Boosted Deep Q-Network

Boosted Deep Q-Network
Bachelor-Thesis von Jeremy Eric Tschirner aus Kassel
Tag der Einreichung:

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: M.Sc. Samuele Tosatto

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Boosted Deep Q-Network
Boosted Deep Q-Network

Vorgelegte Bachelor-Thesis von Jeremy Eric Tschirner aus Kassel

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: M.Sc. Samuele Tosatto

Tag der Einreichung:

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

Abstract

Deep reinforcement learning achieved outstanding results in the last years, since it has emerged from reinforcement learning combined with the capabilities of deep neural networks. This thesis is about Boosted Deep Q-Network (BDQN), a novel deep reinforcement learning algorithm that employs *gradient boosting* to estimate the action-value function in reinforcement learning problems. The aim with this approach is to meet some of the major challenges still present in the field. On the one hand, they consist of the necessary model complexity for high-dimensional problems and moreover, of the requirement to make efficiently use of the environment's data. BDQN was evaluated empirically first on two standard control problems, and then on one more complex environment with a high-dimensional observation space. In comparison to Deep Q-Network (DQN), the algorithm performed competitively on the standard tasks using a smaller model complexity, but had serious problems learning a reasonable policy in the complex environment.

Zusammenfassung

Deep-Reinforcement-Learning hat in den letzten Jahren hervorragende Ergebnisse erzielt, seit es aus Reinforcement-Learning in Verbindung mit den Fähigkeiten tiefer neuraler Netze entstanden ist. In dieser Thesis geht es um Boosted Deep Q-Network (BDQN), einen neuartigen Deep-Reinforcement-Learning Algorithmus, welcher *gradient boosting* verwendet, um die Action-Value Funktion in Reinforcement-Learning Problemen anzunähern. Ziel dieses Ansatzes ist es, einige der großen Herausforderungen zu meistern, die auf diesem Gebiet noch bestehen. Diese bestehen zum einen aus der benötigten Modellkomplexität für hochdimensionale Probleme, und zum anderen aus der Anforderung, Daten aus der Umgebung möglichst effizient zu nutzen. BDQN wurde empirisch zunächst auf zwei Standard-Kontrollproblemen und dann in einer komplexeren Umgebung mit hochdimensionalem Beobachtungsraum ausgewertet. Im Vergleich zu Deep Q-Network (DQN) hat der Algorithmus mit geringerer Modellkomplexität kompetitiv die Standardaufgaben gelöst, hatte aber starke Probleme beim Erlernen einer vernünftigen Strategie in der komplexen Umgebung.

Acknowledgments

I want to thank my supervisor Samuele Tosatto for his patience and support. During my work on this thesis, I learned a lot from him and his guidance helped me to successfully get through this process.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
ANN	Artificial Neural Network
BDQN	Boosted Deep Q-Network
DP	Dynamic Programming
DQN	Deep Q-Network
i.i.d.	independently and identically distributed
MDP	Markov Decision Process
SGD	Stochastic Gradient Descent
TD	Temporal Difference

List of Symbols

Notation	Description
$\boldsymbol{\theta}$	Weight vector of an ANN

1 Introduction

During the last few years, the field of artificial intelligence (AI) has evolved rapidly and became more and more integrated in our daily lives. One of its primary goals is to develop autonomous systems that are able to adapt to new situations and learn independently through interaction with their surrounding environment. This important learning paradigm is based on behaviorist psychology, and connected with mathematical formalisms from optimal control known as the *reinforcement learning* framework [1].
In the past, reinforcement learning methods could be applied successfully to various domains like robotics [2][3], game playing [4] and even communication [5]. However, many of these methods were limited to very specific situations and low-dimensional spaces and thus hardly scalable to more general real-world problems. With *deep learning*, promising possibilities to overcome these limitations have emerged in the last years. Equipped with the ability to find compact representations of high-dimensional data (e.g. visual input), *deep neural networks* significantly improved many areas in machine learning [6]. The arising *deep reinforcement learning* scene was accompanied by a huge success and breakthrough in 2015. With the release of DQN, the company DeepMind showed that an algorithm could learn to play a wide range of Atari 2600 video games above human level directly from raw image pixel input using a convolutional deep neural network [7]. Across the games, the networks architecture and parameters stayed the same, with only few changes and prior knowledge.
Another outstanding result was achieved one year later, when *AlphaGo* managed to defeat the current human world champion in Go, a board game with a much larger state space than for example chess [8]. Instead of using handcrafted features and human domain knowledge, AlphaGo is a system composed of neural networks relying only on deep reinforcement learning and tree search procedures.

1.1 Motivation

All these recent advances in machine learning seem to be promising steps towards constructing general AI systems where human domain knowledge and feature design are becoming less important. Unfortunately, most of these approaches are still impractical and scale badly with respect to the observation space dimension. This issue is known as the *curse of dimensionality*. To achieve a reasonable accuracy, the number of required samples and amount of computational effort grows exponentially with the observation space dimension.
This thesis provides an approach to address some of these problems. More precisely a modified version of DQN where *gradient boosting* [9] is used to approximate the action-value function. BDQN allows a sequential approximation with simpler models that require less computational effort and thus should theoretically bring more efficiency to the learning process. An additional analysis will refer to the data efficiency and possible improvements through BDQN in this regard. Data efficiency is one of the main limitation of current reinforcement learning algorithms especially in real-world environments (e.g. autonomous driving), where obtaining samples through trial and error comes with a high cost.

1.2 Related Work

Currently there are mainly two different works including boosting directly applied to reinforcement learning algorithms. In the first study, Tosatto et al. [10] introduced an iterative off-line algorithm, called *Boosted Fitted Q-Iteration*. Supported through a complete finite-sample error analysis, this paper empirically demonstrates how boosting can be exploited to successfully approximate complex value functions using neural networks or extra-trees and outperform the standard Fitted Q-Iteration while using a lower model complexity. The second work from Abel et al. [11] provides a variant of boosting combined with an exploration strategy that is able to perform consistently in complex domains like the sandbox video game Minecraft. The algorithm is based on the model-free reinforcement learning technique called *Q-Learning* and uses regression trees as base models.
Besides that, a variety of methods have been proposed to improve the performance of deep reinforcement learning algorithms and in particular DQN. Wiering et al. [12] describe a general approach of applying ensemble methods to reinforcement learning algorithms with the aim to enhance the learning speed and final performance. They show that a combination of different algorithms can improve over the result of the best single algorithm by using majority voting. However, each algorithm itself was unchanged and the multiple learned value functions combined to derive a final policy for the agent.

Several other approaches are concerned with increasing the sample efficiency. Lee et al. [13] proposed a way to effectively propagate state values to the previous states of the same episode. By sequentially sampling backwards through an entire episode, errors generated by consecutive updates of correlated states could be reduced. In line with the previous approach, Schaul et al. [14] came up with yet another way experience replay could be used. Instead of the random sampling strategy, non-uniform probabilities were assigned to the sample transitions. These probabilities reflect the information value of each transition with the result that the learning could be done on the most important samples (*Prioritized Experience Replay*). In another study, Van Hasselt et al. [15][16] found a way to obviate the overestimation bias, which sometimes can harm the performance of DQN. This adaptation, namely *Double Q-Learning* not only led to a reduction of the overestimation but also increased the overall stability and performance of the algorithm. Both, Double Q-Learning and Prioritized Experience Replay will be further explained and their application to BDQN will be investigated in this thesis.

2 Foundations

The following section provides background information for the individual components on which the algorithms presented in this thesis are based on.

2.1 Reinforcement Learning

The idea of reinforcement learning is to teach some agent or learner to act in an optimal way. This is done by allowing it to try various actions in the surrounding world and receive rewards. The goal for the agent is to find an optimal policy in order to maximize these rewards.
Solving the reinforcement learning problem can be very hard despite the fact that it can be formulated in a simple way. As shown in Figure 2.1 the two key components are *agent* and *environment*. The *agent* executes an action based on the observation of the *environments* state. The *environment* then responds to this *action* by changing its state and providing an immediate feedback in form of a *reward*.
Commonly the algorithmic implementation of machine learning methods is divided in two big areas namely *supervised* and *unsupervised* learning. However reinforcement learning is different and can not be classified as one of them. A fundamental characteristic is the absence of a teacher or supervisor who provides labeled examples. Instead, the agent is following a trial and error strategy and has to discover the valuable behavior by itself through the reward signal. Because of this time sequential interaction, the data is strongly correlated and not i.i.d. for the most part. In this dynamic system the agents actions heavily affect the subsequent data it receives. Reinforcement learning also differs from unsupervised learning as it is not the goal for the agent to find hidden structure in the data but to maximize a certain reward signal.

2.1.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is a mathematical framework which formally describes the interaction between an agent and the environment in a reinforcement learning problem.
It can be defined using the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mu_0)$ where $\mathcal{S}$ denotes the set of states, $\mathcal{A}$ the set of possible actions and $\mathcal{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$ the probability of observing state $\boldsymbol{s}' \in \mathcal{S}$ after applying action $\boldsymbol{a} \in \mathcal{A}$ in the current state $\boldsymbol{s} \in \mathcal{S}$.
$\mathcal{R}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$ is the immediate expected reward received after transitioning from state $\boldsymbol{s} \in \mathcal{S}$ to state $\boldsymbol{s}' \in \mathcal{S}$ due to action $\boldsymbol{a} \in \mathcal{A}$. The initial state distribution $\mu_0(\boldsymbol{s})$ indicates the probability for each state to be the initial state and $\gamma \in [0,1]$ is a discount factor which represents the trade-off between immediate and future rewards.
At a given time step t the decision maker or agent observes a specific state $\boldsymbol{s}_t \in \mathcal{S}$ of the environment and selects an action $\boldsymbol{a}_t \in \mathcal{A}$ following a certain policy π. Policies provide a mapping from state to action space $\mathcal{S} \rightarrow \mathcal{A}$ and can either be *deterministic* ($\pi(\boldsymbol{s}) = \boldsymbol{a}$) or *stochastic* ($\pi(\boldsymbol{a}|\boldsymbol{s}) = \mathbb{P}[A_t = \boldsymbol{a}|S_t = \boldsymbol{s}]$). As a result of its action the agent receives a numerical reward r_t while the environment transitions to the new state $\boldsymbol{s}_{t+1} = \boldsymbol{s}'$ according to the transition dynamics. The objective of the agent is to find an optimal policy π^* which maximizes the expected discounted return over all time steps $R = \sum_{t=0}^{\infty} \gamma^t r_t$. Considering an infinite time horizon the discount factor is necessary for convergence of the sum.

$$\mathcal{P}_t(\boldsymbol{s}'|\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}, \dots) = \mathcal{P}_t(\boldsymbol{s}'|\boldsymbol{s}_t, \boldsymbol{a}_t). \tag{2.1}$$

If $\mathcal{S}$ and $\mathcal{A}$ are finite the MDP is also said to be finite. Furthermore a MDP satisfies the *Markov property*, namely that transition dynamics only depend on the current state and action, not on the preceding sequence (see Eq. 2.1).

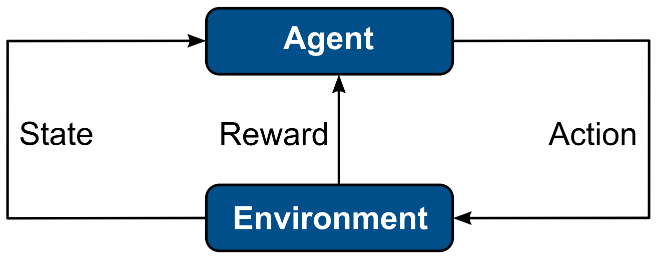


Figure 2.1: Interaction between agent and environment in a reinforcement learning setup. The agent observes a state of the environment and performs an action which leads to a reward signal and a transition of the environment to a new state. Adapted from [1].

