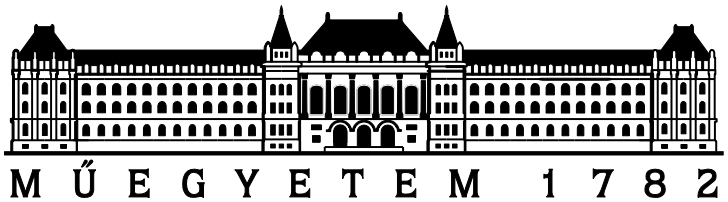


Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Reinforcement Learning Based on Visual Information

Master's Thesis

Author

Gergő Kincses

Supervisor

István Engedy

2017

CONTENTS

HALLGATÓI NYILATKOZAT

Alulírott Kincses Gergő, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 05. 27.

.....

Kincses Gergő

Kivonat

A gépi tanulás, beleértve a megerősítéses tanulást, az életünk szerves részét képezik, gyakran anélkül, hogy észrevennénk. Az otthonaink nemsokára teljesen önvezéreltek lesznek, az autóink hamarosan emberi beavatkozás nélkül juttatnak el minket egyik pontból a másikba, illetve rövid időn belül az egyszerű emberi feladatok nagy részét autonóm robotok fogják majd végezni. Az MI algoritmusok csalás detektálást és kockázatelemzést végeznek, összegyűjtik a számunkra fontos és érdekes híreket, ajánlásokat tesznek amikor Netflixen vagy az Amazon webáruházában nézelődünk, illetve figyelik a házunkat vagy a rendszereinket, mikor mi éppen nem tudjuk. A listának koránt sincs vége, ám már ebből a pár példából is sejthető, hogy a gépi tanulás egy meglehetősen felkapott témakör, rengeteg érdeklődővel mind kutatási, mind piaci oldalról.

A megerősítéses tanulás, illetve a gépi tanulás mellett a vizuális adatfeldolgozás témaköre is jelentős fejlődésen megy keresztül, egész egyszerűen azért, mert szükségünk van rá. Kellenek olyan gépek, amelyek egy szempillantás alatt képesek felmérni a legyártott rozsdamentes acél alkatrészek minőségét egy kamerával, kellenek olyan algoritmusok, amelyek képesek nagy mennyiségű írott szöveg ellenőrzésére és kiválogatására, illetve kellenek olyan rendszerek, amelyek életeket menthetnek úgy, hogy döntéstámogatást nyújtanak az orvosoknak röntgenfelvételek elemzésével.

Jelen dolgozatomban megvalósítottam egy kis keretrendszert, aminek segítségével kombinálom a megerősítéses tanulást és a vizuális adatfeldolgozást, illetve kipróbálok különböző megerősítéses tanulás alapú algoritmusokat egy egyszerű játékon. Az algoritmusok futása során az ágens a tanuláshoz kizárólag a játékból jövő képi információra támaszkodik. A futtatott algoritmusok eredményeit kiértékeltem, kidomborítva az általuk használt felfedezési stratégiák hatékonyságát.

Abstract

Machine learning, including reinforcement learning plays an important role in our everyday life, sometimes without us even noticing it. Soon, our homes can fully manage themselves, our cars can drive to their destinations without any human interaction and most simple human tasks will be done by autonomous robots. AI algorithms do fraud detection and risk analysis, can collect the most important news for us based on our preferences and interests, make recommendations for us when browsing Netflix or Amazon, and can monitor our property or systems when we can not. The list is far too long to be fully introduced here, but these few examples show that machine learning is currently a very hot topic, and it is in the focus of many researchers and market participants.

Besides reinforcement learning and machine learning, the field of visual data processing is going through a significant development as well. The reason behind this development is the fact that we need it. It is necessary to build machines that can check the quality of freshly manufactured stainless steel parts with a camera above the production line in the blink of an eye, or to run algorithms which can check and sort immense amounts of handwritten text, or to create systems which can save lives by helping doctors to make decisions by analysing radiograms.

In this thesis I created a small framework for combining reinforcement learning and visual information processing, and for testing different reinforcement learning algorithms on a simple game. During the running of these algorithms, the RL (Reinforcement Learning) agent learns by receiving only visual information about the game. I evaluated the results of these algorithms highlighting the differences between their efficiency in exploration.

1. Introduction

In this chapter I describe the theoretical background and challenges of this work in accordance with the motivation and objectives.

1.1. Context

1.1.1. Visual Information

In our days, the most frequently used way of communication, expressing intentions or conveying information is through visual representation. Since the strongest sense of human beings is sight, this representation is obvious, easy to interpret and in general is a very good descriptor for particular structures or situations. Visual representation of information appears in several areas of life. From signs and pictograms on a highway or an airport, through large billboards and artworks, to multimedia advertisements and videogames.

As for information technology, the practice of handling visual data changed and improved a lot in the past decades, both in methodology and technology. As the image and video capturing hardware became more advanced and the processing algorithms more efficient, working with visual data became more and more easy and for some type of problems turned into the primary tool for solution. Such problems are for example the automatic quality check on production lines, optical character recognition for helping sight impaired users, or the lane-centring steering and traffic sign recognition of self-driving cars.

In the field of machine learning, the usage of visual information as input for the algorithms became prevalent as well. Partly because of the technological advancements in the last few years, but mainly because a fundamentally vision-based problem from real life can be best described visually, and the outcome is better when less transformation is done during the execution of the algorithm. It is simply due to the fact that if the input is not transformed into another representation (a matrix for example), those errors and flaws which would occur during the transformation will not affect the input and the algorithms.

1.1.2. Deep Learning

From the late 80's the concept of neural networks slowly became more and more neglected and forgotten as a supervised learning tool or a field to study. New procedures

started to show up, which were thought of as more effective and manageable methods. By the mid 90's the AI community almost entirely lost faith in classic neural networks. [1]

A handful of dedicated scientists however did not give up, and continued to work on the field of neural networks. One of them was Geoffrey Hinton, who carried on researching and publishing with his team even in the quietest times, and he was tirelessly working on restoring the old recognition of the concept of neural networks. [2]

Years passed and in 2006 the long-waited break-through finally arrived: The Canadian government gave found to Hinton and his team. However, achieving this did not go easy.

The funds for research in the field of neural networks were lacking, as the future of any upcoming solution would have been highly questionable. Hinton and his colleagues put tremendous amount of effort to consciously wrap the old concept of neural networks into something fresher and less biased. This is how the name "deep learning" came into being.

Thanks to this communication trick, the funding from the Canadian government, and of course the work, and initial successes of the researchers, deep learning started to develop rapidly. From that time on, papers came one after the other, records were broken month by month, and the new ideas were sucked up by the industry.

As can be seen from this introduction as well, the phenomenon of deep learning is nothing more than neural networks with more layers, more neurons and - thanks mostly to the technological advancements – more effective training algorithms. It is however a very effective tool for supervised learning, be it either a labelled image database or a bunch of time series data from a stock market. Deep neural networks are more protected against overfitting thanks to the many new ideas, and they excel at finding those hidden connections among the data which are invisible to their predecessors.

1.1.3. Reinforcement Learning

Reinforcement learning is a unique type of learning within the field of machine learning. In some aspects, it is similar to supervised learning, in others it is more like a non-supervised method. Despite these similarities, reinforcement learning generally makes up a third, separate category. In an RL algorithm the learning agent operates in a more or less controlled and observable environment and collects rewards. The goal of the agent is to

maximize the obtained reward and to learn the optimal actions and action selection strategy (policy) by which the most possible reward can be collected. From this two-sided goal comes the centre dilemma of the reinforcement learning, the exploration-exploitation trade-off, which will be explained more deeply later. [3]

In general, an RL problem and algorithm builds up as follows. There is a set of *states* S, which describes the actual state of the agent. These states are made up of observations on the environment, and the inner condition of the agent combined. The states can come directly from observations or as a result of calculations done on those observations. Besides set S, there is a set A which contains the actions performable by the agent in a given state and a scalar reward signal R which is realized in certain states by the agent.

In addition, an RL algorithm generally contains an oracle and a strategy. The oracle can come in many forms. It can be a model, a set of rules, a function, a table, or anything, the only criteria is that it should be able to tell what effect will executing a certain action in a certain state has on the sum of the collected rewards from that point onwards. The strategy (*policy*) does not have a general form either. Basically, its role is to explicitly define how the agent shall use the information given by the oracle, that is, which action should be executed in a certain state.

The previous paragraph shows that reinforcement learning differs from supervised learning, in a way that there is no knowledge at the agent's disposal on what are the optimal actions in the observed states. For this, the agent must rely on the reward signal, but it is a bit complicated. The problem is that the number of states where rewards can be observed is very small compared to the number of states where rewards can not be observed.

In the end, when the agent chooses an action, it all comes down to one simple question: how useful are those states which can be reached by actions currently available? This is calculated from the estimated sum of rewards that can be obtained starting from that state. This is called *utility* and it is given by the Bellman equation:

$$V^{\pi}(s) = \mathrm{R}\big(\mathrm{s}, \pi(\mathrm{s})\big) + \gamma \sum_{s'} P\big(s' \big| s, \pi(s)\big)\, V^{\pi}(s') \qquad \text{(Eq. 1.1.3.1)}$$

In the equation π is the policy, and P is the probability of getting into state s' if policy π is being followed in state s. Variable γ is the so-called *discount factor* which weights the calculated utilities according to their distance from the starting state, that is, the further a particular state is from the starting state, the smaller the effect it will have on its utility value. It is important to note, that the equation presumes the existence of a state transition

model. However, if such a model is not available, the equation is still usable with some modifications.

From the previous paragraph one can figure out that the more actions have been chosen in a certain state - so more state transitions have been observed - the more accurately can the agent define the utility of that state. Usually, trying every action in every state is not possible, so the dilemma naturally arises: is it better to try new actions in hope of finding one with great return, or harness the current knowledge and collect as many reward as possible? Explore or exploit? Finding the balance between exploration and exploitation is the fundamental challenge of reinforcement learning, because if only one of them is done exclusively during the execution of the learning algorithm, the underlying problem can not be solved for sure.

Finally, with regards to reinforcement learning, one has to say at least a few words about *Q-learning*, which is currently one of the most prevalent methods in the field of RL. Q-learning got its name after the *Q-values* which it uses instead of state utility values defined by the Bellman-equation previously. *Q-values* are also called as *action values*, and they represent the utility of an action in a certain state. Similarly to state utility, *action value* is the maximum possibly achievable reward from a certain state if the agent takes that action. Although they are different, it is easy to extract one form the other:

$$V(s) = \max_a Q(s,a) \qquad \text{(Eq. 1.1.3.2)}$$

That is, the utility of a state is the biggest value among the values of the actions that can be chosen in that state. Basically, the Q-value version of the Bellman equation comes right from the previous formula:

$$Q(s,a) = R(s,a) + \gamma \max_a Q(s',a) \qquad \text{(Eq. 1.1.3.3)}$$

Q-learning, as can be seen in the equation above, does not need to know the transition probabilities P, which is very useful in some situations, and makes some aspects of the learning procedure simpler.

1.2. Motivation

The process of making decisions based on visual information is becoming more and more important within the field of machine learning. In some cases, it is only about supporting decisions by providing details and additional pieces of information to the user. But in

some other cases, when the decisions need to be made fast or the goal is to extract the human factor as much as possible, the task is fully about making actual decisions.

