Efficient Exploration using Value Bounds in Deep Reinforcement Learning

Effiziente Exploration in Deep Reinforcement Learning mittels beschraenkter Nutzenfunktionen

Master-Thesis von Volker Hartmann aus Stuttgart

März 2019

TECHNISCHE UNIVERSITÄT DARMSTADT

Efficient Exploration using Value Bounds in Deep Reinforcement Learning
Effiziente Exploration in Deep Reinforcement Learning mittels beschraenkter Nutzenfunktionen

Vorgelegte Master-Thesis von Volker Hartmann aus Stuttgart

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr. Joni Pajarinen
3. Gutachten:

Tag der Einreichung:

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Volker Hartmann, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Datum / Date: Unterschrift / Signature:

Abstract

Deep reinforcement learning (DRL) offers many opportunities in the machine learning area due to its ability to solve fundamentally different objectives using the same architecture and algorithm. Deepmind for instance has shown that the same configuration of deep Q-learning (DQN) can be used to achieve human-level performance in a vast number of different Atari games. However, DRL suffers from poor sample efficiency and is dominated by naive exploration strategies like the ϵ-greedy policy. DQN usually requires a vast number of experiences, which the agent samples through interactions with an environment, to learn good policies. This thesis investigates the usage of value bounds for faster reward propagation [21] in combination with bootstrapped DQN for deeper exploration [4]. Enforcing constraints on value functions potentially leads to more accurate estimations in poorly explored state spaces (early training). Several environments included in OpenAI's Gym toolkit and a subset of Atari games are used to evaluate the proposed method w.r.t. its exploratory ability. It is demonstrated that performance can be improved in early training by a combination of deep exploration and fast reward propagation in some environments. However, results also show that the proposed method sometimes has difficulties to converge to an optimal solution. This thesis lays a foundation for future work in the area of more sophisticated exploration strategies in DRL, as the proposed algorithm requires improvement in some areas to consistently achieve good performance.

Zusammenfassung

Deep reinforcement learning (DRL) bietet aufgrund seiner Fähigkeit mittels eines identischen Agenten unterschiedliche Aufgaben zu lösen viele Möglichkeiten im Bereich des maschinellen Lernens. Deepmind zeigte zum Beispiel, dass dieselbe Konfiguration von Deep Q-Learning (DQN) verwendet werden kann, um in einer großen Anzahl verschiedener Atari-Spiele mit Menschen vergleichbare Leistung zu erzielen. DRL leidet jedoch unter einer schlechten Stichprobeneffizienz und wird von naiven Explorationsstrategien, wie z.B. ϵ-greedy, dominiert. DQN benötigt normalerweise eine große Anzahl an Daten, die durch Interaktion mit einer RL Umgebung gesammelt werden müssen, um gute Handlungsstrategien zu lernen. Diese Masterarbeit untersucht die Verwendung von beschränkten Nutzenfunktionen, für eine schnellere Ausbreitung von Belohnungen [21], in Kombination mit bootstrapped DQN für tiefere Exploration [4]. Die beschränkte Optimierung von Wertefunktionen kann insbesondere in wenig erforschten Zustandsräumen (frühes Training) zu genaueren Schätzungen führen. Das vorgestellte Verfahren wird auf verschiedenen Umgebungen des RL toolkits Gym und einigen Atari-Spielen bezüglich seiner explorativen Fähigkeit ausgewertet. Es wird gezeigt, dass die Leistung im frühen Training durch eine Kombination aus tiefer Exploration und schneller Ausbreitung von Belohnungen in mehreren Umgebungen verbessert werden kann. Allerdings wird auch gezeigt, dass das vorgestellte Verfahren gelegentlich Schwierigkeiten hat, zu einer optimalen Handlungsstrategie zu konvergieren. Diese Masterarbeit bildet ein Fundament für zukünftige Arbeiten im Bereich komplexerer Erforschungsstrategien in DRL, da Verbesserungen am vorgestellten Verfahren notwendig sind, um konsistent gute Ergebnisse zu erzielen.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
ALE	arcade learning environment
ANN	artificial neural network
BDQN	bootstrapped DQN
BOT	bootstrapped optimality tightening
CL	convolutional layer
CNN	convolutional neural network
DDQN	double DQN
DL	deep learning
DQN	deep Q-Network
DRL	deep reinforcement learning
EBU	episodic backward updates
FL	fully-connected layer
i.i.d.	independently and identically distributed
KL	Kullback-Leibler divergence
MDP	Markov decision process
MSE	mean squared error
OT	optimality tightening
PM-BOT	penalty mask bootstrapped optimality tightening
POMDP	partially observable Markov decision process