2.1.2 Solution Methods

Most approaches to solve a reinforcement learning problem contain a calculation or estimation of *value functions*. Based on the expected return these functions provide a measurement for how valuable a state or a state-action pair is for the agent following a certain policy π. The *state value function* $V^\pi(s)$ (Eq. 2.2) describes the value to be in a given state s and the *state-action value function* $Q^\pi(s,a)$ (Eq. 2.3) describes the value to take a certain action a while being in a given state s. As both transition function and policy can be stochastic, the value corresponds to the expected return when following a certain policy π.

$$V^\pi(\boldsymbol{s}) = \mathbb{E}_\pi[R_t|\boldsymbol{s}_t = \boldsymbol{s}] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|\boldsymbol{s}_t = \boldsymbol{s}\right] \tag{2.2}$$

$$Q^\pi(\boldsymbol{s},\boldsymbol{a}) = \mathbb{E}_\pi[R_t|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}\right] \tag{2.3}$$

$$\text{with } r_i \sim \mathcal{R}(\boldsymbol{s}_i,\boldsymbol{a}_i),\ \boldsymbol{a}_i = \pi(\boldsymbol{s}_i),\ \boldsymbol{s}_{i+1} \sim \mathcal{P}(\boldsymbol{s}_i,\boldsymbol{a}_i), \text{ and } \boldsymbol{s}_0 \sim \mu_0 \tag{2.4}$$

Value functions which assign the largest expected return achievable by an optimal policy are called *optimal value functions* ($V^*(\boldsymbol{s}) = V^{\pi^*}(\boldsymbol{s})$ and $Q^*(\boldsymbol{s},\boldsymbol{a}) = Q^{\pi^*}(\boldsymbol{s},\boldsymbol{a})$). Determining these optimal value functions or policies is the goal when solving a reinforcement learning problem. The most important equations which are used for solving are the *Bellman equations*

$$\begin{aligned} V^\pi(\boldsymbol{s}) &= \sum_{\boldsymbol{a}}\pi(\boldsymbol{a}|\boldsymbol{s})\sum_{\boldsymbol{s}'}\mathcal{P}^{\boldsymbol{a}}_{\boldsymbol{ss}'}[\mathcal{R}^{\boldsymbol{a}}_{\boldsymbol{ss}'} + \gamma V(\boldsymbol{s}')] \\ Q^\pi(\boldsymbol{s},\boldsymbol{a}) &= \sum_{\boldsymbol{s}'}\mathcal{P}^{\boldsymbol{a}}_{\boldsymbol{ss}'}[\mathcal{R}^{\boldsymbol{a}}_{\boldsymbol{ss}'} + \gamma\sum_{\boldsymbol{a}'}\pi(\boldsymbol{a}'|\boldsymbol{s}')Q^\pi(\boldsymbol{s}',\boldsymbol{a}')], \end{aligned}$$

where $\mathcal{P}^{\boldsymbol{a}}_{\boldsymbol{ss}'}$ is the transition probability and $\mathcal{R}^{\boldsymbol{a}}_{\boldsymbol{ss}'}$ the expected reward for ending up in state $\boldsymbol{s}'$ after starting in state $\boldsymbol{s}$ and taking the action $\boldsymbol{a}$. These equations build up a relation between values of different states. Therefore, value functions can be improved by *bootstrapping*, i.e. current values of the estimate can be used to improve the estimate of the value function.

Dynamic Programming

If the state transition function $\mathcal{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$ and the reward function $\mathcal{R}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$ of the underlying MDP are both known, this model knowledge can be used to acquire the optimal value function or policy. Most of the methods being used at this point fall in the category of Dynamic Programming (DP) which is a general algorithmic method for solving complex problems by reducing them into smaller subproblems. In general, DPs algorithms work with updated rules based on the Bellman equations to iteratively improve the desired value functions and policies. Starting with an arbitrary policy π the according state-value function V^π can be computed with iterative *policy evaluation*, where the old values are replaced sequentially by the new values over the whole state space per iteration (Eq. 2.5). Based on this value function, the policy π can be improved following the *policy improvement* step (Eq. 2.6). Alternating policy evaluation and policy improvement leads to monotonically improving policies and value functions and is called *policy iteration*.

$$V^\pi(\boldsymbol{s}) \leftarrow \sum_{\boldsymbol{s}'}\mathcal{P}^{\pi(\boldsymbol{s})}_{\boldsymbol{ss}'}[\mathcal{R}^{\pi(\boldsymbol{s})}_{\boldsymbol{ss}'} + \gamma V^\pi(\boldsymbol{s}')] \tag{2.5}$$

$$\pi'(\boldsymbol{s}) \leftarrow \arg\max_{\boldsymbol{a}}\sum_{\boldsymbol{s}'}\mathcal{P}^{\boldsymbol{a}}_{\boldsymbol{ss}'}[\mathcal{R}^{\boldsymbol{a}}_{\boldsymbol{ss}'} + \gamma V^\pi(\boldsymbol{s}')] \tag{2.6}$$

Since the only objective is to find an optimal policy it is also possible to directly re-define the policy before policy evaluation converges in each iteration. One special case is called *value iteration*, where policy improvement (see Eq. 2.8) is done after each policy evaluation step (see Eq. 2.7). This method often takes less iterations when the policy converges to the optimal policy before the value function.

$$V(\boldsymbol{s}) \leftarrow \max_{\boldsymbol{a}}\sum_{\boldsymbol{s}'}\mathcal{P}^{\boldsymbol{a}}_{\boldsymbol{ss}'}[\mathcal{R}^{\boldsymbol{a}}_{\boldsymbol{ss}'} + \gamma V(\boldsymbol{s}')] \tag{2.7}$$

$$\pi(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}}\sum_{\boldsymbol{s}'}\mathcal{P}^{\boldsymbol{a}}_{\boldsymbol{ss}'}[\mathcal{R}^{\boldsymbol{a}}_{\boldsymbol{ss}'} + \gamma V(\boldsymbol{s}')] \tag{2.8}$$

DP includes the fundamental approaches to solve the reinforcement learning problem. However mostly it is not directly applicable to real world problems because it requires perfect model knowledge and is computationally expensive.

Model Free Approaches

Another way of finding the optimal policy is directly learning the value function without making any assumptions on the model. In these *model free* approaches the environment is unknown and the agent has to explore it in order to collect samples. Based on these samples the value function can be updated. For a given transition $(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')$ the current estimate of the value function $V(\boldsymbol{s})$ can be compared with the one-time step prediction according to the bellman equation. This Temporal Difference (TD) error δ is then used to update the value function

$$V(\boldsymbol{s}) \leftarrow V(\boldsymbol{s}) + \alpha\delta \text{ with } \delta = r + \gamma V(\boldsymbol{s}') - V(\boldsymbol{s}),$$

where α is a step-size parameter which controls the magnitude of the updates. If the underlying MDP is deterministic, α can be set to 1, otherwise it should be less than 1 serving as an average in the case of a stochastic MDP. The discount factor γ is trading off immediate and long term reward as already described in section 2.1.1.

One of the most classic examples is called *Q-Learning* [17], a TD control algorithm which can solve any finite MDP as already proved in 1992 [18]. Q-Learning directly estimates the optimal Q-values of each state-action pair by iterating the bellman equation with

$$Q(\boldsymbol{s}, \boldsymbol{a}) = r + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}', \boldsymbol{a}') \tag{2.9}$$

For equation 2.9 the environment is considered to be *deterministic* so that the transition probabilities can be omitted. Q-Learning builds the basis for DQN and BDQN which will be described in chapter 3.

2.1.3 Value Function Approximation

Previously value functions where considered as a table assigning a value for each state or state-action pair. However most of the time the number of states and actions is so large that it is not feasible to store values for all of them. In this case it is necessary to generalize from a small subset of the state space to create a good approximation for the value function describing the whole state space. There are many possibilities of function approximation but Artificial Neural Networks (ANNs) have been largely employed for reinforcement learning in recent years. They seem to be particularly effective in high-dimensional state-space, since they can autonomously extract the relevant features which is important in many cases e.g. exploiting space correlation in raw pixel images. On the other side, ANNs are data-inefficient and often lead to unstable results especially combined with online reinforcement learning.

2.2 Artificial Neural Networks

ANNs are parameterized non-linear functions. Inspired by biological neural networks they consist of interconnected nodes (artificial neurons) usually organized in layers. Each neuron processes incoming signals and then transmits a resulting signal to its subsequent neurons. Commonly ANNs are used for function approximation in regression or classification tasks. Training ANNs includes an adaptation of their parameters to produce some sort of desired output from a given input.

The most elementary ANN, called *perceptron* was first described in 1958 by Rosenblatt [19]. To calculate its output, the input values are directly multiplied by their corresponding network parameters and summed up. This weighted sum is fed into an activation function which already provides the output of the network. An example of such a single layer *perceptron* can be seen in Figure 2.2. The model described above, however is not able to approximate any function. In order to guarantee an ANN to be a universal function approximator, it is required to have at least one hidden layer combined with a non-constant, bounded and monotonically increasing continuous activation function.

In recent years it became very popular to choose models with a large number of hidden layers, as employing these structures in ANNs allows to obtain a much higher level of complexity leading to outstanding performance in e.g. computer vision tasks [20]. A major drawback of this practice, known by the name *deep learning*, is that such complex models with a lot of parameters require extensive computational resources and huge amounts of data to be processed.

Activation functions in general should be non-linear and differentiable almost everywhere to allow a non-linear mapping for the ANN and efficient gradient-based algorithms for the training process. Three of the most popular activation functions are shown in Figure 2.3.

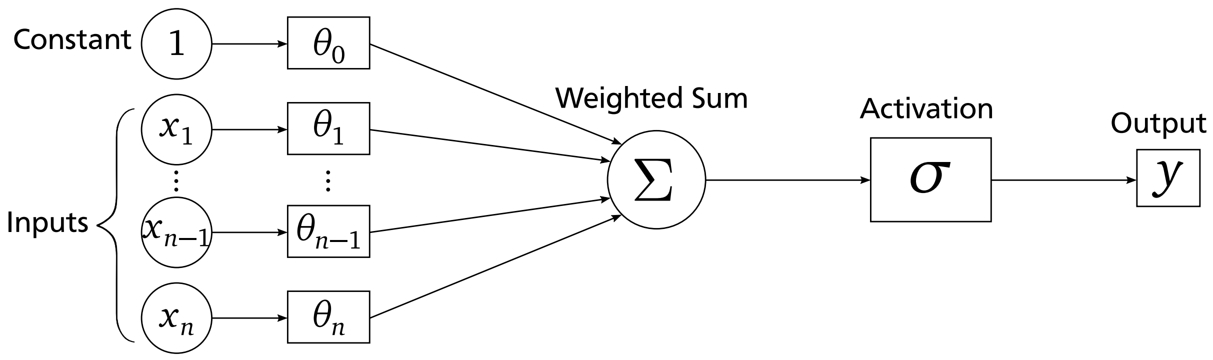


Figure 2.2: Schematic representation of a perceptron. The output is calculated by a non-linear activation function applied to the weighted sum of the inputs: $y(\boldsymbol{x},\boldsymbol{\theta}) = \sigma\big(\sum_{i=1}^{n} x_i\theta_i + \theta_0\big)$. The parameter θ_0 corresponds to the bias, a unit without any incoming connections which allows a translation of the sum. Figure adapted from [21].

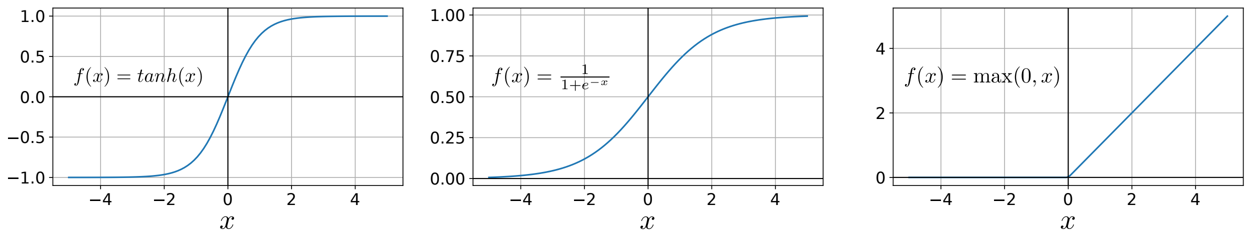


Figure 2.3: Three common activation functions (left to right) hyperbolic tangent, sigmoid function and rectifier. Due to its simplicity and excellent performance rectifier became the most popular activation function for deep neural networks [22].