The three fields described in the previous section are a very good starting points to accomplish such tasks. Visual data recording and processing is required for a computer or an algorithm to be able to formalize the problem. Reinforcement learning provides a great theoretical framework and methodology to deal with complex processes. And finally, deep learning is ideal for recognizing patterns and aiding the calculations of both reinforcement learning and image processing. In addition, these three fields are actively researched and technologically matured enough to be tried to be used in cooperation with each other.

My motivation behind writing this thesis – in accordance with the factors detailed above - was to see how these approaches and methods work together, and to check whether they are effective if used together or not.

1.3. Obstacles

The fields introduced before has several theoretical and practical difficulties when it comes to implementation or design. In this part of the chapter some of these difficulties are overviewed briefly.

1.3.1. Hardship of Handling Visual Information

When working with images, especially when there are many of them, the two most challenging tasks are to store them, and to process them.

The difficulty with storing is that if you look at an image as a "single piece of information", it is usually multiple orders of magnitude bigger than a similar data piece from a different domain. For example, a 640x480 pixel RGB image holds almost one million different values, a time series of a one hour period in second division - even with multiple dimensions - holds a few thousands, while both of them represent a single piece from the information set describing a task.

The problem with processing is that during the execution of image processing algorithms it is often necessary to keep the image and sometimes the partial results of the calculations constantly available, which can be very resource-demanding, primarily with respect to memory usage.

1.3.2. Deep Learning Challenges

Using a deep neural network is not that difficult, especially if there is a well-written framework for it, which is very likely nowadays. Using a deep neural network effectively is a whole different story. Lots of things can go wrong, if for example the input data is not made or handled correctly, or if the learning algorithm is not used properly, not to mention the configuration of hyperparameters. There are usually more hyperparameters in a deep network than in a shallow one, and in general there are more things on which one needs to keep an eye on when using a deep neural network.

The other difficulty is that in most cases deep neural networks use significantly larger weight matrices and input dimension space, from which various computational issues can arise, such as the slow convergence of the learning algorithm or high memory and CPU demand.

1.3.3. Reinforcement Learning Complexity

The root of most challenges in a reinforcement learning problem - especially if it is not some benchmark problem - lies in the size of the state space. In most cases it is huge, complex and extracting good features and descriptors from it are extremely difficult. Usually, the state space is so big, that even keeping the whole thing in memory at the same time is impossible.

From these challenges, comes many other. With so many states, and complex representation, how can one differentiate every state from each other? How can the “distance” be measured between states? How can the state space be traversed? Inconvenient questions as such arises, as one works her way deeper into solving a certain problem, not to mention the exploration itself, which can be very hard in lack of the ability to plan a few states ahead.

1.4. Objective and Contribution

My main objective was to combine visual information representation, reinforcement learning and deep learning, by creating an agent which can learn to play a simple game. The agent only receives images of the environment, and uses a deep neural network in the decision-making procedures.

Working on this objective, I created a framework in Python, which can use different environments, implements a deep neural network, and provides a list of parameters through

which the execution of the learning algorithm can be influenced. When an is agent being run, the framework logs its progress real-time.

In this framework, I implemented three reinforcement learning algorithms which I tested on a simple game. The agent in the algorithms only got the visual representation of the game as input, which was a snapshot of the game field in each frame. After the testing of the implemented algorithms, I evaluated their performance using the logs created during each run.

1.5. Structure of the Thesis

This thesis is structured as follows. Chapter 2 presents some of the most prevalent reinforcement learning algorithms and exploration methods. This chapter was the result of the literature research I've done on the different exploration strategies. Chapter 3 details some other approaches and related works, and their difficulties and shortcomings. This part was necessary in order to assemble my framework and to implement exploration strategies. Chapter 4 demonstrates my work, which is the implementation of some of the exploration strategies I found. Chapter 5 shows the results and evaluates the performance of the implemented algorithms. Chapter 6 concludes the thesis.

2. Algorithms and Exploration Strategies

In this chapter I detail the results of the literature research I made on reinforcement learning algorithms and exploration strategies. First, I introduce a formalism by which the algorithms and their inner data structures and operations are described, then I define a classification over the different types of algorithms and methods. Finally, I describe the methods based on the defined classification.

2.1. Overview

The literature of reinforcement learning is very extensive and has a long history. During the years, several algorithms and methods were born, effectives and less effectives alike. In this section I made a brief overview about these algorithms, highlighting their exploration methods or strategies, if they use any special. To keep this work simple and clear-cut, I divided the algorithms into two groups based on their inner representation of the problem on which they are applied to. The two groups are the model-based algorithms and the model-free algorithms. The main reasons behind this classification were the following. Firstly, it was always a strong aspect during the development of the field of reinforcement learning. Secondly, it strictly divides the underlying principles of the algorithms while emphasizes the differences between exploration strategies.

Model-based algorithms are operating on problems which are represented with MPDs (Markov Decision Processes). An MDP is a mathematical framework (with huge toolset and theoretical background) which was designed mainly to model decision making problems in discrete time. An MDP consists of states $s \in S$ which can be changed by taking actions $a \in A$. The outcome of an action a in a state s is not always deterministic, it is defined by the so-called *transition function* or *transition model* $P(s, a, s')$. The function tells the probability of getting into state s' from state s if action a was taken. The executable actions and reachable next states from a specific state are known in most types of MDPs. If an action a is taken in state s which results in state s' a so-called reward can be observed, which is defined by the function $R(s, a, s')$. The transition function is sometimes denoted as T, and the chosen action sometimes is in the subscript, for example $P_a(\dots)$. The goal of the model-based approach is to approximate the transition model P and the reward function R as accurately as possible and find the optimal policy π in order

to collect as much reward as possible. The policy is a function that tells the agent which action to choose at each state.

Model-free algorithms, as opposed to the model-based ones, do not maintain transition functions during execution. These algorithms usually use so called *action-values* or *Q-values* in order to determine the optimal policy. The value of an action in a state is calculated from the maximal possibly obtainable reward in the future if the agent takes that action. The goal of the model-free algorithm is to learn the real Q-values and to learn the optimal policy on them. In most cases the model-free methods acquire the optimal action-values *off-policy*. This means that during the update of the learned Q-values the agent estimates the total discounted future reward for every state-action pair assuming a greedy policy were being followed despite the fact it is not necessarily following a greedy policy. Model-free algorithms come in many forms but the most prevalent ones are mainly from the family of Q-learning methods. However, it must be noted, that although the algorithms in this group do not learn the model of the environment, in most cases they strongly depend on the distinguished handling and storing of states or state-action pairs. Examples for this can be seen below.

Before moving on to the next section, I want to write a few words about the difficulties of the literature research. First, the field of reinforcement learning as a whole is enormous, a comprehensive overview is far too out of scope for this thesis. Second, it was often difficult to separate the exploration strategy from the algorithm itself, so in some cases the exploration strategies are explained by reviewing the algorithm in which it was used.

2.2. Model-based Methods

In this section I describe a few algorithms and algorithm families which are fundamentally model-based.

2.2.1. E^3

The Explicit Explore or Exploit (E^3) algorithm [4] is a very effective method to learn the optimal policy in an MDP. The main idea of the algorithm is that it maintains three separate MDPs: M, $\widehat{M}$ and $\widehat{M}'$. [5]

Before explaining the three models, the notion of *known* and *unknown* state needs to be introduced. A state is *known* if it had been visited at least *m* times, where *m* is given at the beginning of the algorithm. Otherwise a state is *unknown*. The difference between a

known and an *unknown* state, is that the *known* states' transition probabilities and obtainable rewards are more or less well-approximated and are closer to the real values.

Let's see the models. M is the original problem, with its states, transition probabilities and rewards. $\widehat{M}$ consist of a set of states S which is the set of *known* states in M, and a new state s_0. Transition probabilities and rewards of $\widehat{M}$ are the following:

- for every a action $P(s_0, a, s_0) = 1$
- transactions which are between two known states are the same as in M
- transactions which are between a state in the known set and a state outside of it in M, will be redirected to s_0 in $\widehat{M}$
- rewards in $\widehat{M}$ are the same as in M, for s_0 it is 0

$\widehat{M}'$ contains the same states in similar groups, and the same transition probabilities as $\widehat{M}$, but it has different rewards. All states have zero reward, except for s_0 which has R_{max} (it is the upper bound for the theoretically obtainable maximum reward in M).

Without going too deep into the details and the theoretical background, the E^3 algorithm works as follows:

1. The algorithm performs a so called *balanced wandering* at any time when it's not in a *known* state (in S).
2. If a state during the *balanced wandering* has been visited at least *m* times, the state enters the *known* set S, and the balanced wandering ends.
3. When the algorithm reaches a known state, it calculates two optimal policies. $\hat{\pi}$ on $\widehat{M}$ and $\hat{\pi}'$ on $\widehat{M}'$
    a. If $\hat{\pi}$ achieves a return in $\widehat{M}$ that is big enough to exceed a predefined limit, the algorithm will execute that policy for the next T steps on M. This part is called *attempted exploitation*.
    b. If $\hat{\pi}$ is not sufficient, the algorithm will execute $\hat{\pi}'$ for the next T step on M. This part is called *attempted exploration*.
4. If *attempted exploitation* or *attempted exploration* reaches a state which is not in S, the algorithm immediately returns to balanced wandering.

The main idea behind the two policies on the two different MDPs is the following. If the algorithm follows the exploitation strategy $\hat{\pi}$, it will collect almost as much reward as the real optimal policy would collect in M under T steps with high probability. Or if the algorithm follows the exploration strategy $\hat{\pi}'$, it will observe an unknown state in M under T steps with high probability. With this two types of behaviour, the algorithm can maintain a good balance between exploration and exploitation.

2.2.2. R-max

R-max [6] algorithm is one of those older procedures which were not particularly designed for RL, but for more general problems. R-max for example was originally created for zero sum stochastic games which contain the MDP problems as special case. Here it will be explained with MPD terminology and notations. [7]

The algorithm fulfils two basic tasks. First, it tries to learn the true reward function $R(s,a)$ and the true transition model $P(s'|s,a)$. Second, it addresses the exploration-exploitation trade-off.

Handling of this trade-off is done with an interesting idea. First, if the reward for a particular transaction $P(s'|s,a)$ is unknown, the algorithm simply uses the value R_{max} instead. Second, the algorithm adds a fictitious state s_E to the existing states, and if the transition probabilities are unknown for an (s,a) pair, it simply assumes that the transition leads to this fictitious state s_E. This state has unique values when it comes to the transition probability function, and the reward function: $P(s_E|s_E,a)=1$ and $R(s_E,a)=R_{max}$.

R-max starts with the following initializations: let $R(s,a)=R_{max}$ and $P(s_E|s,a)=1$ for all *s* and *a*. The algorithm maintains a Boolean variable for every state signalling whether it is *known* or *unknown*. It is initialized to *unknown* for every state, except for s_E, which is *known*.

In addition, R-max stores the states reached by taking specific actions - basically (s',a) pairs - to every state and the number of times it occurred. It also stores the reward for every state-action pair. These are both initialized to 0.