ReLU	rectified linear unit
RL	reinforcement learning
UCB	upper confidence bound

List of Symbols

Notation	Description
a	possible action given state s
ϵ	random exploration factor
γ	discount factor for Markov decision processes
π	acting policy that returns an action given a state
$Q(s, a)$	Q-value for state action pair
r	reward signal from environment
s	state of an agent in reinforcement learning
θ	neural network parameters (online network)
θ^-	neural network parameters (delayed target network)

1 Introduction

Efficient exploration of state spaces is one of the main challenges in reinforcement learning (RL), where state spaces tend to explode with more challenging tasks. Exploration directly determines how fast RL agents learn, as agents learn on data sampled from the environment by interacting with it. The exploration-exploitation trade-off states that an algorithm has to find the right balance between acting greedily, i.e. making a choice that maximizes some goal measure according to the current knowledge, and exploring new states that may lead to suboptimal immediate results, but could lead to better performance in the long run.

Agents are inherently greedy, as they strive to maximize some reward signal returned by the environment, where the reward determines how useful an action executed from the current state is. While acting greedily may lead to immediate best actions, it makes no assumption about the long term effects of these actions. Acting only greedily dampens an agent's ability to learn, as unseen states with potential high rewards are never explored. Thus, exploration has to be induced through random actions or some heuristic that evaluates the potential information growth of visiting states non-greedily. Directing exploration to useful states is difficult. Hence, many algorithms rely on simple exploration strategies that randomly select actions with some probability. The random action probability is usually decreased over time to encourage random exploration in early training, while actions are chosen more greedily once the state space is sufficiently explored and the agent is able to make more accurate predictions.

RL agents learn from trial and error. An agent interacts with an environment and receives reward signals based on its actions. The current state of the agent, executed actions, and environment feedback in form of rewards act as experience for an agent to learn from. Although RL is able to learn vastly different tasks with the same agent, it requires sophisticated algorithms to minimize the training time.

DQN uses ANNs in combination with Q-learning to approximate value functions and automatically extract relevant feature representations from network inputs. Using ANNs in combination with RL enables an agent to deal with high-dimensional inputs and large state spaces. However, the problem of exploration still persists. Rewards are often spare in the sense that they are received only rarely, which makes it difficult to find correlations between executed actions and observed rewards. DQN iteratively improves the parameters of a Q-function via gradient steps, but reward propagation is slow. He et al. propose an OT approach, where lower and upper bounds are enforced by a penalty on the basic DQN loss function [21]. The usage of value bounds leads to faster reward propagation, which significantly improves results in early training, where the state space is sparsely explored.

To tackle the problem of sample efficiency in DQN, Osband et al. propose BDQN [4], which extends the DQN architecture by approximating several Q-functions using different network heads. The parameters of each head are updated using the same data, but each head is initialized randomly and trained against a distinct target network. Osband et al. show that bootstrapped DQN can partially solve some hard exploration Atari games like Montezuma's Revenge, where DQN is unable to find any useful policy. For sparse reward environments, where several tasks have to be solved before any positive rewards are received, random exploration alone is insufficient. Whenever a long sequence of actions are required to receive positive rewards, DQN either does not get into positive reward states at all or is unable to draw a connection between cause (action) and effect (reward).

This thesis combines OT via value bounds and BDQN. The agent should get into positive reward states more frequently through faster reward propagation (OT) and a deeper exploration (BDQN). The proposed method BOT is described in detail in section 4.3. The performance, benefits and drawbacks of BOT are evaluated in chapter 5 by running the modified DQN agent on several smaller environments offered by OpenAI's Gym toolkit (https://gym.openai.com/), which is commonly used to evaluate the performance of RL agents, and some Atari games, the common evaluation platform for DQN agents.