2.2.1 Network Training

Given a set of input vectors $\boldsymbol{x}_n$ and a set of target output vectors $\boldsymbol{t}_n$ the objective of a neural network is to approximate the underlying function $f(\boldsymbol{x}_n) = \boldsymbol{t}_n$ in the case of regression. A common error measurement is the squared difference between actual and desired output vector of the network. The goal is to find an optimal value for each weight in the network where the error is smallest.
Gradient descent is a popular optimization technique which is used to improve the accuracy of a models prediction. The idea is to start with a random initialization of the parameters and proceed with iteratively updating them in the direction of the steepest descent towards the minimum error value which corresponds to the negative gradient of the error function $\Delta\boldsymbol{\theta} \propto -\nabla(E)$. The gradient includes all the partial derivatives w.r.t each individual network parameter. To calculate these partial derivatives the common approach is *backpropagation*. This method had a huge impact in 1986 when Rumelhart et al. demonstrated for the first time that it can be used to train neural networks efficiently [23].
Backpropagation essentially makes use of the chain rule for the derivation of two or more composed functions $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$. ANNs can be seen as complex nested functions where each layer can be represented as a single function whose inputs are a weight vector and the outputs of the previous layer. An example network with three input units, two output units and one hidden layer is shown in Fig. 2.4. Starting with the squared error function

$$E(\boldsymbol{\theta}) = \frac{1}{2}\sum_{n=1}^{N}\left|\left|y(\boldsymbol{x}_n,\boldsymbol{\theta}) - \boldsymbol{t}_n\right|\right|^2$$

the partial derivatives w.r.t the parameters can be calculated according to the chain rule

$$\frac{\partial E}{\partial \theta_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial a_j}\frac{\partial a_j}{\partial \theta_{ij}}$$

where a_j denotes the linear combination before the activation, o_j the output and $\sigma(\cdot)$ the activation function. These terms lead to a recursive formula describing the error signal δ of each neuron starting with the last layer

$$\frac{\partial E}{\partial \theta_{ij}} = \delta_j o_i \quad \text{with} \quad \delta_j = \begin{cases} \sigma'(a_j)(o_j - t_j) & \text{if } j \text{ is an output neuron} \\ \sigma'(a_j)\sum_k \delta_k\theta_{jk} & \text{if } j \text{ is a hidden neuron} \end{cases}$$

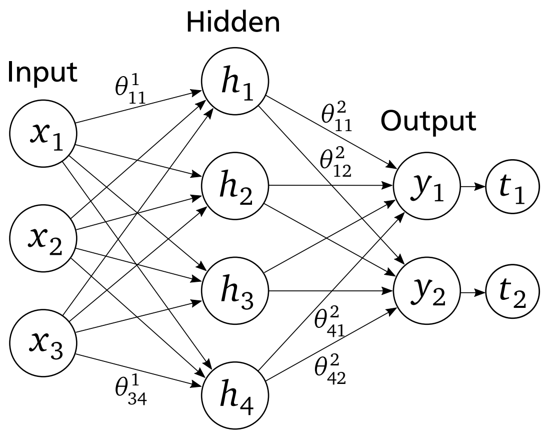


Figure 2.4: Simple ANN with one hidden layer. All neurons are fully connected, the corresponding parameters are represented by the edges. The parameters are identified by using the notation θ^k_{ij} where k denotes the current layer, i the number of the preceding neuron and j the number of the subsequent neuron. Each layer can be described as a unique function. To keep clarity, the bias units for the hidden layer and the output were omitted in this graph.

where k indicates all subsequent neurons after j. With the last layer finished the formula can be applied to the second to last layer and so on propagating the errors backwards through the network.
With all partial derivatives been calculated, the parameters can be adjusted according to a gradient descent step

$$\theta_{ij} \leftarrow \theta_{ij} - \alpha \frac{\partial E}{\partial \theta_{ij}}.$$

The parameter α is called *learning rate* and determines the magnitude of the parameter change. Obviously it can be computationally expensive to calculate the gradient w.r.t all available training data in a large ANN. Stochastic Gradient Descent (SGD) is a common method to reduce the time it takes to perform an update by just taking one training example into account at each step. Adaptive Moment Estimation (ADAM) [24] is a state of the art algorithm which combines SGD and adaptive learning rates and is used for the experiments in this thesis.

2.3 Boosting

Boosting is a machine learning technique with the aim to reduce the estimation bias and create strong learners by combining several simple models. The idea is that each simple model has to perform just slightly better than a random guess, leading to a powerful yet efficient learner. The most widely used boosting meta-algorithm is Ada-Boost [25].

2.3.1 Gradient Boosting

Gradient boosting [9] can be seen as a generalization of boosting for regression and classification problems with arbitrary differentiable loss functions. The model $F_m(x)$ is built in a sequential way

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x)$$

where at each iteration a new simple model $h_m(x)$ is chosen to minimize a certain loss function $L(\cdot)$ given the current model F_{m-1}

$$h_m = \arg\min_h \sum_{i=1}^{n} L(y_i, F_{m-1} + h(x_i)).$$

If the loss function is the squared difference of the targets y_i and the predicted values, the method is called *L2-Boosting*.

3 Implemented Methods

This chapter includes a detailed description of the two algorithms DQN and BDQN, the main subject of this study. The experimental setup, implementation details and results are presented and discussed afterwards.

3.1 Deep Q-Network (DQN)

DQN is a model-free reinforcement learning algorithm that combines Q-Learning with non-linear value function approximation. Its release in 2015 led to a huge revival of the deep reinforcement learning scene showing that human-level performance could be achieved on a set of Atari 2600 games [7].
Like with any model-free approach the agent learns in a sequential way through interaction with its environment (online learning). According to a certain policy $\pi(\boldsymbol{s})$ the action is chosen based on the observed state of the environment. After receiving the reward signal and the environments next state the Q-function can be updated. In DQN the optimal Q-Function $Q^*(\boldsymbol{s},\boldsymbol{a})$ is approximated with an ANN called *Q-Network*

$$Q^*(\boldsymbol{s},\boldsymbol{a}) \approx Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}).$$

Based on the given inputs (state $\boldsymbol{s}$ and action $\boldsymbol{a}$) the Q-Network predicts the corresponding Q-Value through a forward pass. The target for updating the network parameters $\boldsymbol{\theta}$ is calculated each time step based on the current observation which includes the reward signal r and the new state $\boldsymbol{s}'$ of the environment. Together with the previous state $\boldsymbol{s}$ and the executed action $\boldsymbol{a}$ this *transition* $(\boldsymbol{s},\boldsymbol{a},r,\boldsymbol{s}')$ is used to update the Q-Network.

$$E(\boldsymbol{\theta}) = \Big(\underbrace{r + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}',\boldsymbol{a}'|\boldsymbol{\theta})}_{\text{target}} - \underbrace{Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta})}_{\text{prediction}}\Big)^2 \tag{3.1}$$

At first the target is calculated following the Bellman equation (see Equation 2.9). The error can then be obtained with the squared difference between the current prediction and the target. Finally the network parameters are updated according to a gradient descent step on the error.

3.1.1 Target Network

One problem which occurs during the learning are instabilities caused by the changing targets. The parameters are updated based on the error between the current estimate $Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta})$ and the target which itself depends on the network parameters $\boldsymbol{\theta}$ (see Equation 3.1). This often leads to oscillation or divergence of the parameters and thus the policy.
To avoid this, a different set of fixed parameters $\boldsymbol{\theta}'$ can be kept for generating the target.

$$E(\boldsymbol{\theta}) = \Big(r + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}',\boldsymbol{a}'|\boldsymbol{\theta}') - Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta})\Big)^2 \tag{3.2}$$

Equation 3.2 shows how the error is calculated using the *target network* with fixed parameters $\boldsymbol{\theta}'$. After a certain number of updates the fixed parameters can be replaced by the new parameters $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$. This way the target changes delayed in time, therefore the dependency is interrupted resulting in a more stable learning behavior.

3.1.2 Experience Replay

Another big issue with online-learning can be that samples are highly correlated and dependent on the agents behavior policy. This can lead to parameters diverging or ending up in a poor local minimum. Furthermore as every sample transition is only used once for training and then discarded, the agent tends to "forget" already learned behavior from the past (re-learning problem). *Experience replay* [26] is a successful approach to solve the re-learning problem. Instead of learning just with the most recent sample, each observed transition can be stored in a replay memory. The parameters are updated on a whole minibatch of experiences which is drawn randomly from the stored samples.
This randomization reduces the variance of the updates, preventing the divergence of the parameters. Additionally it leads to a more efficient use of the data since each sample is potentially used many times. Figure 3.1 shows the flowchart of the complete DQN algorithm.

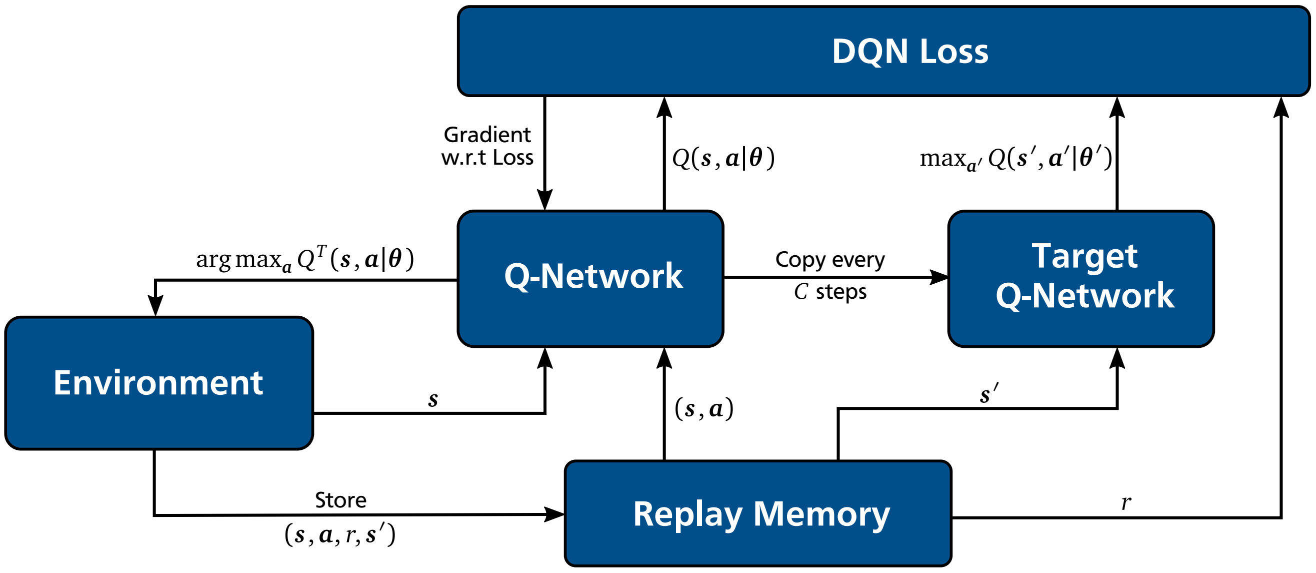


Figure 3.1: DQN with experience replay and target network. Each transition is stored in the replay memory buffer. The loss is calculated on a random mini-batch of transitions as the error between the current predictions and the calculated targets. The parameters of the Q-Network are updated w.r.t the gradient of the loss. Every C steps the parameters are copied to the target Q-Network.

3.1.3 Exploration vs. Exploitation

Choosing which action to take during learning is a crucial point in online reinforcement learning. A simple approach would be to always take the action with the highest value according to the current value function (greedy policy). However this *exploitation* may lead the agent to a poor local optimum and prevent it from seeing more promising states. To reach more valuable states the agent should execute different actions from time to time (*exploration*). Ensuring sufficient exploration, in order to get a trade-off between these two different behaviors, the *ϵ-greedy policy* is commonly used. Following this policy the agent always chooses the action with the maximum value and with a certain probability ϵ a random action. The complete DQN algorithm with experience replay and target network is shown in Algorithm 1.