After choosing an initial state s_0 which is set to *known*, the algorithm repeats the following two steps iteratively:

1. Computes an optimal policy on the MDP and follow it until a new state becomes *known*
2. During the execution of the policy:
   2.1. Updates the reward value for every visited (s,a) pair.
   2.2. Updates the stored (s',a) pairs and their occurrence counters for every visited *s*.

2.3. If there is an (s, a) pair which is visited enough times (calculated from predefined parameters), the algorithm sets that state to *known* and updates its $P(s'|s, a)$ values in the transition model.

As can be seen, R-max is quite simple. It starts with an initial estimate of the model parameters that assumes all states and all actions yield maximal reward and lead to the fictitious state s_E with probability 1. Once the agent has enough information about where some action leads to from some state, it updates the entries associated with this particular state and action in the model. After each model update, the agent recomputes an optimal policy and repeats the above steps.

However, unlike the E^3 algorithm described before, the agent does not need to explicitly decide to do exploration or exploitation. In fact, the agent may never learn an optimal policy or it may follow an optimal policy without knowing that it is optimal.

2.2.3. UCB

UCB (Upper Confidence Bound) exists in numerous forms and it is being used with many modifications. [8] In fact, it is not really an algorithm in the classical sense, it is rather a family or group of reasonably simple algorithms, where the common base is that UCB algorithms realize the so called OFU (Optimism in the Face of Uncertainty) methodology, by maintaining reward distributions over different state-action pairs with the highest possible means. That is, the yet unknown means for different actions are as large as plausibly possible based on the collected experiences. The main reason behind the successful operation of UCB algorithms is that either this optimism is well founded, which means the agent acts optimally, or it is unfounded, which means the agent will take actions with less reward than expected, and soon it will learn the true reward distribution of that action and will not choose it anymore. [9]

For further analysis of the UCB methods, some preliminaries need to be defined. If X is a probabilistic variable observed n times independently, $E[Xi] = 0$ and $\hat{\mu} = \sum_{t=1}^{n} \frac{X_t}{n}$, then following inequation stands:

$$P(\hat{\mu} \geq \varepsilon) \leq exp\left(\frac{-n\varepsilon^2}{2}\right)$$

Equating the right-hand side with δ and solving for ε leads to

$$P\left(\hat{\mu} \geq \sqrt{\frac{2}{n}\log\left(\frac{1}{\delta}\right)}\right) \leq \delta$$

Furthermore, the upper bound for the possible reward in sate *s* for action *a* can be derived from this inequation with two additional quantities. Let $\hat{\mu}_{t-1}(s,a)$ be the observed empirical mean of rewards received in state *s* for taking action *a* up until time step *t*, and let $T_{t-1}(s,a)$ be the number of times action *a* was taken in state *s* up until time step *t*. Then the upper bound for the possible reward in sate *s* for action *a* in time step *t* is

$$\hat{\mu}_{t-1}(s,a) + \sqrt{\frac{2}{T_{t-1}(s,a)}\log\left(\frac{1}{\delta}\right)}$$

From this, an action selection method in state *s* at time *t* can be easily derived:

$$a_t = \begin{cases} argmax_a\left(\hat{\mu}_{t-1}(s,a) + \sqrt{\frac{2}{T_{t-1}(s,a)}\log\left(\frac{1}{\delta}\right)}\right), if\ t > K \\ t, \qquad otherwise \end{cases}$$

The use of *K* means that the algorithm spends the first *K* steps with trying different actions which it had not tried before. Of course, after every action selection, the algorithm stores its observations, and recalculates the corresponding $\hat{\mu}$ and *T* values.

By observing the action selection method, it can be stated, that the algorithm selects an action in two cases: if it looks either worth taking ($\hat{\mu}(s,a)$ is large), or worth exploring ($T(s,a)$ is small). Without any deeper explanation, this behaviour ensures that the UCB algorithm family naturally finds the optimal balance between exploration and exploitation.

2.2.4. Thompson sampling

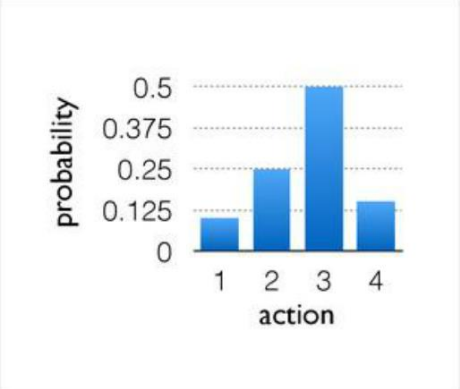
The following method is also a frequently used one and falls just a little behind UCB in terms of effectiveness. Thompson sampling [10] in a nutshell is about estimating posterior distributions of random variables based on experiences in the past, and sampling one of these distributions to achieve the best result.

Formally, this means choosing variable X which solves

$$\max_X \mathbb{E}[X(\theta)]P(\theta|H)$$

where H is the collected experience, θ denotes the parameters of the distribution of X, and $P(\theta|H)$ is the posterior distribution of the parameters based on the experiences.

In reinforcement learning this means selecting an action according the probability that the action is optimal.



1. Figure: Four actions in a problem, with their estimated probability of being optimal. In this case action 3 would be chosen.

This optimality is calculated from the collected rewards in the past, coming from taking those actions. This means one has to know the outcomes of a particular action up to this point, which has two main requirements. First, the agent needs to be able to differentiate between states and state transitions unequivocally, and second, it needs to store some basic information for every action of every state.

2.2.5. MBIE

Model-based Interval Estimation (MBIE) is a generalization of the so-called Interval Estimation (IE) algorithm for the k-armed bandit problem. [11] The algorithm works by maintaining multiple possible MDPs over the real MDP with confidence intervals on the transition function and the reward function. These confidence intervals are calculated separately for every (s, a) pair. When the agent takes an action, it calculates the best action-value with the classic Bellman equation enhanced with the confidence intervals. After taking that action, the agent stores its observations, so they can be used right before the next step to construct the confidence intervals. [12]

For further explanation, some variables need to be defined. $T(s, a, \cdot)$ is the true transition probability vector, $\hat{T}(s, a, \cdot)$ is the empirical distribution of the transition, $\hat{R}(s, a)$ is the sample mean of the observed reward, R_{max} is an upper limit for the value of the reward function, and $n(s, a)$ is the number of times action a has been chosen in state s. δ is the

confidence factor of the model. It is used as δ_T for the confidence interval of the transition function and as δ_R for the confidence interval of the reward function. Last but not least, ε is the allowed maximum error of the model represented by ε_T and ε_R.

The assumed reward for an (s, a) pair is

$$\hat{R}(s,a) + \varepsilon^R_{n(s,a)}$$

where

$$\varepsilon^R_{n(s.a)} := \sqrt{\frac{ln\left(\frac{2}{\delta_R}\right) {R_{max}}^2}{2n(s,a)}}$$

As can be seen, the reward assumption calculation contains an upper confidence interval.

The transition confidence interval is calculated from previous results [13], where it is stated that with at least 1-δ_T probability the L_1 distance between $T(s,a,\cdot)$ and $\hat{T}(s,a,\cdot)$ is at most

$$\varepsilon^T_{n(s,a)} = \sqrt{\frac{2[\ln(2^{|S|}-2) - ln(\delta_T)]}{n(s,a)}}$$

From this result, the confidence interval is

$$CI = \{\tilde{T}(s,a,\cdot) \mid \left\|\tilde{T}(s,a,\cdot) - \hat{T}(s,a,\cdot)\right\|_1 \leq \varepsilon^T_{n(s,a)}\}$$

With these confidence intervals, the MBIE algorithm finds the probability distribution $\tilde{T}(s,a,\cdot)$ within CI, which leads to the policy with the largest value for each state-action pair. The calculation is done by the enhanced version of the Bellman equation mentioned before:

$$\tilde{Q}(s,a) = \tilde{R}(s,a) + \max_{\tilde{T}(s,a,\cdot)\in CI} \gamma \sum_{s'} \tilde{T}(s,a,s') \max_{a'} \tilde{Q}(s',a')$$

After solving this equation, the agent follows a greedy policy with respect on $\tilde{Q}$ to choose the next action.

2.2.6. OIM

The OIM (Optimistic Initial Model) algorithm has some similarities with the R-max algorithm. It starts with an optimistic model, and step by step it becomes more realistic as

new experiences arise. The exploration is driven by the optimism of the model as the unknown states yield significantly larger rewards than the known ones. [14]

The algorithm introduces a new state with reward $R_{max} \coloneqq \max_{s,a,s'} R(s,a,s')$ a.k.a. "garden of Eden" state s_E (this can also be found in the R-max algorithm). Furthermore, it maintains three variables for every state and action: $N_t(s,a)$, $N_t(s,a,s')$ and $C_t(s,a,s')$, which denotes the number of times action *a* was selected in state *s*, number of times transition $s \xrightarrow{a} s'$ occurred, and the sum of rewards for transitions $s \xrightarrow{a} s'$, respectively.

The agent builds an approximate model of the environment using the components above. The model is made up of two main parts. First, the transition function:

$$\hat{P}_t = \frac{N_t(s,a,s')}{N_t(s,a)}$$

Second, the reward function:

$$\hat{R}_t = \frac{C_t(s,a,s')}{N_t(s,a,s')}$$

The Q values calculated for every state-action pair are built up from two components.

$$Q(s,a) = Q^r(s,a) + Q^e(s,a)$$

The first is the value function calculated from the external rewards $\hat{R}$, the second comes from the exploration reward, which is defined as:

$$R^e(s,a,s') \coloneqq \begin{cases} R_{max}, & if\ s' = s_E \\ 0, & if\ s' \neq s_E \end{cases}$$

The algorithm starts with the following initializations: for each state *s* and action *a*, $N_0(s,a) = 1$, $N_0(s,a,s') = 0$, $N_0(s,a,s_E) = 1$, $C_0(s,a,s') = 0$, $C_0(s,a,s_E) = 0$.

After the initialization, the algorithm acts greedily with respect to the calculated Q values, and recalculates the model approximation at every step. Without any further explanation, this recalculation is done by dynamic programming, with the following equations:

$$Q^r_{t+1}(s,a) \coloneqq \sum_{s'} \hat{P}_t(s,a,s')\left(\hat{R}_t(s,a,s') + \gamma Q^r_t(s',a')\right)$$

$$Q^e_{t+1}(s,a) \coloneqq \gamma \sum_{s'} \hat{P}_t(s,a,s') Q^e_t(s',a') + \hat{P}_t(s,a,s_E) Q(s_E,a_E)$$

Where $Q(s_E, a_E)$ represents the best observable action value. The goal of the algorithm is to learn the optimal Q-value function, by approximating the two equations above as accurately as possible.

As can be seen, the optimal policy according to the agent's model will either explore new information that helps to make the model more accurate, or follows a near-optimal path. The extent of optimism regulates the amount of exploration.

2.3. Model-free Methods

In this section I describe a few algorithms and algorithm families which are fundamentally model-free.

2.3.1. ε-greedy

The ε-greedy strategy is one of the oldest methods for exploration, and basically can not be circumvented in case of model-free reinforcement learning. Although, it is the most prevalent method in the family of dithering exploration algorithms, it is very simple and computationally tractable. [8]

The algorithm focuses on the action selection. In state *s* the agent will select an action according to its policy (in most cases greedily) with probability 1-ε or select an action randomly with probability ε. The algorithm has many variants and modifications, but in the end they all come down to this simple step.