Algorithm 1: DQN, adapted from Mnih et al [7]

```
1  Initialize replay memory $\mathcal{D}$
2  Initialize Q-Network with random parameters $\boldsymbol{\theta}$
3  Initialize Target Q-Network with random parameters $\boldsymbol{\theta}' = \boldsymbol{\theta}$
4  for $i$ = 1 to $N$ do
5      for $t$ = 1 to $C$ do
6          Select action $\boldsymbol{a}_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(\boldsymbol{s}_t, \boldsymbol{a}_t|\boldsymbol{\theta}) & \text{otherwise} \end{cases}$
7          Execute action $\boldsymbol{a}_t$ and observe $r_t$ and next state $\boldsymbol{s}_{t+1}$
8          Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $\mathcal{D}$
9          Sample random minibatch of transitions $(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}_{j+1})$ from $\mathcal{D}$
10         Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(\boldsymbol{s}_{j+1}, \boldsymbol{a}'|\boldsymbol{\theta}') & \text{otherwise} \end{cases}$
11         Perform a gradient descent step on $(y_j - Q(\boldsymbol{s}_j, \boldsymbol{a}_j|\boldsymbol{\theta}))^2$ w.r.t the network parameters $\boldsymbol{\theta}$
12     Set $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$
13 return: $\pi(\boldsymbol{s}) = \arg\max_a Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta})$
```

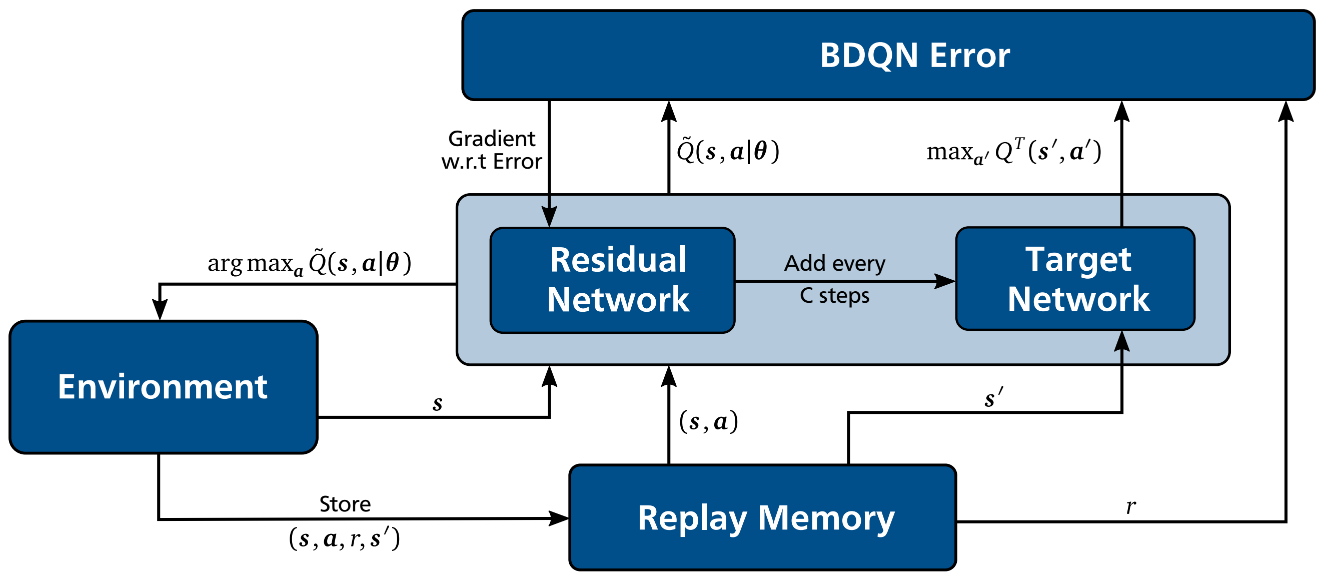


Figure 3.2: BDQN with an additive model to approximate the Q-Function. The parameter of the residual network are updated according to the difference between the target network output and the current prediction.

3.2 Boosted Deep Q-Network (BDQN)

BDQN is an adaptation of DQN where boosting is used to approximate the optimal Q-Function. Instead of approximating the function directly in each step, BDQN works with an additive model $\tilde{Q}$. The components of this model are a fixed target network Q^T and a residual network Q^R. A corresponding graph is depicted in Figure 3.2.
$\tilde{Q}$ is built iteratively following the gradient boosting approach described in section 2.3.1. At first, $\tilde{Q}$ is used to decide which action the agent should take in the environment. Similar to DQN, the resulting transition is stored in a replay memory from which a whole set of transitions (s,a,r,s') can be sampled for generating the target values which are again calculated through a one-step prediction from the target network

$$y = r + \gamma \max_{a'} Q^T(s',a').$$

In each update step these targets are used to get an estimate of the *Bellman residual* ϱ which can be approximated through the current TD-error

$$\varrho = r + \gamma \sum_{s'} \mathcal{P}(s'|s,a) \max_{a'} Q(s',a') - Q(s,a) \approx r + \gamma \max_{a'} Q^T(s',a') - Q^T(s,a).$$

The residual network Q^R is used to approximate the Bellman residual and can be trained using gradient descent. As a function of the network parameters θ, the error is calculated with the squared difference between the residual network and the estimate of the Bellman residual

$$E(\theta) = \Big(\underbrace{r + \gamma \max_{a'} Q^T(s',a') \overbrace{- Q^T(s,a)}_{}}_{\approx\varrho} \overbrace{- Q^R(s,a|\theta)}^{-\tilde{Q}} \Big)^2.$$

After a certain amount of update steps the residual network and the target network are added together to form the new target network. The complete BDQN algorithm is shown in Algorithm 2.

Algorithm 2: BDQN

```
1  Initialize replay memory $\mathcal{D}$
2  Initialize additive model $\tilde{Q}$ with fixed network $Q^T$ and residual network $Q^R$ with random parameters $\boldsymbol{\theta}$
3  for $i$ = 1 to $N$ do
4      for $t$ = 1 to $C$ do
5          Select action $\boldsymbol{a}_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_{\boldsymbol{a}} \tilde{Q}(\boldsymbol{s}_t, \boldsymbol{a}|\boldsymbol{\theta}) & \text{otherwise} \end{cases}$
6          Execute action $\boldsymbol{a}_t$ and observe $r_t$ and next state $\boldsymbol{s}_{t+1}$
7          Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $\mathcal{D}$
8          Sample random minibatch of transitions $(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}_{j+1})$ from $\mathcal{D}$
9          Set $\boldsymbol{y}_j = \begin{cases} r_j - Q^T(\boldsymbol{s}_j, \boldsymbol{a}_j) & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{\boldsymbol{a}'} Q^T(\boldsymbol{s}_{j+1}, \boldsymbol{a}') - Q^T(\boldsymbol{s}_j, \boldsymbol{a}_j) & \text{otherwise} \end{cases}$
10         Perform a gradient descent step on $(\boldsymbol{y}_j - Q^R(\boldsymbol{s}_j, \boldsymbol{a}_j|\boldsymbol{\theta}))^2$ w.r.t the residual network parameters $\boldsymbol{\theta}$
11     Set $Q^T \leftarrow Q^T + Q^R$
12     Initialize new residual network $Q^R$ with parameters $\boldsymbol{\theta}$
13 return: $\pi(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} Q^T(\boldsymbol{s}, \boldsymbol{a})$
```

4 Empirical Results

Before reinforcement learning algorithms can be applied to real-world settings like robotics or industrial control systems, it is essential to analyze their behavior through extensive testing. Simulated environments can be very useful for this purpose, preventing any kind of damage or dangerous situations. Furthermore, a direct application to the real system often is impractical or even impossible due to cost and time effort.
This chapter provides an overview on the technical details of the implementation, the environments and the achieved results.

4.1 Implementation Details

All algorithms and necessary routines for this thesis were implemented with Python 2.7 [27] as it provides a good trade-off between performance and complexity. Equipped with several helpful libraries for scientific computing (e.g. NumPy) and visualizing results (Matplotlib), Python is well suited for prototyping and experimenting in the domain of Machine Learning. To allow fast and efficient neural network computation all models were created using the Tensorflow [28] open source framework for Python. It supports GPU computation and already includes a large set of optimizers and efficient built-in routines for numerical computation which makes it really useful when developing computationally expensive methods (e.g. deep learning).

4.1.1 Model Architecture

The models used to represent the Q-Functions were designed as fully connected ANNs. It turned out that one hidden layer was sufficient to yield good results, so only the number of neurons in this hidden layer was changed during the experiments. As expected, hidden neurons with a rectifier activation function (ReLU [22]) showed the best overall performance. For the last layer, the activation was always kept linear which is usual when it comes to regression problems. The gradient descent for the parameter updates is done with the Adam optimizer [24] using Huber loss [29] which yields a more stable learning process as it is less sensitive to outliers in data than the squared error loss. Apart from the learning rate which can be found in table 4.1, all other parameters were left at default values.
Originally the Q-Function maps an input state and one action to its according Q-Value. When the action and target values are generated, several evaluations of the Q-Function have to be made in order to calculate all values which are necessary to apply the *max* or *argmax* operator. In case of ANNs being used as Q-Function approximators these evaluations can be computationally expensive because a feed forward pass is required for every value. A simple way to avoid this problem and reduce the number of evaluations is to change the architecture of the ANN. With a mapping from input state to Q-Values for each possible action, one forward pass is sufficient to get all required values. This leads to a much faster and more efficient algorithm. Figure 4.1 shows the difference between the two possible architectures.

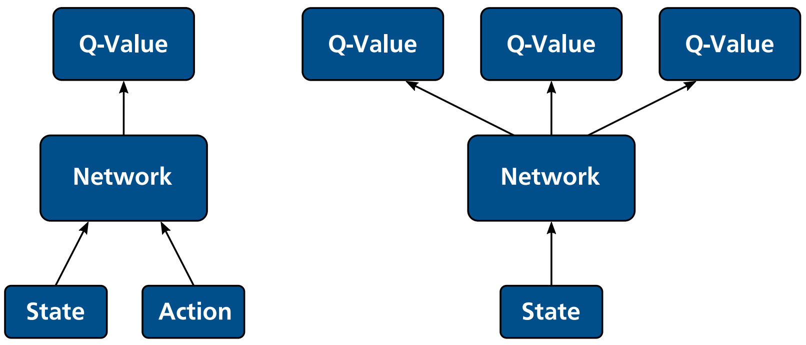


Figure 4.1: Two different Q-Network architectures. The left network takes state and action as input and outputs the corresponding value for that action while the network on the right generates values for each possible action from a given state. As all Q-Values are necessary to apply the *max* and *argmax* operator, the architecture on the right is preferable because a single forward pass is sufficient to generate the values.

For BDQN an addition of the target network Q^T and the residual network Q^R is necessary. A simple way to implement this would be to keep the residual model and and the target network separated and just add both outputs together for the evaluation of the whole model. However this is computationally inefficient as with each boosting iteration a new residual model has to be stored and kept separately. A more elegant and suitable approach is to combine the residual parameter with the target model parameter in one model and use the properties of linear combination to wrap the addition of the two models into a single forward pass.

Let N be the number of training examples (batchsize), D the state dimension, H_i the number of hidden neurons in the i-th layer. Starting with the first layer, the input matrix $\boldsymbol{s} \in \mathbb{R}^{N\times D}$ is multiplied by its respective parameter matrix which is a simple horizontal concatenation of the fixed target parameters $\boldsymbol{\theta}_1^T \in \mathbb{R}^{D\times H_i}$ and the new residual parameters $\boldsymbol{\theta}_1^R \in \mathbb{R}^{D\times H_i}$. After the equally concatenated bias parameter $\boldsymbol{b}_1^T \in \mathbb{R}^{D\times H_i}$ and $\boldsymbol{b}_1^R \in \mathbb{R}^{D\times H_i}$ are added, the activation function $\sigma_1(\cdot)$ is applied to generate the output of the first layer $\boldsymbol{l}_1$ (see Eq. 4.1).

$$\boldsymbol{l}_1 = \sigma_1\left(\begin{bmatrix} \boldsymbol{s}_1 \\ \vdots \\ \boldsymbol{s}_N \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{\theta}_1^T & \boldsymbol{\theta}_1^R \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_1^T & \boldsymbol{b}_1^R \end{bmatrix}\right) \in \mathbb{R}^{N\times(2\times H_1)} \tag{4.1}$$

For every next layer $\boldsymbol{l}_i$ the output of the previous layer $\boldsymbol{l}_{i-1}$ is taken as input. To fit the dimensional constraints for the multiplication and keep each result separated, the parameters now have to be concatenated diagonally (see Eq. 4.2).

$$\boldsymbol{l}_i = \sigma_i\left(\boldsymbol{l}_{i-1} \cdot \begin{bmatrix} \boldsymbol{\theta}_i^T & 0 \\ 0 & \boldsymbol{\theta}_i^R \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_i^T & \boldsymbol{b}_i^R \end{bmatrix}\right) \in \mathbb{R}^{N\times(2\times H_i)} \tag{4.2}$$

If i is the last layer, the number of hidden neurons H_i is just the number of outputs for the whole network (number of different actions values (see 4.1)). The output of this layer now contains the target network and the residual network output $\boldsymbol{y}^T$ and $\boldsymbol{y}^R$ concatenated horizontally. They can be simply added together to get the desired output. Of course with each boosting iteration the size of the target model grows as we concatenate the new residual parameters to its parameters to get the new target network for the next iteration. However the concatenation can still be done as described above just with a larger target parameter matrix. The output then can always be generated with one forward pass regardless of the number of concatenated models.

4.1.2 Environments

Originally the DQN algorithm was evaluated on a set of Atari 2600 games from the Arcade Learning Environment [30]. Using convolutional ANNs the control policies were learned directly from the high-dimensional raw pixel input. However, this requires a large amount of computational resources and training time as a result of the environments complexity and high-dimensional state space. In this thesis, the algorithms were evaluated and compared on much simpler toy problems where the agent has direct access to the important state information without the need for preprocessing any input. The problems were taken from the OpenAI Gym toolkit [31], which offers a large set of standardized simulation environments. For this thesis, two control problems and one more complex environment were chosen to evaluate the algorithms on.

The *Swing-Up Pendulum* is a classic non linear control problem from the Reinforcement Learning literature. The task is to perform a swing up on the underactuated pendulum and keep it in an upright position with the least effort. At each time step the agent can fully observe the state of the pendulum which consists of information about angle and angular velocity of the rod. The action space is the continuous joint effort but as the Q-Networks only output a discrete amount of action values for each state, the applied actions were limited to two actions (highest and lowest possible force). The model of the pendulum is shown in Figure 4.2a. The reward signal (Equation 4.3) depends on the angular deviation θ, the angular velocity $\dot{\theta}$ and the magnitude of the applied action a. During the experiments, it turned out that a regularized reward signal with $r = 10$ led to much better results. This regularization was only applied during the learning, the evaluation scores were unaffected.

$$R = -(\theta^2 + 0.1 \cdot \dot{\theta}^2 + 0.001 \cdot a^2)/r \tag{4.3}$$

In the second control problem called *Mountain Car*, the task is to get an under powered car to the top of a hill. The agent can observe the position and velocity of the car while pushing it either to the right, to the left or do nothing. For each time step the environment provides a reward signal of -1 until the goal position on the hill is reached. At this point the reward signal is 0 and the episode ends. This problem can be particularly hard to learn as the reward is really sparse. The agent has to build momentum to finally get to the goal while the reward does not give any information about how good the current approach is. Figure 4.2b shows an example of this environment.

The last environment, called *Lunar Lander* has a much higher complexity. With the goal to land a spaceship at a desired target on the ground without crashing, the agent can control the ships thrusters to maneuver it around. States include the landers position (s_0 and s_1) and velocity (s_2 and s_3) in the two-dimensional space as well as its angular position and velocity (s_4 and s_5). Additional information for each leg whether it has contact to the ground is given as a boolean value (s_6 and s_7). The agent has the option to fire the main engine, one of the side engines or do nothing. Firing two engines at the same time is not possible. At each time step the reward signal is based on the landers euclidean distance to the landing zone, its velocity, angular deviation and if it has contact to the ground.

$$\begin{aligned} r_t &= -100 \cdot \sqrt{(s_0^2 + s_1^2)} - 100 \cdot \sqrt{s_2^2 + s_3^2)} - 100 \cdot |s_4| + 10 \cdot s_6 + 10 \cdot s_7 \\ R_t &= \begin{cases} -0.3 \cdot e_m - 0.03 \cdot e_s & \text{if } r_{t-1} = 0 \\ r_t - r_{t-1} - 0.3 \cdot e_m - 0.03 \cdot e_s & \text{otherwise} \end{cases} \end{aligned} \tag{4.4}$$

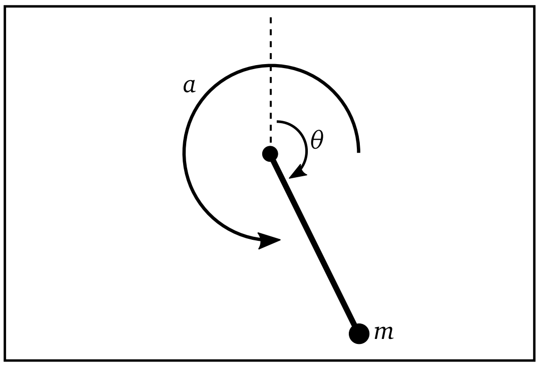
The reward is then calculated as the difference between the signal from the current and the last time step where additionally points are subtracted for firing the engines (e_m and e_s, see Equation 4.4). An episode ends if the lander crashes ($R_t = R_t - 100$) or successfully comes to rest on the ground ($R_t = R_t + 100$). Figure 4.2c shows an example time step in the environment.

4.1.3 Hyperparameter

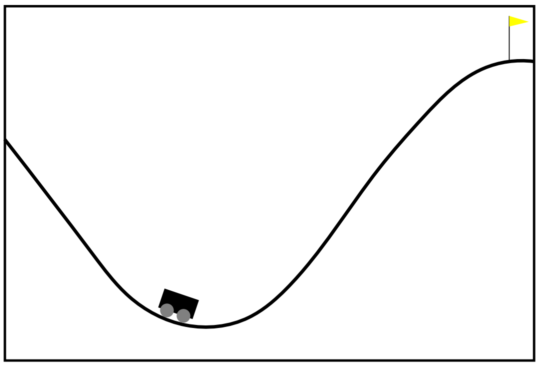
Hyperparameter define higher level concepts of the models used for machine learning algorithms (e.g. number of training examples for each update step, learning speed). In contrast to the model parameter which are optimized based on the data, they need to be predefined and are usually kept fixed during the learning process. Determining the right hyperparameters for a specific learning algorithm can be challenging and time consuming. Nevertheless the performance strongly depends on these parameters so a sound choice is necessary. For the experiments with the less complex environments (Swing-Up Pendulum, Mountain Car) the hyperparameters were chosen based on a simple grid search for both algorithms. With a list of possible values for each hyperparameter a model was trained to each combination and the best one selected. If not explicitly stated differently, the hyperparameters were then kept fixed to these values across all experiments according to Table 4.1.

Table 4.1: Hyperparameter for Pendulum, Mountain Car and Lunar Lander

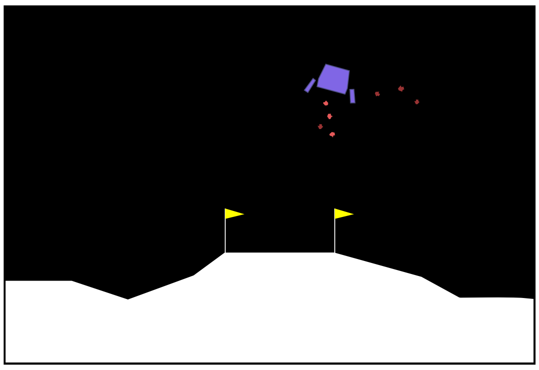
Hyperparameter	**Pendulum**	**Mountain Car**	**Lunar Lander**	**Desciption**
minibatch size	128	128	400	number of training examples over which SGD is computed
replay memory size	100000	100000	100000	maximum number of transitions in the buffer
discount factor	0.95	0.99	0.999	discount factor γ used in the Q-network update
learning rate	0.003	0.001	0.001	learning rate used by ADAM
initial exploration	1	1	1	initial value for ϵ in ϵ-greedy policy
final exploration	0.01	0.01	0.01	final value for ϵ in ϵ-greedy policy
epsilon decay factor	0.0005	0.00002	0.00006	$\epsilon = \epsilon_{fin} + (1 - \epsilon_{fin}) \cdot \exp(-\epsilon_{dec} \cdot steps)$



(a) Swing-Up Pendulum Environment. The state consists of information about the angular position $sin(\theta) \in [-1, 1]$ and $cos(\theta) \in [-1, 1]$ as well as the angular velocity $\dot{\theta} \in [-8, 8]$. The original continuous action space is discretized into the minimum and maximum joint effort $a \in [-2, 2]$.



(b) Mountain Car Environment. The state includes the position $s_0 \in [-1.2, 0.6]$ and the velocity $s_1 \in [-0.07, 0.07]$ of the car. The three discrete actions are either push the car to the left, to the right or do nothing. The end of the left hill acts like a wall where the car can not go any further.



(c) Lunar Lander Environment. The state space is 8 dimensional including both position and velocity (cartesian/angular) of the lander. Besides that there is a boolean value for each leg indicating if it has contact to the ground. The four possible actions are firing one of the three engines or do nothing.

Figure 4.2: Environments used for evaluation of the algorithms

4.2 Results

All results from the Swing-up Pendulum and the Mountain Car environment are averaged over ten experiments to get a good evidence for the performance. During training the agent always followed the $\epsilon-$greedy policy with an exponential decay over time (see tabular 4.1). For the Swing-up Pendulum the evaluation of the policy during training was done by averaging over five executions with a standard greedy policy starting from a random state $\boldsymbol{s}_0$. The evaluation was stopped after 200 time steps returning the accumulated reward as the *Score*. With the pendulum starting at a random state, the optimal policy can achieve a score between 0 (optimal start with $\theta = 0$ and $\dot{\theta} = 0$) and about -200 (worst start with $\theta = \pm\pi$ and $\dot{\theta} = 0$) resulting in an average score of about -100.
Throughout all the results and plots, *steps* refers to the number of time steps passed in the environment, C to the number of steps after the target network is updated and *iterations* denotes how often the target network has been updated.
Across all experiments the learning directly started after enough samples were collected to fill one batch. Of course, extending this warm-up time and filling the whole replay memory beforehand might further improve and stabilize the learning, but as both algorithms are meant to work on-line, they should preferably run without any previous collected data.

4.2.1 Model Complexity

The first result is based on a comparison between DQN and BDQN on the Swing-Up Pendulum environment. Both algorithms were trained for 30 target update iterations ($C = 200$) with different numbers of hidden neurons to analyze how the model complexity affects the performance. Figure 4.3 shows the results of this experiment with two heat-maps indicating the score achieved by each algorithm and a third one the difference between them.
Both algorithms learned a good policy given enough hidden neurons and iterations. The evaluation peaks at a score of about -150 for BDQN and slightly higher for DQN. However, the difference heat-map shows, that BDQN strongly outperformed DQN at a certain range of neurons (8-10) and iterations (9-15). This matches the expectation that BDQN requires less model complexity for a successful learning process.

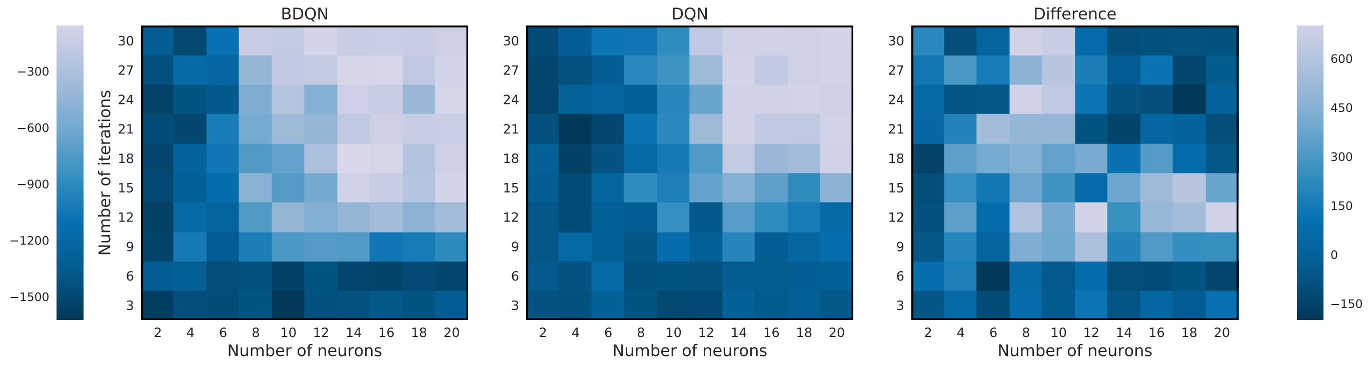


Figure 4.3: Evaluation in the Swing-Up Pendulum environment. The plot shows a comparison of the scores achieved by the greedy policy across 30 iterations w.r.t the model complexity and $C = 200$. The heat-map on the right shows the difference between both algorithms ($\text{Score}_{BDQN} - \text{Score}_{DQN}$)

4.2.2 Sample Efficiency

One important point to analyze is the hypothesis stated in the beginning, saying that BDQN is working more sample efficient (see section 1.1). The heat-maps in Figure 4.4 show a comparison of both algorithms, this time with different values for C across 30 iterations and 10 hidden neurons. As indicated by the difference heat-map, both DQN and BDQN were performing rather similar with values above 200. For faster target update rates DQN seems to have required more iterations, coming up with a good policy very slowly while BDQN still performed reasonably well. However these results do not provide any evidence regarding sample efficiency, as both algorithms required about the same amount of steps for high target update rates until the score converged.
Figure 4.5 shows the score evaluated after 30 iterations including a 95% confidence interval. Except for $C = 50$ both algorithms managed to learn stable policies given enough hidden neurons. With slower target update rates, both algorithms required less hidden neurons to achieve scores up to -100. Throughout all different target update rates, BDQN seems to have improved upon DQN, especially with less complex models. This is coherent with the findings from the previous section but it is still not clear whether BDQN works more efficiently on the samples.