Although ε-greedy is simple and seems like it naturally finds the balance between exploration and exploitation, it is not efficient in practice, and can not ignore suboptimal actions which it got acquainted to during the exploration steps.

2.3.2. Boltzmann Exploration

Boltzmann exploration (a.k.a. softmax action selection with Boltzmann distribution) is a simple method but it needs more knowledge about the environment than the ε-greedy strategy. [8] While the latter only works with the list of actions of the current state, Boltzmann exploration integrates the relative action values into its calculations. [14]

With a given action value function $Q(s, a)$ the algorithm selects an action greedily in time *t* from the following multivariate distribution $\pi(a)$:

$$\pi(a) = \frac{exp(\beta Q_t(a))}{\sum_{a' \in \mathcal{A}} \exp(\beta Q_t(a'))}$$

Where $\beta > 0$ controls the greediness of the method (the larger the β the greedier the algorithm).

With this simple extension, it solves the problem of selecting low-rewarding actions on the long run, which difficulty the ε-greedy method can not overcome.

2.3.3. OIV and small negative rewards

The two methods explained here are not standalone algorithms or strategies for exploration, but rather useful techniques to implicitly enhance exploration without directly modifying the RL algorithms. These methods are not unfamiliar for other algorithms, OIV (Optimistic Initial Values) [14] for example is more or less used in both R-max, and OIM algorithms mentioned above.

Instead of assuming the reward value to be unknown or zero for unknown state-action pairs, the OIV method assumes that rewards are immensely large in those cases. By this initialization trick, the agent will experience lower rewards than it expects in every state, so it will keep on trying yet unknown actions because of the assumed high initial rewards. As the agent keeps on trying different actions (some actions will always have a slightly larger action value than the others) the action values slowly converge to the optimal values.

The method of small negative rewards addresses this implicit exploration too, but in a different way. It associates small negative rewards to every state. When the agent visits a state it will slightly worsen its value function if that state otherwise does not have a positive external reward. With this method in the RL algorithm two kinds of states will have high value after a certain amount of time: the high rewarding ones and the rarely visited ones. Ultimately this method urges the agent to explore (just like OIV) and tries to address the problem of finding the balance between exploration and exploitation, especially if a small randomness is introduced to the action selection, for example through ε-greedy.

2.3.4. Delayed Q-learning

The Q-learning algorithm is almost as elementary among model-free reinforcement learning methods as for example ε-greedy is among exploration strategies. It is comprehensive, easy to implement and can be a very effective tool to benchmark or develop RL agents

and algorithms. Q-learning has extensive literature, so instead of adding another brief explanation to the existing ones, I would like to introduce a variant of the algorithm which has some interesting features and has a few common aspects with some of the previously mentioned methods and strategies.

Delayed Q-learning [15] was built upon the simple Q-learning algorithm, so they have a lot in common. Both maintains Q-value estimates for every (s, a) state-action pair ($Q(s, a)$) at every time $= 1,2,3 \ldots$. The agent always acts greedily with respect to the Q-values, so in state *s* at time *t* the next action taken will be $a_{t+1} = \underset{a \epsilon \mathcal{A}}{\operatorname{argmax}}\, Q_t(s, a)$.

Delayed Q-learning maintains an $\epsilon \in (0,1)$ value, an *m* positive integer, a counter $l(s, a)$ and a Boolean flag $Learn(s, a)$. The counter holds the number of times action *a* was taken in state *s*. The $Learn_t(s, a)$ flag indicates whether the agent will attempt to update its $Q(s, a)$ value or not right after taking action *a* in state *s* at time *t*.

The algorithm attempts to perform an update for $Q(s, a)$ at time *t* when the agent manages to collect *m* observations for (s, a) state-action pair, and the $Learn_t(s, a)$ flag is true. $Learn(s, a)$ is set to true, when any of the Q-values are updated, and set to false when an attempted update fails on $Q(s, a)$. All $Learn(s, a)$ values are initialized to true in the beginning of the algorithm.

The algorithm works the same as Q-learning, except for the update rule. At time *t* in state *s*, let $s_{k_1}, s_{k_2}, \cdots, S_{k_m}$ be the last *m* observations on (s, a) pair in time $t = k_1, k_2, \cdots, k_m$ respectively and let r_{k_i} be the reward that is observed at time k_i. The Q-value update at time $t + 1$ given the previously introduced variables will be

$$Q_{t+1}(s, a) = \frac{1}{m} \sum_{i=1}^{m} \left(r_{k_i} + \gamma \max_{a \in \mathcal{A}} Q\left(s_{k_i}, a\right) \right) + \epsilon$$

The update will only be successful if the following equation is satisfied:

$$Q_t(s, a) - \left(\frac{1}{m} \sum_{i=1}^{m} \left(r_{k_i} + \gamma \max_{a \in \mathcal{A}} Q\left(s_{k_i}, a\right) \right) \right) \geq 2\epsilon$$

Otherwise:

$$Q_{t+1}(s, a) = Q_t(s, a)$$

Delayed Q-learning updates its Q-values when it collected enough samples, as opposed to the simple Q-learning algorithm, which does it at every step. This alteration has impressive effects on the learning, such as it lowers the variation with averaging the Q-values, and from some aspects it mitigates the randomness of the algorithm. A further advantage of this method is that it combines this "delay" with the optimistic value function updates introduced by ϵ and the success threshold of the attempted update. Because of the greedy action selection this optimism leads to a more directed exploration and the learning of the near optimal Q-values, just like for example in the previously described R-max algorithm or the OIM method.

3. Related Efforts and Challenges

In this chapter I describe two of the works which successfully combined the technologies introduced in the first chapter. These papers, especially the methods and the approaches, were the basis of my thesis. I built my work around the implemented solutions and experiments introduced in these publications. It is important to note that despite of the theoretical inspiration I got from those papers, my work contains lots of alterations and differences. It is partly because the lack of appropriate hardware[1] and the experience necessary to implement and develop RL agents aided by deep learning.

In the course of the following description, besides the realization of the solution, I focus on the challenges and those difficulties which the authors could not fully overcome.

3.1. Human Level Control through Deep Reinforcement Learning

The first paper was published in Nature with the title 'Human-level control through deep reinforcement learning'. [16] In this work the researchers created an agent, which learned to play 49 of the well-known Atari games [17], most of them better than a human player.

During training, the agent interacted with the environment the same way as a human player would. It only received visual information about the playfield of each game, and could only control the environment with real-time actions adjusted to the step rate. With reinforcement learning terminology this means, that a state s is an image capturing a snapshot of the game, and an action a is a valid move on the controller.

However, in fact deriving a state from a single frame (captured snapshot of the game) is bad practice. There are for example objects in some games, which are "flickering" - that is, some of them are present in even frames while others appear only in odd frames. Besides this, the frames from the games generally need some pre-processing (such as rescaling or adjusting colours) anyway. To address the first problem, a state is created not form one frame but from four consecutive frames put next to each other.

In the action selection process, the agent uses ε-*greedy* method with annealing of ε. This means that at the start of the training the action selection is mostly random, and as the agent moves on with the learning process it becomes more and more conscious.

[1] I used the VGA of my laptop, which is an NVIDIA GeForce GTX 960M, but in the introduced works the researchers used more advanced hardware.

The applied learning algorithm is Q-learning, with some modifications. Q-learning works by using action values as defined in the previous chapter. These action values need to be known in every state. Storing these values explicitly however is not possible because of the immense size of the state space. In order to address this difficulty, function approximation methods are often applied to calculate the action values. In this paper, a non-linear function approximator was used in the form of a deep convolutional neural network. This type of network which calculates Q-values from states and actions are called *Deep Q-Network* (DQN).

During the learning, the agent collects its observations and actions at each time-step t into an experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$, which tuples are then stored in a so-called *replay memory* $D = \{e_0, \dots, e_t\}$. Later on, in every certain number of steps, this replay memory is used to train the neural network. Without deeper explanation, the training of the network is done against the difference between the best action value theoretically available and the current action value given by the network. For simplicity, this will be called loss function from now on, and formally it looks like this:

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\right)^2\right] \qquad \text{(Eq. 3.1.1)}$$

Where θ_i represents the network parameters, and γ is the discount factor. The weights are modified by the gradient of the loss function using RMSProp algorithm. [18]

Using the DQN parameters like that however results in a significant variance in the approximated Q-values, because the loss function changes in every iteration. To solve this problem and to smoothen the sequence of predicted Q-values a little, there are two DQNs being used instead of one with the same structure and hyperparameters. One is called q-network the other is target-network. With this modification, the calculation of the loss at each iteration looks a bit different:

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \qquad \text{(Eq. 3.1.2)}$$

Where θ_i^- are the parameters of the target-network, and are overwritten with θ_i periodically.

This improvement on the loss function calculation reduces the variance in the training, because the difference which is used during the update of the parameters of the q-network

are now calculated using the target-network. The target-network is much less volatile than the originally used q-network, due to previously introduced parameter copying.

An additional part in the algorithm on which improvements were made, is the usage of the replay memory. As explained before, a single piece of experience is generated at every step of the agent, which are then loaded in to the replay memory one after the other. This method leads a large amount of local correlations among the stored data, because consecutive sates looks very similar. To address this difficulty, the agent uniformly samples the replay memory rather than taking sequences from it to create batches for the DQN. This sampling process is called *experience replay*.

3.2. Bootstrapped-DQN

The second paper was published under the title 'Deep Exploration via Bootstrapped DQN'. [19] The work in this paper is based on the previously described algorithm [16], but presents a few fundamental modifications by which it performs better on the same Atari games.

In the previous work, two DQNs was used to calculate the action values. By this idea, the estimations contained much less variance, and the learning curve smoothened. However, the action value estimation method can be improved even further with another simple idea, which is called Double DQN. [20]

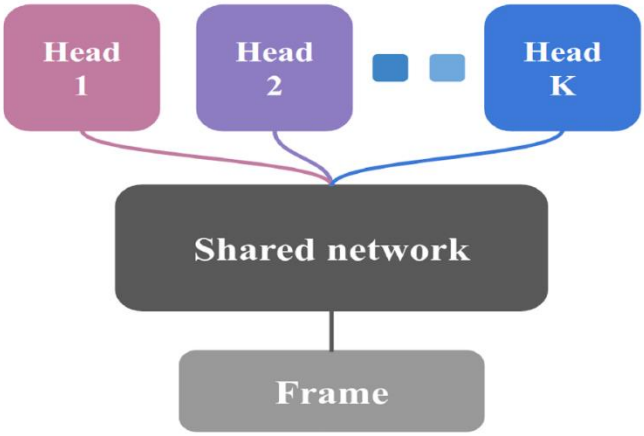
The problem with DQN is that it is often susceptible to overestimation of the action values under certain circumstances. This can be observed in the first paper: in (Eq. 3.1.1) and (Eq. 3.1.2) the same values are used both to select and to evaluate an action. Because of this, there is a high chance of selecting overestimated values, which causes overoptimistic action value updates. However, if the action selection is decoupled from the action evaluation in the aspect of value estimation, this overoptimistic behaviour can almost totally be avoided.