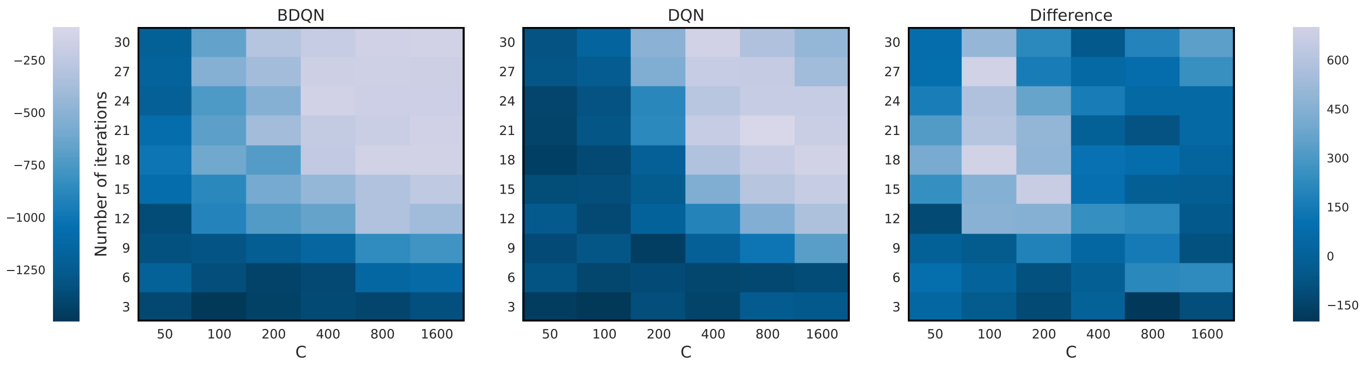


Figure 4.4: Evaluation in the Swing-Up Pendulum environment. The plot shows a comparison of the scores achieved by the greedy policy across 30 iterations w.r.t C and 10 hidden neurons. The heat-map on the right shows the difference between both algorithms ($\text{Score}_{BDQN} - \text{Score}_{DQN}$)

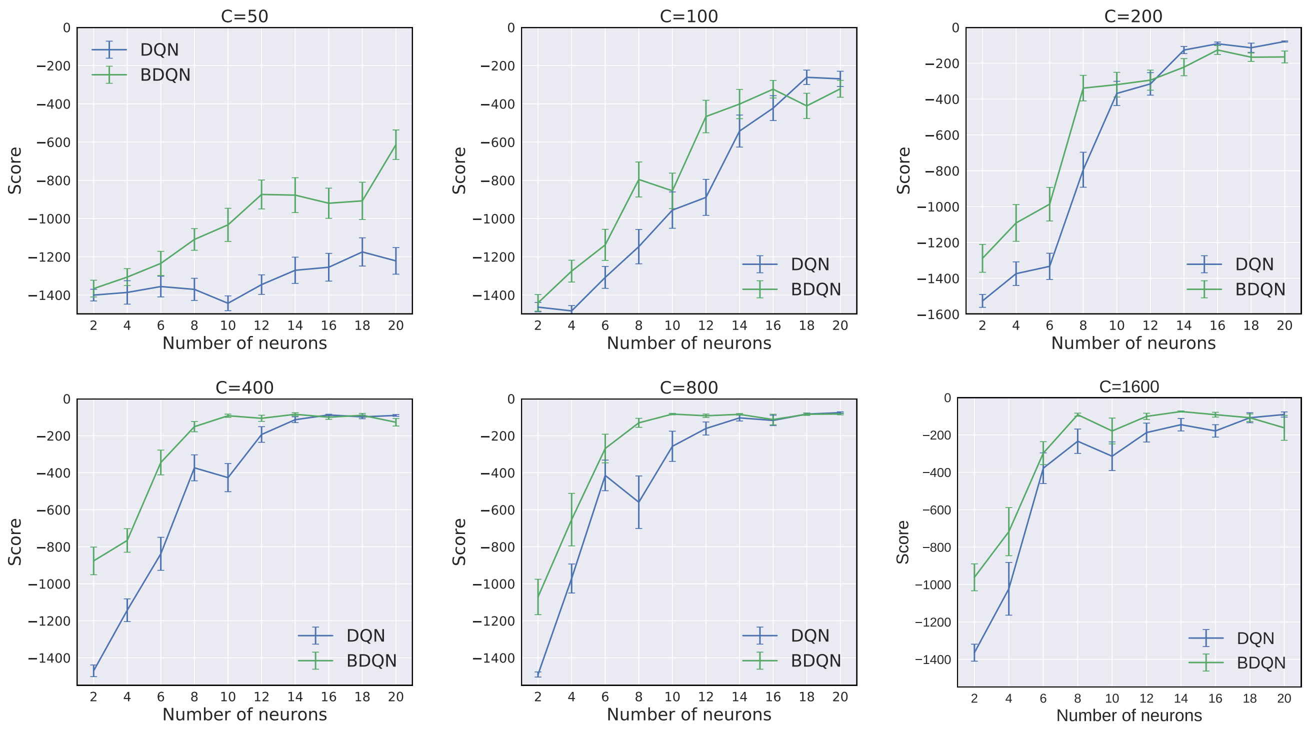


Figure 4.5: Evaluation in the Swing-Up Pendulum environment. The plots show the score with a 95% confidence interval after 30 iterations w.r.t model complexity for different target update rates.

4.2.3 Sparse Rewards

In some environments e.g. the Swing-Up Pendulum the agent can benefit from a smooth and continuous reward signal given for each possible action in the whole state space. This is the optimal situation for a fast learning process as the agent can constantly gain knowledge out of every experience. However in many environments and especially real world applications the reward signal is sparse which means that the agent may experience the same reward in many different states and thus can not gather as much information as with a continuously changing signal. In extreme cases the reward just gives the information if a certain goal is reached or not. To cope with this kind of rewards the agent mostly has to explore a lot of the state space first and then use the little information efficiently to learn a good policy.
The Mountain Car environment is a good example for sparse rewards as the agent receives the same negative reward at every time step except for the goal state. Figure 4.6 shows the evaluation of DQN and BDQN for different model sizes and target update rates. Each experiment lasted until the agent had finished 50 episodes. As the plots show, each agent took around 10.000 steps until finishing the first episode and then needing much less steps for the following episodes.

For a small model size (10 hidden neurons) it seems, that BDQN was able to learn a better policy compared to DQN as it required less steps overall to finish the episodes. With a slower target update rate the performance was even better and both algorithms finished the episodes with less variance. The most consistent results however could be achieved with a higher model complexity of 50 hidden neurons where DQN and BDQN solved the task in about 300-500 steps after finishing the first 20 episodes. During the last episodes BDQN seemed to slightly overfit on the data as the variance increased and more steps per episode were required again.

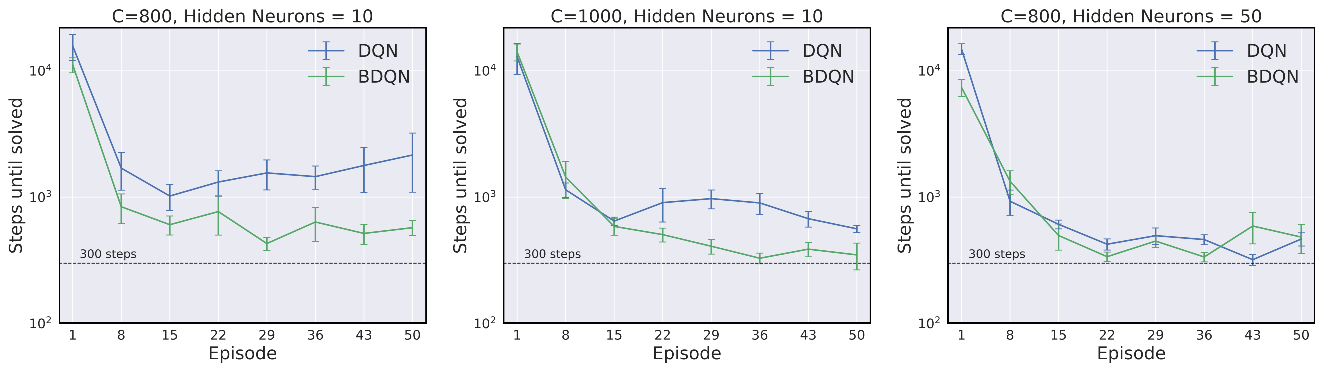


Figure 4.6: Evaluation in the Mountain Car environment. Each agent was trained until 50 episodes were successfully finished without any evaluation policy. The corresponding amount of steps needed for each episode is plotted on a logarithmic scale including a 95% confidence interval for all ten experiments. The dashed line indicates the number of steps approximately required to get from the valley to the goal following a near optimal policy.

4.2.4 Complex Environments

The Lunar Lander environment is in many ways a much more challenging environment. First, the state space is much larger with eight dimensions and furthermore the reward function is far more complex as the previous ones. Dependent on a successful landing or a crashing of the lander, the agent receives additional positive or negative reward points. This leads to discontinuities within the function, which can cause problems to the learner. For the following experiments, both algorithms have been evaluated in this environment. Due to the long learning process, each result is an average over just two experiments. If the lander has neither landed successfully nor crashed, the episode was terminated after 2000 time steps. The policy evaluation was then conducted after each episode for at most 500 time steps. In order to achieve reasonable results in the first place, the target update frequency had to be set much slower to $C = 10.000$ for BDQN. DQN performed convincingly with a much faster update rate of $C = 600$.
Figure 4.7 shows the result for both algorithms across three different model complexities. Both algorithms were able to achieve a score of about -60 reasonably fast. This score corresponds to a local optimum where the lander successfully hovers above the ground without crashing, getting only slight negative reward for firing the engines.
With a higher model complexity, DQN manages to eventually collect about 200 points, which corresponds to a fast and successful landing. Unfortunately, BDQN never managed to achieve positive scores consistently despite different model complexities.

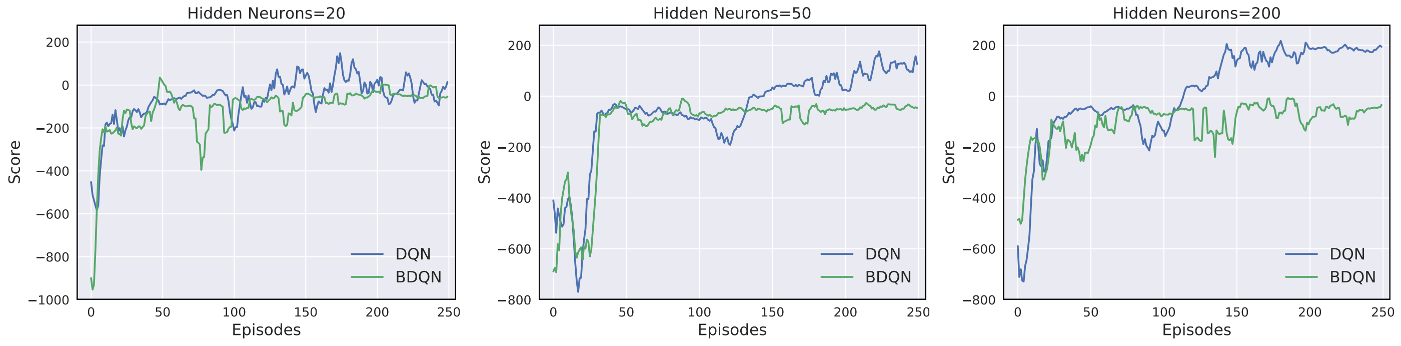


Figure 4.7: Evaluation in the Lunar Lander environment. Each agent was trained for 250 episodes or around 150.000 time steps with $C = 600$ for DQN and $C = 10.000$ for BDQN. The score of the greedy policy is represented as a moving average over five episodes to reduce the noise and emphasize the movement trend.