The solution is to change the calculation method of the target action value in the loss function with the following modification of (Eq. 3.1.2):

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma Q(s', \underset{a}{\operatorname{argmax}}\, Q(s', a; \theta_i)\,; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \text{(Eq. 3.2.1.)}$$

With this alteration, picking an action is done according to the action values estimated by the q-network, but the actual value of the chosen action is determined by the target-network. This idea naturally leads to decisions based on values estimated with reduced optimism, because the target-network – depending on the copy frequency - has much less variance in its estimations than the q-network.

Although Double DQN is a significant improvement, the main challenge addressed by this paper is the problem of exploration. In the first paper, although it has very good results, the exploration is done by the ε-greedy method, which is – in practice – dithering, so the agent does not make decisions on exploration consciously. In order to make exploration more planned and be a part of the aware action selection process, the researcher came up with an interesting DQN structure:



2. Figure: Bootstrapped DQN

The network in 2. Figure is called Bootstrapped DQN. It consists of a shared part and multiple heads. It is called bootstrapped because during training the well-known bootstrap method [21] is used to select data samples. Every head is trained on different bootstrapped sub-samples of the data, while the shared part is trained on every data sample.

In practice, it looks like the following. When observing the environment and the current transition, the agent uniformly selects an m head index from $\{1, \cdots, K\}$ and insert the experience into the memory with that head index as $e_t = (s_t, a_t, r_t, s_{t+1}, m_t)$. Note that this index selection happens in every step.

During the execution of the algorithm, before every episode or in every certain step a head is chosen randomly to be used in that period. Inferencing the network is the same as in DQN, but training is a bit different. When assembling the batches, experiences are

picked from those whose stored head index corresponds to the index of the currently used head. This means that every head has a different set of data which they are trained on.

Training and using the network this way results in an uncertainty over the Q-values it gives during an inference. This is because the heads are selected randomly and yield different Q-values on each action of an inferenced state. In addition, the bootstrapped deep neural network naturally implements the concept of "posterior distribution based on past experiences", for it is trained on the samples of previous observations. This means that bootstrapped DQN samples a single Q-value function from its approximate posterior. The agent then follows the policy which is optimal for that sample.

After a little thinking it becomes clear that the bootstrapped method used in this article is basically the implementation of the idea of Thompson sampling in a model-free algorithm, despite the fact that Thompson sampling is fundamentally a model-based method.

4. Implementation

4.1. Used technologies

In this section I provide a brief description of the most important technologies I used for the implementation of the learning algorithms.

4.1.1. Environment

The learning algorithms implemented in the papers described in the previous chapter were trained to play Atari 2600 games. To emulate these games, the researchers used the Arcade Learning Environment (ALE) [22], which contains a wrapper for a classic Atari 2600 game emulator[2].

Besides containing a wrapper, ALE is a framework that implements functionalities for building reinforcement learning agents and provides a common general interface for interacting with the Atari games. Using ALE as an environment, the agent receives the current state of the game as a 160 pixels wide and 210 pixels high image of the game screen with 128 colours. The agent can choose from 18 actions which comes from real Atari games: three positions of the joystick for each axis, and a button. When running, the framework can generate 60 frames per second up to 6000 frames per second.

The problem with ALE is that it only contains Atari 2600 games. These are great challenges to be solved, but in order to develop RL agents I needed problems with less complexity, especially in the beginning of my work.

For my thesis, I chose OpenAI Gym [23] as an RL environment framework. OpenAI Gym is an open source interface and framework under continuous development, with roughly the same attributes and behaviour as ALE. It must be highlighted that because it is open source, anybody can develop an environment for it, which resulted in a vast and colourful environment set. From the classic board games through control tasks up to complex algorithmic challenges, many problems are implemented in it.

The main reasons behind choosing this framework were that on one hand, it provides a very similar interface for the agent as the previously introduced ALE. On the other hand, OpenAI Gym contains much - if not all - of the Atari games from the ALE in addition to

[2] Stella - https://stella-emu.github.io/

many other, simpler reinforcement learning problems, such as the classic mountain car or the inverted pendulum.

4.1.2. Deep Learning Framework

In order to efficiently train and use deep neural networks, I used the TensorFlow [24] toolkit. This is an open source framework, which is able to make numerical computations efficiently and fast. The main reason behind this is that the tool is able to run the calculations on GPUs, which are multiple orders of magnitude faster than normal CPUs.

TensorFlow uses a data flow graph as inner representation for the calculations. The nodes of the graph represent operations, and the edges are for moving the so-called *tensors* between the input and output of the operations. Tensors are dynamic data structures which store data efficiently. The execution of the graph is data driven, which means that an operation is executed as soon as every input is available for it.

This framework is mainly used for building and training deep neural networks, but it is important to note that TensorFlow can be used in many other fields and for many other problems, the only criteria is that the computations must be defined to be compatible with the data flow graph introduced before.

Another important part of TensorFlow which I used for this thesis is the TensorBoard. It is a suite of tools for visualizing the used data flow graph, arbitrary metrics and performance indicators about the execution, or the data itself that is passing through the graph.

4.1.3. Preprocessing Visual Data

In order to transform the image given by OpenAI Gym into a proper state representation suitable for the input of the learning algorithm, I used OpenCV [25]. This is an open source computer vision and machine learning software library.

Although - as its website states - OpenCV has a large amount of algorithms both from the field of computer vision and from the machine learning domain, I only used the library to do the preprocessing steps necessary to feed the snapshots coming from the emulator into the deep neural network used by the learning algorithm.

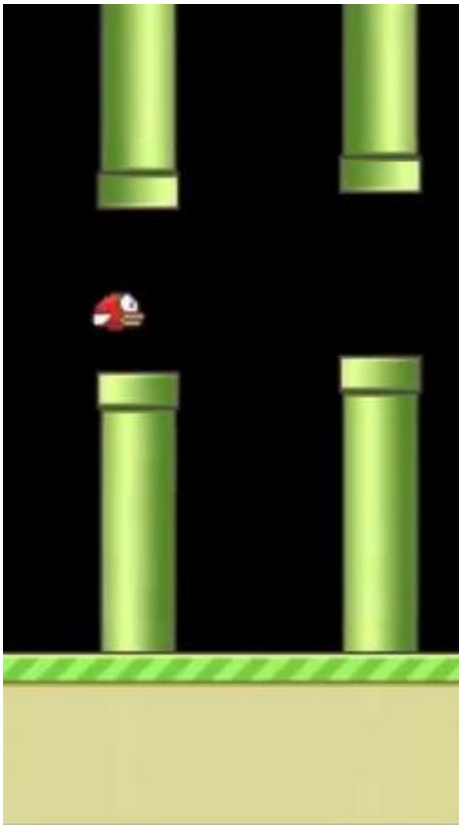
4.2. Preliminaries

In this section I describe those non-technical aspects of my work which undamentally influenced the created algorithms, but not necessary obvious for the reader at first.

4.2.1. Game

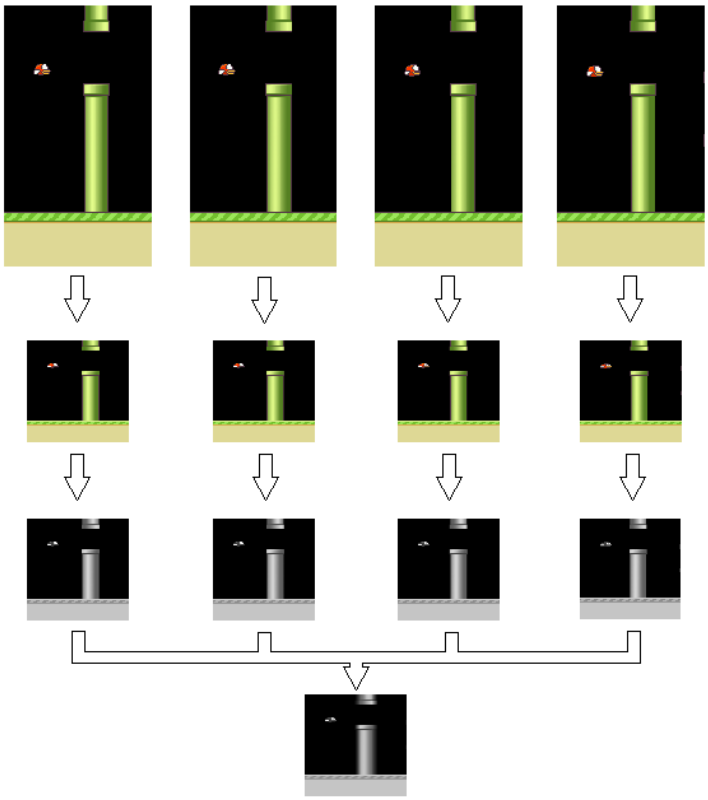
The first such factor is the environment I used. In order the keep the problem simple, and the execution of the algorithm traceable, I chose the well-known game Flappy Bird. This is an easy game with a simple state space. The action space consists of two actions: one for going up, and one for doing nothing.

In the original game, there are two types of background with different shapes and colours. In addition, the bird comes in three different colours and the pipes in two. These features make the game prettier and less boring, however they cause the algorithm to converge much slower. As my goal was to compare methods, the speed of the convergence is irrelevant as long as they are being run under the same circumstances. Therefore, I replaced the background with a black image and kept only one bird and pipe colour, in order to achieve shorter runtimes and faster convergence.



3. Figure: FlappyBird environment with simplified colour palette

During the training, a state in time *t* was constructed by stacking the frames from time *t-3* to *t* on each other. The reward was 0.1 in every non-terminal state, and -1.0 in a terminal state.



4. Figure: The creation of one state representation from frames *t-3*, *t-2*, *t-1* and *t*.

4.2.2. Code

At the beginning of my work I spent a significant amount of time on searching for simple implementations realizing the combination of simple games, reinforcement learning and deep neural networks. As a result, I found two works which I used as a foundation and example for my code base. They can be found on GitHub as [26] and [27].

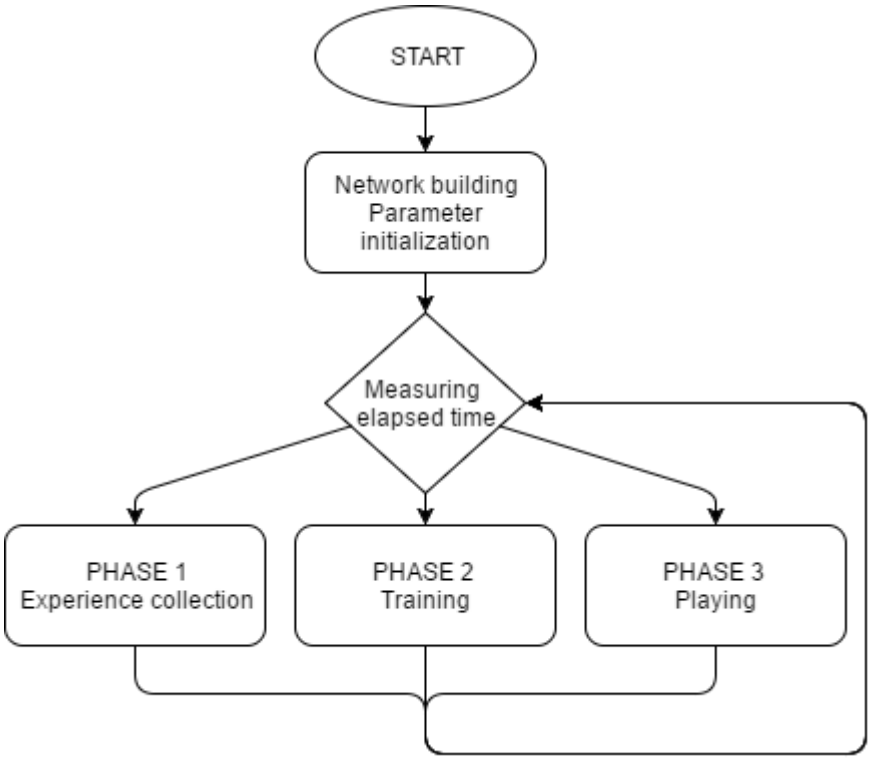
4.3. The Learning Process

4.3.1. Execution flow

During the implementation of the learning algorithms I aimed to keep it clear-cut and straightforward. As can be seen on 5. Figure the frame of the algorithm is a cycle which starts to run after the neural network is built and the parameters are initialised. The type of the executed operations inside the cycle depends on the number of iterations done by the algorithm. According to this, the algorithm has three types of operation mode.