4.2.5 Double Learning

A common problem with the DQN algorithm and Q-Learning in general is that the agent often overestimates the true Q-Function value. This is caused by the max operator used for setting the targets

$$Q(\boldsymbol{s},\boldsymbol{a}) \leftarrow r + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}',\boldsymbol{a}').$$

Suppose there is a set of actions in a given state whose true Q-Values are all approximately equal. The Q-network has not converged yet and predicts a really noisy estimate of the values. According to the max operator, the agent will always select the action with the highest value and thus with the highest error. This error will get propagated to other action values and therefore yield an overestimation bias. Of course a uniform overestimation of all states is not necessarily a problem and the resulting policy may stay the same. However if this bias is concentrated on a small subset of states, it could have a large impact on the stability of the algorithm.
A solution to this problem is called *Double Q-Learning*, which was proposed by Hado van Hasselt in 2010 [15] for the tabular setting and later generalized for arbitrary function approximation [16]. The idea is to decouple the action selection from its evaluation by learning two value functions independently.

$$Q^A(\boldsymbol{s},\boldsymbol{a}) \leftarrow r + \gamma\, Q^B(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} Q^A(\boldsymbol{s}',\boldsymbol{a}'))$$
$$Q^B(\boldsymbol{s},\boldsymbol{a}) \leftarrow r + \gamma\, Q^A(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} Q^B(\boldsymbol{s}',\boldsymbol{a}'))$$

With each transition either Q^A or Q^B is updated. As each Q-Function is updated with a value from the other Q-Function the resulting values can be considered as unbiased. In the case of DQN and BDQN as described in section 3, the target network can be used for decoupling the value estimates leading to a simple change in the target formula

$$\begin{aligned} Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}) &\leftarrow r + \gamma\, Q(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} Q(\boldsymbol{s}',\boldsymbol{a}'|\boldsymbol{\theta})|\boldsymbol{\theta}') \text{ for DQN,} \\ \tilde{Q}(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}) &\leftarrow r + \gamma\, Q^T(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} \tilde{Q}(\boldsymbol{s}',\boldsymbol{a}'|\boldsymbol{\theta})) \quad \text{for BDQN.} \end{aligned}$$

The following experiments provide an analysis of BDQN including the double learning modification. Figure 4.8 shows an evaluation comparison of BDQN and double BDQN in the Swing-Up Pendulum Environment. Throughout the different target update rates and number of hidden neurons, both algorithms seem to have performed rather similar during 30 iterations.

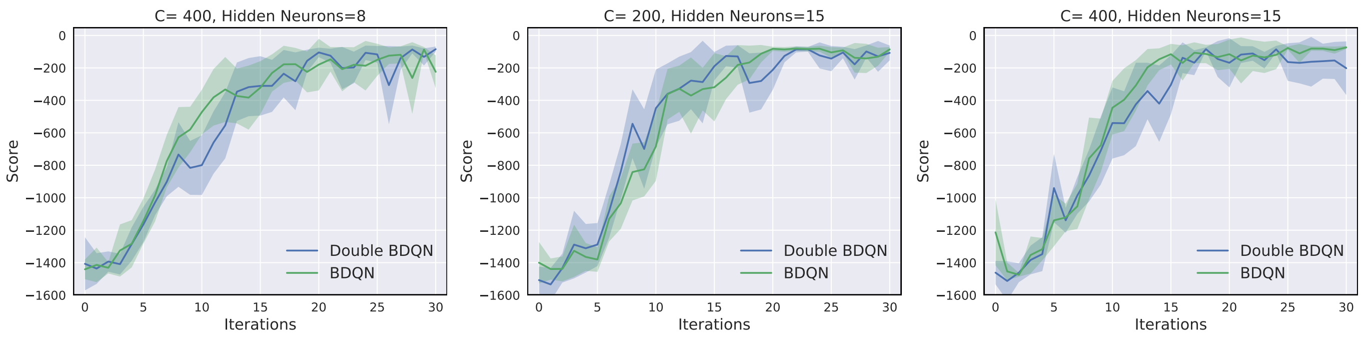


Figure 4.8: Evaluation in the Swing-Up Pendulum environment. The plots show a comparison of BDQN and double BDQN for different target update rates and model complexities through 30 iterations. The results include a 95% confidence interval over the ten experiments.

4.2.6 Model Visualization

A good way to analyze the behavior of a trained agent and further analyze the importance of double learning is to look at its learned value function for the whole state space. With the Swing-Up Pendulum environment this is possible, because the state space only consists of the angle and angular velocity which can be represented in two dimensions. Figure 4.9 shows the value function learned by three different agents for the Swing-Up Pendulum problem. As expected, the highest values always appear along a diagonal bar in the center of the state space where the reward is the highest ($\theta = 0$ and $\dot{\theta} = 0$). If the pendulum is slightly deviated from the upright position, a velocity in the opposite direction leads to high values as the pendulum will most likely reach the upright position in near future.

A close look on the value bar further reveals, that the double learning architecture introduced in section 4.2.5 indeed influenced the value estimation of the agent. Given the reward function for the Swing-Up Pendulum environment (Eq. 4.3), the true value of a state can never be positive. Because of that, both DQN and standard BDQN clearly tended to overestimation, predicting values up to 6 for some states. Only Double BDQN stayed below 0 with all estimated values. In this case however, the overestimation did not lead to worse results as the general shape of the value function was unaffected.

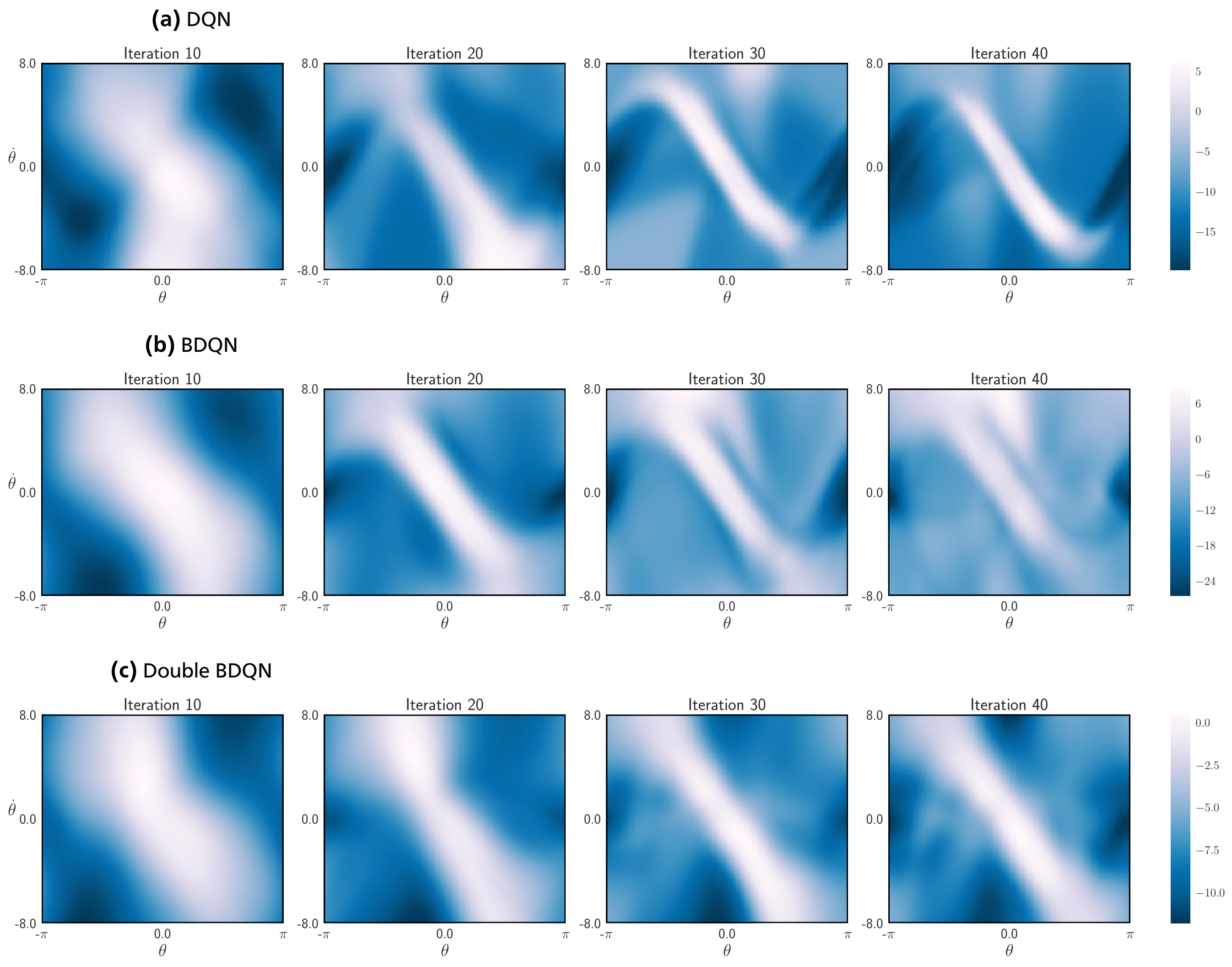


Figure 4.9: Visualization of the Value Function learned at different iterations for a) DQN, b) BDQN and c) Double BDQN with 15 hidden neurons and C=400. The function is evaluated over the whole state space with the pendulums angle θ on the horizontal axis and its angular velocity $\dot{\theta}$ on the vertical axis.

4.2.7 Increasing Sample Efficiency With Prioritized Experience Replay

Another possible improvement for deep reinforcement learning already mentioned in section 1.2 is to modify the way experience is used. With the approach mentioned in section 3.1.2, experience samples are drawn from the replay memory buffer according to a uniform distribution. This way all samples are treated the same, no matter how much information they contain. The idea of Prioritized Experience Replay (PER) [14] is to change the sampling distribution in a way that transitions with a potential higher information gain are preferred.
As it is not possible to directly measure how much the agent can learn from a particular transition in the current learning state, this information has to be approximated. The TD-error of the transition is a feasible choice for this approximation since it indicates how much the transition differs from the expectation and therefore how much can be potentially learned from it. For Double BDQN this error of a sample transition $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}')$ can be expressed as

$$error = \left|\tilde{Q}(\boldsymbol{s}, \boldsymbol{a}) - t(\boldsymbol{s}, \boldsymbol{a})\right| \quad \text{with} \quad t(\boldsymbol{s}, \boldsymbol{a}) = r + \gamma\, Q^T(\boldsymbol{s}', \arg\max_{\boldsymbol{a}'} \tilde{Q}(\boldsymbol{s}', \boldsymbol{a}'|\boldsymbol{\theta})).$$

The error can then be translated into a priority of being chosen for the particular transition with

$$p = (error + \epsilon)^{\alpha}.$$

The small positive constant ϵ ensures that no transition has zero priority, even if its TD-error is zero. The parameter $\alpha \in [0, 1]$ determines the magnitude of the prioritization; $\alpha = 0$ would be equivalent to the uniform case. Finally, the priority can be converted to a probability P_i of being chosen for sample i.

$$P_i = \frac{p_i}{\sum_k p_k}$$

With each update step, the batch is now sampled from this probability distribution and used for the training of the Q-Network. New transitions can be initialized with a high priority to ensure that they are used for training at least once. As the sampling can be computationally expensive, it is important to store the samples and their respective errors efficiently. The approach proposed in the original paper is using an unordered binary sum-tree for the data structure. Each sample is stored in a leaf node with its priority and each parents node value is the sum of its children. Sampling from the distribution now follows the simple process of picking a random number $s \in [0, \sum_k p_k]$ and choosing the sample where the sum exceeds s by propagating down the tree. The errors can also be updated fastly by propagating the difference to the leaves ($\mathcal{O}(log\ n)$) and changing the priority accordingly.

Figure 4.10 shows the evaluation of BDQN using prioritized experience replay with $\epsilon = 0.01$ and $\alpha = 0.3$. This time, the model complexity was kept fixed to 15 hidden neurons, but the batch-size was reduced to 16 and 32 samples per batch. With $C = 200$ and a batch-size of 32, priority sampling did not improve the results, except for the last iterations, where less variance occured compared to uniform sampling. With an even smaller batch-size of 16 and a faster target update rate, it seems that the priority sampling improved the learning process slightly.

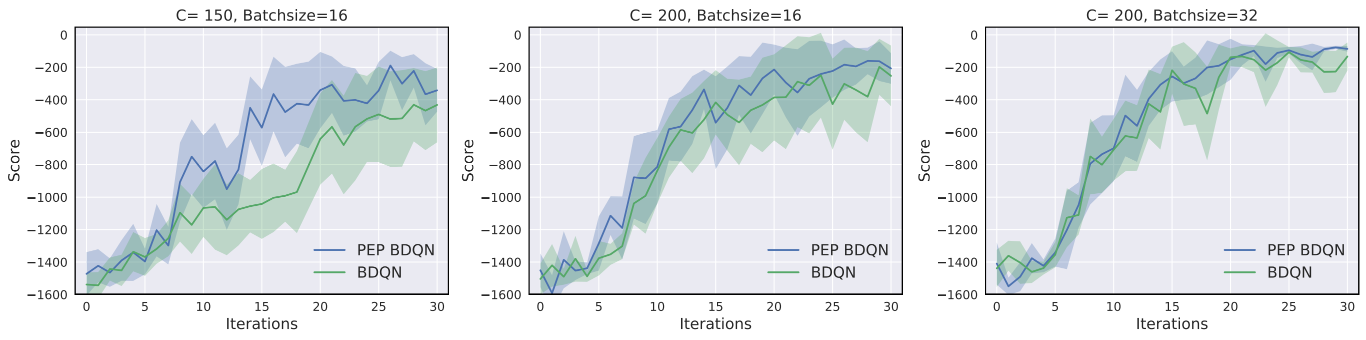


Figure 4.10: Evaluation in the Swing-Up Pendulum environment. The plots show a comparison of the score achieved with different target update rates and batch-sizes including a 95% confidence interval.

4.2.8 Time Performance

The last experiment is about the time performance of the different algorithms and variations. Figure 4.11 shows two results, where DQN, BDQN and each modification were trained for 50 target update iterations with and without GPU support. No evaluation took place during training while the time between each target update was measured for five experiments and then averaged. Apparently, BDQN took much more time than DQN on just the CPU as well as with GPU support. Because the target update for BDQN includes adding a new residual network to the additive model, the time for each update keeps increasing. However, GPU support clearly improved the time performance of both algorithms but in particular of BDQN, where learning could be speeded up almost by a factor of 7. This is not surprising, as the required matrix multiplications within the increasing model can extensively benefit from the parallel computation.
Double BDQN requires one additional evaluation of the Q-Value function each update step and therefore needed even more time between each target update. Using prioritized experience replay (PEP) led to the slowest learning in the beginning, as each update step includes the priority sampling and a propagation of the new priorities through the binary tree. However, the computation time is increasing slower, as the underlying data structure of the prioritized replay memory is more efficient with large amounts of samples.
At around 32 iterations appears a remarkable drop in the GPU computation time throughout all algorithms except for DQN. This could be due to a matrix operation optimization, which starts working at a certain threshold and size of the model.

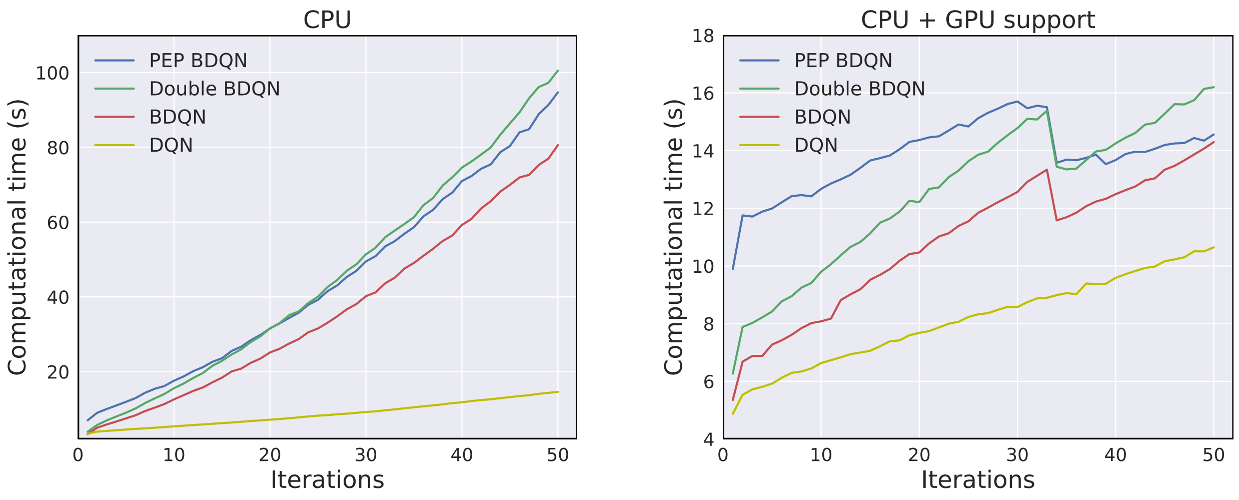


Figure 4.11: Time performance evaluation in the Swing-Up Pendulum environment. Both agents were trained over 50 iterations using a model complexity of 200 hidden neurons and $C = 1000$. The plots show the computation time measured between each target update. The training was done using an Intel Core i5-7400 @ 3.00GHz x 4 and a GeForce GTX 1050.

5 Conclusion And Future Work

This thesis includes an analysis of the deep reinforcement learning algorithm DQN and provides a modified version of the algorithm, namely BDQN, which is using *gradient boosting* to approximate the action-value function. Both algorithms have been successfully implemented and evaluated on three different sample environments including two classic control problems and one high-dimensional environment.
Throughout all experiments with the Swing-Up Pendulum environment, both algorithms performed as expected, deriving near optimal policies within the discretized action-space. Comparing model complexity and sample efficiency was particularly important as these are the two main issues DQN suffers from. The empirical results have shown that BDQN was consistently able to achieve at least the same performance with a simpler model compared to DQN, which allows the conclusion that BDQN requires less model complexity in this specific environment. Regarding sample efficiency, the results were not that clear. With faster target update rates, BDQN could outperform DQN, but did not consistently manage to derive the optimal policy faster. Furthermore, as the evaluation after 30 iterations indicates, BDQN only converged faster, when the model complexity was sufficiently low. Therefore, to allow any reliable statement concerning sample efficiency, further experiments would be required to substantiate the supposition.
Although BDQN performed consistently on the classic control problems and demonstrated robustness when exposed to sparse rewards, it did not manage to learn a policy for solving the more complex Lunar Lander environment. After a few episodes, the agent successfully found a way to hover and avoid crashing, but could not escape this local optimum, even with sufficient exploration. This might be a major problem of using a model which is built sequentially over time.
During the learning process, each residual is just trained for a rather short period and then the parameters are kept fixed in the target network. In the beginning, the agent mainly observes states, where the lander is above the ground or crashes with a high velocity. Therefore most of the residuals are being trained without information about a successful landing and predict high values for hovering and low values for being at the ground, as the small positive rewards for contact between legs and ground are outweighed. At some point the agent may experience a successful landing and the residual is trained to the corresponding samples. However, after it has been added to the target model, the information could be predominated by the preceding residuals and therefore be lost. In a comparable example, DQN would not have this issue, as the whole model is being trained throughout each iteration.
The time performance experiments revealed that BDQN greatly benefits from GPU support. Nevertheless, this is still a big issue, as DQN performs almost twice as fast while occupying less memory space.

Apart from that, two of the propositions for possible improvements, mentioned in section 1.2, have been successfully employed and evaluated with BDQN. Using *Double Learning*, the overestimation tendency of the standard BDQN clearly could be prevented as the inspection of the learned value functions has shown. For the Swing-Up Pendulum environment however, it seemed that the results were unaffected and noticeable changes to the final policies stayed out. This is most likely caused by the environment's simplicity, which still allows the derivation of a near optimal policy out of an evenly overestimated value function.
More convincing results were achieved by sampling transitions with *Prioritized Experience Replay*. Again, with the standard hyperparameter settings from Table 4.1, the performance did not differ from using uniform sampling, but exposed to a really small batch-size and fast update rates, BDQN clearly could benefit from the different sampling strategy.

5.1 Future Work

Despite several positive results of BDQN, there are still many limitations and problems that have to be further investigated. Particularly, all experiments were conducted in a very much simplified setting using deterministic toy environments and rather small models with only one hidden layer. Because of that there is currently no information about how BDQN would scale on far more complex problems like the Atari 2600 games, where input states first have to be preprocessed and features extracted through several hidden layers.
However, before the algorithm could be evaluated on such environments using convolutional ANNs, some refinements are inevitable. Currently the biggest issue is the worse time performance caused by the increasing additive model. A first step towards improving this would be, to keep the target values for the update of the residual network saved. These values include the output of the complete target model. After one update step, this output just changes according to the addition of the output of the new residual model. Therefore, the target values could be stored with the whole transition and updated by adding them to the output of the simpler residual model.

A further important development would be to implement a method which allows dynamically changing the model complexity during the learning process. With the current version of BDQN, the same model is being added with each target update step. During learning however, as the action-value function is approximated more accurately, the bellman residual is getting smaller and its complexity decreases. Therefore, the residual network should be adjusted accordingly to avoid overfitting. Reducing the number of hidden neurons would be suitable in the case of ANN being used as action value function approximator. This approach could address multiple issues at once, as it would not only diminish the chance of overfitting the bellman residual, but would also lead to an increased time performance and less memory requirements, since the new models get smaller at each target update iteration. This approach could be completed by a certain stopping strategy like *Early Stopping* [32], where at some point no more models are being added.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX*, pp. 363–372, Springer, 2006.

[3] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 3, pp. 2619–2624, IEEE, 2004.

[4] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[5] S. Singh, D. Litman, M. Kearns, and M. Walker, "Optimizing dialogue management with reinforcement learning: Experiments with the njfun system," *Journal of Artificial Intelligence Research*, vol. 16, pp. 105–133, 2002.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[9] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[10] S. Tosatto, C. D'eramo, M. Pirotta, and M. Restelli, "Boosted fitted q-iteration," in *Proceedings of the International Conference of Machine Learning (ICML)*, 2017.

[11] D. Abel, A. Agarwal, F. Diaz, A. Krishnamurthy, and R. E. Schapire, "Exploratory gradient boosting for reinforcement learning in complex domains," *CoRR*, vol. abs/1603.04119, 2016.

[12] M. A. Wiering and H. Van Hasselt, "Ensemble algorithms in reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 38, no. 4, pp. 930–936, 2008.

[13] S. Y. Lee, S. Choi, and S.-Y. Chung, "Sample-efficient deep reinforcement learning via episodic backward update," 2018.

[14] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[15] H. van Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

[16] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[17] C. J. Watkins, *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[18] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, 1992.

[19] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, 1958.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[21] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1997.

[22] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *CoRR*, 2017.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, 1986.

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

[25] R. E. Schapire, “The strength of weak learnability,” *Machine learning*, vol. 5, no. 2, pp. 197–227, 1990.

[26] L.-J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[27] G. Rossum, *Python reference manual*. CWI (Centre for Mathematics and Computer Science), 1995.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[29] P. J. Huber, “Robust estimation of a location parameter,” *The annals of mathematical statistics*, pp. 73–101, 1964.

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

[31] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.

[32] G. Raskutti, M. J. Wainwright, and B. Yu, “Early stopping and non-parametric regression: an optimal data-dependent stopping rule.,” *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 335–366, 2014.