In Phase 1 the agent collects experiences, and fills up its replay memory. In this phase, the action selection is done with the initial value of ε. It can be treated as a random action

selection method, even if the ε-greedy method is not used, because it is done with the randomly initialised neural network, which is not retrained, only inferenced in this phase.



5. Figure: Simplified flow chart of the basic phases in the learning process

Phase 2 is about the training itself. The algorithm keeps on collecting experiences, but unlike Phase 1, it now retrains the neural network at every step. In this phase, the ε parameter is annealed from its initial value to a closing value. These values are provided at the start of the execution.

In Phase 3 the agent uses the trained network with the closing ε value to play the game. In this phase, the agent does not evolve further, it just uses its knowledge acquired before. This phase is for the easy judgement of whether the training was successful or not.

4.3.2. Learning methods

As stated in section 1.5, one of my tasks in this thesis was to test different algorithms with different exploration strategies. In the given solution, I used three different algorithms, in the form of three different implementations.

The three tested methods were a vanilla DQN, a Double DQN, and a Bootstrapped Double DQN. All three methods were described before, see 3.1 and 3.2.

During the running of the algorithms I concentrated on the speed of the convergence rather than the actual performance of the agent. That is, the basis for the comparison of the methods were not the "goodness" of the learnt policy, or the amount of received rewards but the speed of reaching the same level of performance. It does not have to be good, but it has to be approximately identical for all three algorithms.

4.4. Implementations

In this section I describe each implementation separately highlighting their main components, their differences, and their similarities.

4.4.1. DQN

The DQN variant of the algorithm uses a deep convolutional neural network with five layers: three convolutional layers and two fully connected layers.

The input of the neural network consists of an 80x80x4, grayscale image. The first hidden layer convolves 32 filters of 8x8 with stride 4 with the input image. The output is then put through a 2x2 max pooling layer. The second hidden layer convolves 64 filters of 4x4 with stride 2, again followed by a 2x2 max pooling layer. This is followed by a third convolutional layer that convolves 64 filters of 3x3 with stride 1 followed by a 2x2 max pooling again. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The values at this output layer represent the Q-values for each valid action in the given input state.

The action selection is done with the classic ε-greedy strategy at every frame. The value of ε is annealed in Phase 2 as described in section 4.3.1.

The replay memory is realised by a queue which stores experience tuples $e_t = (s_t, a_t, r_t, s_{t+1})$. The agent adds every new experience to the end of the queue. If the queue reaches a certain size, the agent starts to drop elements form beginning after every insertion.

The training of the network is done with the following loss function:

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\right)^2\right] \qquad \text{(Eq. 4.4.1)}$$

So – as explained before - the training of the DQN is done against the difference between its own inferenced values and the ideal targets which are also calculated from its own inferenced values.

The used network training algorithm was the Adam Optimizer [28]. I used the basic version in TensorFlow, mostly with the default parameters.

The high-level description of the algorithm looks like this:

```
Initialise COLLECT and TRAIN variables for the phases of the training
Initialise INIT_EPS and CLOSING_EPS variables for the annealed ε-greedy method
Initialise Q-network with random θ weights
Initialise step counter t to 0
While TRUE:
    With probability ε select a random action a
         Otherwise select a := argmax_a(Q(s,a;θ))
    Reduce ε by (INIT_EPS – CLOSING_EPS) / TRAIN
    Execute action a
    Store experiences (s,a,r,s') in the replay memory
    If t > COLLECT t <= TRAIN:
         Sample a random minibatch of (s,a,r,s') from the replay memory
         Set y_j = { r_j, if s' is a terminal state in the game
                   { r_j + γ max_a' Q(s',a';θ), otherwise      for every batch entry
         Retrain the network by minimizing (y − Q(s,a;θ))^2
    End If
    Increase step counter t
End While
```

4.4.2. Double DQN

In the Double DQN variant of the algorithm I used the same network architecture as in the DQN variant, but I made two significant alterations.

On one hand, I used the network in two copies in the form of a q-network and a target-network (see section 3.1). On the other hand, I used the two networks the way described in section 3.2, so when calculating the loss for the retraining of the network, the next action with the highest value is selected by the q-network, but after that its actual Q-value is estimated by the target-network.

With the modifications mentioned above, the loss calculation looks a bit different:

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma Q(s', \underset{a}{\operatorname{argmax}}\, Q(s', a; \theta_i)\,; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \quad \text{(Eq. 4.4.2)}$$

Aside from these modifications, the DDQN variant looks almost the same as the DQN variant:

```
Initialise COLLECT and TRAIN variables for the phases of the training
Initialise INIT_EPS and CLOSING_EPS variables for the annealed ε-greedy method
Initialise q variant of Q-network with random θ weights
Initialise target variant Q-network with weights θ⁻ = θ
Initialise step counter t to 0
While TRUE:
    With probability ε select a random action a
        Otherwise select a ≔ argmax_a(Q(s,a;θ))
    Reduce ε by (INIT_EPS – CLOSING_EPS) / TRAIN
    Execute action a
    Store experiences (s,a,r,s′) in the replay memory
    If t > COLLECT t <= TRAIN:
        Sample a random minibatch of (s,a,r,s′) from the replay memory
        Set y_i = { r_i, if s′is a terminal state in the game
                  { r_i + γQ(s′, argmax_a′ Q(s′,a′;θ) ; θ⁻), otherwise   for every batch entry
        Retrain the network by minimizing (y − Q(s,a;θ))²
    End If
    After every certain number of steps overwrite θ⁻ with θ
    Increase step counter t
End While
```

4.4.3. Bootstrapped Double DQN

There are three main differences between the Bootstrapped Double DQN variant and the DDQN variant of the algorithm.

The first difference is that both the q-network and the target-network is modified, to have several different heads as depicted in 1. Figure. This means that every combination of shared networks and heads has a q-network and a target-network variant separately, but during the execution only the corresponding variants interact with each other.

The second difference comes from the different experience collections for each head, because it means that the sampling process for retraining the networks needs to be done differently. For this I implemented a separate container, which supports uniform sampling from experiences grouped by their head index.

The third difference is that I used the ε-greedy strategy for the action selection process differently. The annealing of the ε parameter was done in two ways: in the beginning of the TRAIN phase over a smaller number of steps, and like in the previous variants. It is because the goal here was to test the bootstrap method (see 3.2) in combination with the annealed ε-greedy method. As a sign of difference, I marked the annealing time in the BDDQN algorithm with another variable (ANNEAL_TIME).

With the modifications listed above, the BDDQN variant of the algorithm looks like this:

Initialise COLLECT and TRAIN variables for the phases of the training
Initialise HEAD_COUNT variable for the number of network heads
Create a shared network and #HEAD_COUNT amount of network heads for q variant of the Q-network and initialise them with θ random weights
Create a shared network and #HEAD_COUNT amount of network heads for target variant of the Q-network and initialise weights $\theta^- = \theta$
Initialise step counter t to 0
While TRUE:
    After every certain amount of steps chose a $head\_index$ from range 0..HEAD_COUNT
    With probability ε select a random action a
        Otherwise select $a \coloneqq \underset{a}{\operatorname{argmax}}\big(Q(s, a; head\_index, \theta)\big)$
    Reduce ε by (INIT_EPS – CLOSING_EPS) / ANNEAL_TIME
    Execute action a
    Randomize a head index flag m from range 0...HEAD_COUNT for the observation
    Store experiences (s, a, r, s', m) in the replay memory
    If t > COLLECT t <= TRAIN:
        Sample a random minibatch of (s, a, r, s', m) from the replay memory, where $m == head\_index$
        Set $y_i = \begin{cases} r_i, if\ s' is\ a\ terminal\ state\ in\ the\ game \\ r_i + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; head\_index, \theta); head\_index, \theta^-), otherwise \end{cases}$ for every batch entry
        Retrain the network by minimizing $\big(y - Q(s, a; head\_index, \theta)\big)^2$
    End If
    After every certain number of steps overwrite θ^- with θ w.r.t the current $head\_index$
    Increase step counter t
End While

5. Evaluation of the Implemented Algorithms

In this chapter I describe the process of running and evaluating the algorithms, as well as the work done in order to make them fast enough. First, I introduce the experiment parameter setup, then I describe how the algorithms were tuned in order to reduce the run times, and finally, I detail the results of the executions.

5.1. Benchmarking Preliminaries

5.1.1. Goal and Metrics

The goal of the executions was to see how the different algorithms can solve a simple game such as FlappyBird. According to the papers introduced in Chapter 3, I assumed that DDQN will do better than DQN, and BDDQN will outperform both. The basis of my assumption was the following. Firstly, in [16] it is stated that DQN produce better results with more stable Q-values if the Q-values are updated against a target value rather than themselves. Secondly, in [19] the researchers claim and prove that by using BDDQN, a much faster learning can be realised than with a dithering exploration strategy such as ε-greedy.

In order to show these results, I measured three main metrics among many other subsidiary characteristics. The three main metrics were the Q-values, the sum of collected reward per game, and the loss values. Q-values are measured at every action selection, and are always the value of the best action. The collected rewards are measured continuously and if the agent reaches a terminal state, the sum of the collected rewards is stored. The loss values are measured during the retraining of the deep neural networks and are calculated differently in every variant (see (Eq. 3.1.1), (Eq. 3.1.2), and (Eq. 3.2.1.)).

5.1.2. Parameters

The following table contains the parameters which were the same in all three algorithm variants:

Parameter	Value
Discount factor γ for the loss calculation	0.99
Replay memory size	50000
Batch size used for network training	32
Number of steps spent on experience collection (COLLECT)	10000
Number of steps spent on training (TRAIN)	3000000

1. Table: The common parameter values for the algorithms

When ε-greedy strategy was used primarily (DQN and DDQN) the initial value of ε was 0.1 and the closing value was 0.0001. The annealing of ε was done during the TRAIN phase. In BDDQN the initial and the closing values were the same, but the annealing time was different. I experimented with two values. One of them was one third of the length of the TRAIN phase, the other one was the same length.

When the neural networks were used in multiple variants – so in DDQN and BDDQN – the copy frequency at which the overwriting of the target parameters with the q parameters is done is set to 5000 iteration. In both [16] and [19] this parameter was set to 10000, but this problem and its state space is far simpler than most of the Atari 2600 games, so a small amount of variance should not affect the learning curve as much, but can increase the speed of convergence.

During the execution of the BDDQN variant, I used 10 different heads (as in [19]) and I switched heads on the networks every time a terminal state was reached.

5.1.3. Execution

Before I tested the algorithm, I made a few experiments on the run times by changing different components of the DDQN implementation in order to find the fastest combination. I present five of the tried combinations which are the closest to the final form of the algorithms.

Combination	Environment	Total TensorFlow	Rendering
Comb1	wrapped bird	Yes	-
Comb2	gym bird	No	No
Comb3	gym bird	No	Yes
Comb4	gym bird	Yes	Yes
Comb5	gym bird	Yes	No

2. Table: The different combinations in DDQN used to measure run times

I experimented with two environments: with the one that is in OpenAI Gym, and with another one from [27]. The author of the latter wrote a wrapper around the version of the game implemented in the Gym. The result was a Flappy Bird game with the same attributes defined in 4.1.1. However, as opposed to the Gym version, the rendering could not be disabled. The "Rendering" column indicates whether the render was disabled or not. It is important to note, that the gym bird here was used in its original form, the modifications described in 4.1.1 were implemented later.

There were some calculations - such as the calculation of y for a batch – which were done on CPU instead of in TensorFlow on GPU. When "Total TensorFlow" is Yes, that means I relocated every calculation possible to GPU by using TensorFlow.

The measured run times in the combinations defined above for the different code parts and functionalities are described in the next table.

Combination	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**
Comb1	1.16	12.51	0.25	0.09	10.74	8.61	0.85
Comb2	1.31	2.76	1.15	0.04	11.43	9.98	0.87
Comb3	1.12	12.42	1.05	0.07	11.25	9.35	0.83
Comb4	1.37	12.31	1.35	0.09	7.95	9.52	0.86
Comb5	1.17	2.23	0.91	0.07	7.85	8.65	0.81

3. Table: The run times [ms] of the measured code parts and functionalities

Code	**Name**	**Description**
M1	Inferencing	Inferencing the network on one state.
M2	Rendering	Perform one step on the environment (render included, if there is any).
M3	Preprocessing	Process the image given by the environment for the neural network.
M4	Batch sampling	Sample a batch from the replay memory.
M5	Batch calculation	Make the necessary calculations on a batch before retraining the network (not in TensorFlow).
M6	Retraining	Retrain the network with one batch (in TensorFlow).
M7	TensorBoard logging	Log some of the variables and calculated values in TensorBoard.

4. Table: Description of the measured code parts and functionalities

The values in 3. Table are the median of ten consecutive values from ten iterations selected randomly from the TRAIN phase.

As can be seen from the measurement data, the fastest combination was when gym bird was used without render and with all operations calculated in TensorFlow.

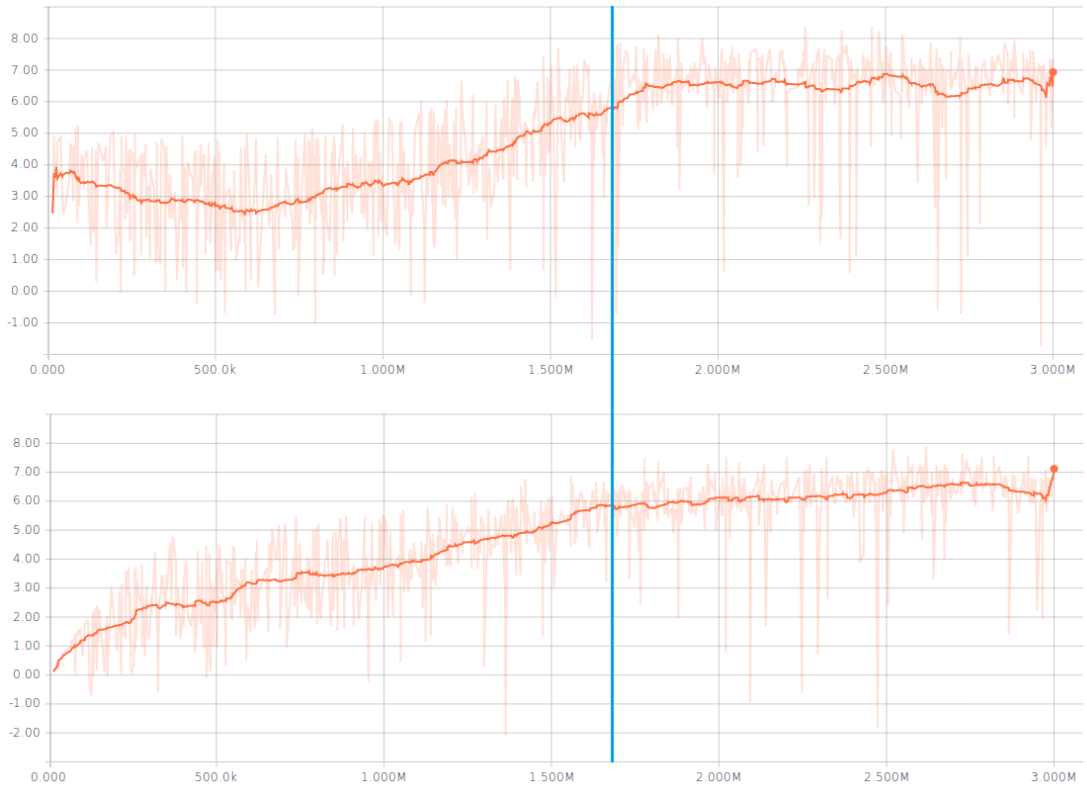
With the information gathered by measuring the different combinations, the final version became the following: the algorithm was run on the gym bird with the modifications of 4.1.1, every calculation was done in TensorFlow, and the rendering of the environment was done only after the TRAIN phase. In this configuration, the simulations ran for approximately 20 hours.

5.2. Results

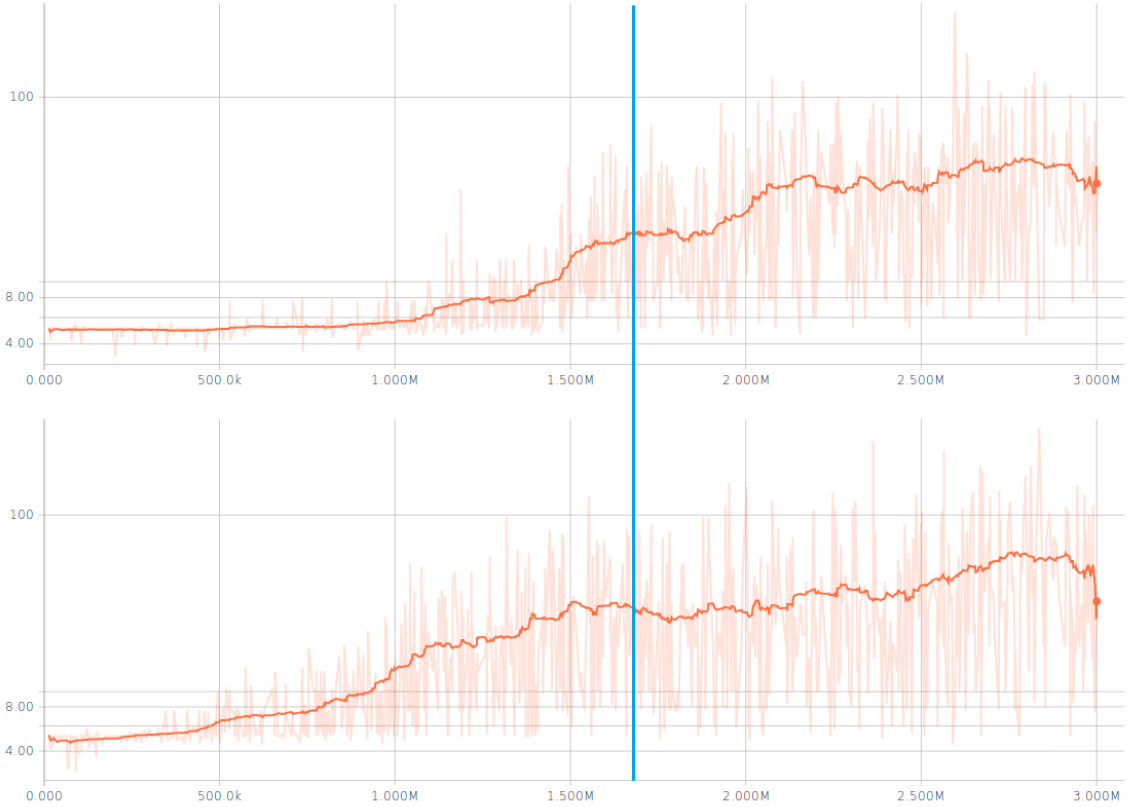
The graphs which I use to show the result were created by TensorBoard. It operates with a sliding-window averaging method. I set the smoothing parameter to 0.6.

5.2.1. DQN vs. DDQN

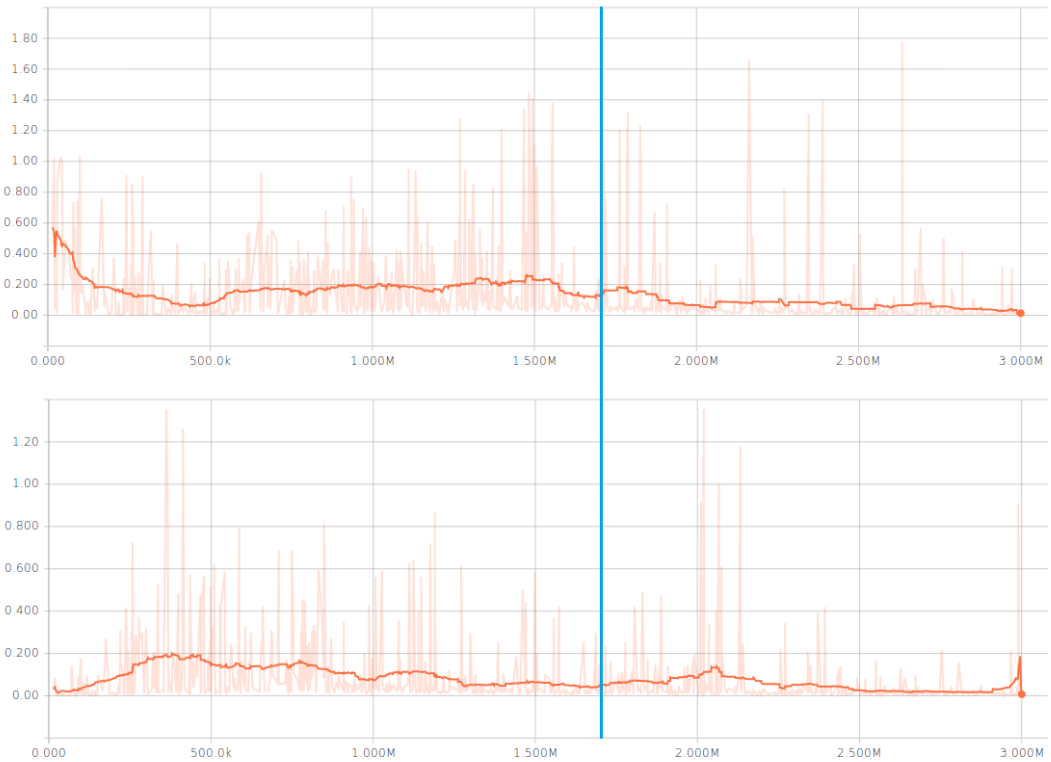
On the following figures, the results of DQN variant (top) and DDQN variant (bottom) can be seen. The depicted values are in order: Q-values, sum of collected rewards, and the result of the loss calculation at every network training iteration. The vertical blue line is roughly at the 1.7Mth step.



6. Figure: The measured Q-values by DQN and DDQN at every action selection.



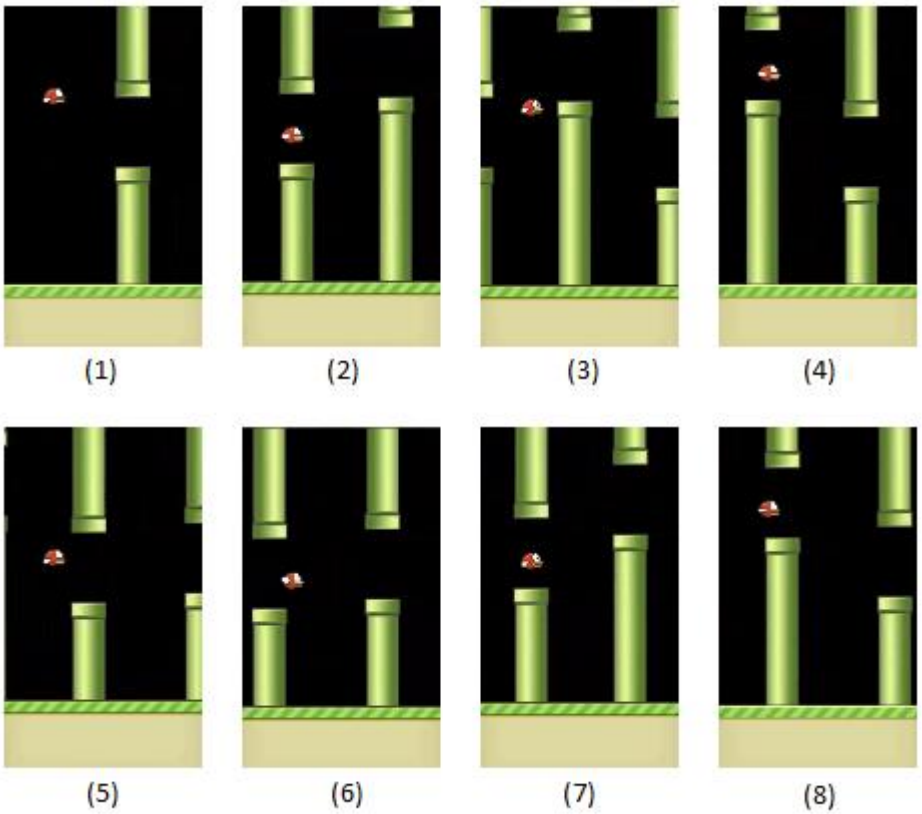
7. Figure: The sum of collected reward in each game by DQN and DDQN (log scale).



8. Figure: The result of loss calculations by DQN and DDQN during the network re-training.

As can be seen on 6. Figure in the interval from the beginning to the blue line DDQN provided better Q-values with less variance, because of the reduced “optimistic bias”. Thanks to the increased stability of the Q-function, the training of the network improved a bit. As depicted in 8. Figure the loss function values of DDQN are smaller, contain less variance, and the shape of the curve looks more realistic. This means that in the beginning the agent gathers lots of new information and experience, so the network improves a lot (higher loss values during training), and as the training goes on, the agent observes fewer unknown states, so the network does not change as much. From 7. Figure it is clear, that on the long run, both agents collected approximately the same amount of reward, but thanks to the more stable early stage, DDQN reaches the same values earlier than DQN.

On the following picture, there are snapshots from a game played by the agent at the end of the training. As it depicts, the agent was successful in learning the policy.



9. Figure: Successfully learnt policy

From this experiment and the depicted results it can be stated that DDQN performed better than DQN in the first part of the execution, when the training was more focused on exploration.

5.2.2. BDDQN

Unfortunately, the BDDQN algorithm could not solve the problem with the tested parameter configurations. In my opinion, there were two main difficulties which caused this.

Firstly, in the end I did not have much time to try more parameter combinations, and modifications in order to see how the running of the algorithm changes.

Secondly, I do not have the hardware required to run the Bootstrapped DDQN for as many steps as the researchers did in [19]. In this paper, the agent was running for 50M steps, which would took me about a week with my current hardware configuration.

6. Conclusions

I did literature research on the more general or nowadays prevalent reinforcement learning algorithms. During the research, I concentrated on the exploration methods these algorithms use or were built on. To summarise this work, I wrote a brief evaluation about the checked algorithms and methods. To efficiently do this, I divided the algorithms into two groups: model-based and model-free methods.

In order to test one of these algorithms, I created a framework, in which I could install multiple reinforcement learning environments, methods for measuring the run times of certain parts of the algorithms, and a tool for monitoring and logging each run.

In this framework, I implemented three distinct algorithms with different Q-function approximation and exploration methods.

And finally, I ran successful experiments with two the implemented algorithms and tested their effectiveness against each other.

7. Future Improvements

During the implementation and testing I saw two fundamental ways to improve the algorithms, and the usage of experience replay and DQN.

The first method of improvement would be the modification of the replay memory. Random sampling – as explained in [16] – proved to be a major advancement, however it is still far from optimal. The main problem is that it does not differentiate between experiences, which results in every experience being equally "good". I think that a few modifications on the replay memory could improve the results significantly. Firstly, storing additional information on experiences about the episodes they came from, such as the length of the game, the sum of collected rewards, whether or not the game resulted in a victory, etc. Secondly, by the reduction of the amount of stored similar states. The whole Q-function approximation concept is based on studying as many kind of states as possible, so it is not necessary to store too many of the same type. It brings a bias to the network training through the random sampling of replay memory no matter what.

The second improvement opportunity would be to calculate how similar an observed state is to the previously seen ones. These values can then lead the exploration, by rewarding actions which result in states that are not as similar to the ones seen before – it would lead the agent to the less explored parts of the state space.

8. Acknowledgements

First, I owe my supervisor István Engedy tremendous amounts of gratitude for the continuous guidance and aid during my work.

I'm deeply grateful to my colleagues at Quanopt Ltd. and to my fellow students at BUTE for helping me solve the technological problems during the implementation and for revising the final written document.

And last but not least, I want to thank my family and friends for their endless support.

References

[1] A. Kurenkov, "A ’Brief’ History of Neural Nets and Deep Learning, Part 3," 24 December 2015. [Online]. Available: http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-3/. [Accessed 3 May 2017].

[2] A. Kurenkov, "A 'Brief' History of Neural Nets and Deep Learning, Part 4," 24 December 2015. [Online]. Available: http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-4/. [Accessed 3 May 2017].

[3] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, London: The MIT Press, 1998.

[4] M. Kearns and S. Singh, "Near-Optimal Reinforcement Learning in Polynomial Time," *Machine Learning*, no. 49, pp. 209-232, 2002.

[5] "Decision-Making in Large-Scale Systems, Handout #12," MIT, Spring 2004. [Online]. Available: https://ocw.mit.edu/courses/mechanical-engineering/2-997-decision-making-in-large-scale-systems-spring-2004/lecture-notes/lec_9_v1.pdf. [Accessed 3 May 2017].

[6] R. I. Brafman and M. Tennenholtz, "R-max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning," *Journal of Machine Learning Research*, no. 3, pp. 213-231, 2002.

[7] C. Guestrin, "Reinforcement Learning, Machine Learning – CSE446," Universtiy of Washington, 5 June 2013. [Online]. Available: https://courses.cs.washington.edu/courses/cse446/13sp/slides/mdps-rl.pdf. [Accessed 3 May 2017].

[8] C. Szepesvári, "Algorithms for Reinforcement Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 4, no. 1, pp. 1-103, 2010.

[9] T. Lattimore, "The Upper Confidence Bound Algorithm," 18 szeptember 2016. [Online]. Available: http://banditalgs.com/2016/09/18/the-upper-confidence-bound-algorithm/. [Accessed 13 május 2017].

[10] W. R. Thompson, "On the likelihood of one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3/4, pp. 285-294, December 1933.

[11] L. P. Kaelbling, Learning in embedded systems, Cambridge, MA: The MIT Press, 1993.

[12] A. L. Strehl and M. L. Littman, "An analysis of model-based Interval Estimation for Markov Decision Processes," *Journal of Computer and System Sciences,* vol. 74, no. 8, pp. 1309-1331, 2008.

[13] T. Weissman, E. Ordentlich, G. Seroussi, S. Verdu and M. J. Weinberger, "Inequalities for the L1 Deviation of the Empirical Distribution," HP Laboratories, Palo Alto, 2003.

[14] I. Szita and A. Lőrincz, "The many faces of optimism: a unifying approach," *Proceedings of the 25th International Conference on Machine Learning (ICML-08),* pp. 1048-1055, 2008.

[15] A. L. Strehl, L. Li, E. Wiewioria, J. Langford and M. L. Littman, "PAC Model-Free Reinforcement Learning," *Proceedings of the 23rd International Conference,* pp. 881-888, 2006.

[16] V. Mnih, K. Kavukcuoglu and D. Silver, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529-541, 26 February 2015.

[17] 2600online.com, "2600 online," [Online]. Available: http://www.2600online.com. [Accessed 15 May 2017].

[18] G. Hinton, N. Srivastava and K. Swersky, "Neural Networks for Machine Learning, Lecture 6, Overview of mini-batch gradient descent," University of Toronto,

Department of Computer Science, 9 October 2012. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. [Accessed 3 May 2017].

[19] I. Osband, C. Blundell, A. Pritzel és B. Van Roy, „Deep Exploration via Bootstrapped DQN," in *NIPS*, Barcelona, 2016.

[20] H. van Hasselt, A. Guez és D. Silver, „Deep Reinforcement Learning with Double Q-learning," in *AAAI*, Phoenix, 2016.

[21] B. Efron és R. J. Tibshirani, An Introduction to the Bootstrap, CRC Press, 1994.

[22] M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253-279, 2013.

[23] "OpenAI Gym," OpenAI, [Online]. Available: https://gym.openai.com. [Accessed 15 May 2017].

[24] "TensorFlow," Google, [Online]. Available: https://www.tensorflow.org. [Accessed 15 May 2017].

[25] "OpenCV," [Online]. Available: http://opencv.org. [Accessed 15 May 2017].

[26] "Deep Q Learning for ATARI using Tensorflow," 17 March 2016. [Online]. Available: https://github.com/gliese581gg/DQN_tensorflow. [Accessed 16 May 2017].

[27] „Using Deep Q-Network to Learn How To Play Flappy Bird," 28 February 2017. [Online]. Available: https://github.com/yenchenlin/DeepLearningFlappyBird. [Hozzáférés dátuma: 16 May 2017].

[28] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *3rd International Conference for Learning Representations*, San Diego, 2015.