2 Foundations

In this chapter, the most crucial fundamentals of the RL literature are explained. First of all, Markov decision processes (MDPs) are discussed in section 2.1, as RL builds on the same principles. Next, basic terminologies and iterative learning approaches of RL are explained in section 2.2. An introduction to the basic components of ANNs is given in section 2.3, followed by an explanation how ANNs are used in combination with RL giving rise to deep reinforcement learning (DRL) in section 2.4.

2.1 Markov Decision Processes

Decision processes can be utilized to model the interaction of an agent with its environment. The agent is in a state and has a possible set of actions to choose from. Each action leads the agent to a new state. Furthermore, each state transition results in a reward signal as feedback to evaluate the optimality of a chosen action or being in some state in general.

2.1.1 Markov Decision Process

An MDP consists of a set of states and transitions, where the probability to land in the next state depends only on the current state and chosen action, not all previously visited states (cf. figure **??**). This dependency is called the Markov property. An MDP is a tuple (S, A, P, R, γ), where:

- S is a finite set of states s,
- A is a finite set of actions a,
- P is the transition model. Each state transition is described by a probability $P(s'|s, a)$, where s is the current state, a the action chosen in state s, and s' the next state,
- R is a reward model that assigns a reward r after the transition from s to s',
- γ is a discount factor.

The discount factor γ serves two purposes. First, the value of γ describes the preference of immediate over future rewards, where a value of 1 weights all rewards equally and a value close to 0 disregards future rewards. Second, the discount factor ensures that an infinite sequence of observed rewards still results in a finite value. This property is important, when dealing with infinite MDPs.
MDPs are generally used to described the state s an agent is in, while choosing some action a returns a new state s'. Each state transition also returns a reward that can be used to analyze the behavior of an agent. Solving the underlying MDP is crucial to obtaining the optimal behavior. Methods that can be used to solve MDPs are described in 2.2.3.

2.1.2 Partially Observable Markov Decision Process

In contrast to MDPs, where the agent assumes full knowledge over its current state, partially observable Markov decision processes (POMDPs) model an environment, where the agent's state is dependent on imperfect observations. The agent has to estimate its belief-state in the environment by maintaining a probability distribution over the set of possible states. Hence, the MDP notation is extended by a set of observations Ω and a set of conditional observation probabilities O.

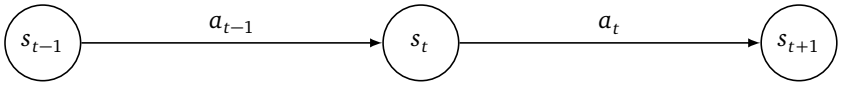


Figure 2.1: Illustration of an MDP. Executing an action a_t in state s_t leads to a transition into a new state s_{t+1}.

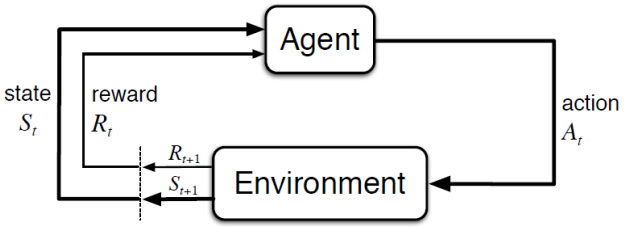


Figure 2.2: Illustration of an agent interacting with its environment (from [1]). Each action A_t executed by the agent in state S_t returns a new state S_{t+1} and reward R_{t+1} by the environment.

After each action $a \in A$, the agent updates its belief based on observation $o \in O$. Due to the Markov property, updating the belief state depends only on the previous belief state b, action a and observation o. By solving the POMDP problem, we can obtain the optimal action in environments with incomplete or uncertain information about the agent's current state. POMDPs are essential to model environments, where an agent has to assume imperfect or incomplete state information due to uncertainties, e.g. environments with probabilistic state transition, where the agent is unable to predict the next state, but has to maintain a probability distribution over possible states.

2.2 Reinforcement Learning

RL is an area of artificial intelligence, where an agent tries to learn from interactions with its environment. Instead of defining exactly how to solve a task, the agent has to learn to solve tasks based on reward signals that indicate how good chosen actions are. For each action taken, the agent obtains a reward and gets to a new state (cf. figure 2.2). Eventually, the goal is to behave in such a way as to maximize the cumulative reward over time, where performance is usually measured as the total reward per episode, i.e. from start to either fail or success. In such a manner, the agent learns, which actions to take depending on its current state. RL is generally unsupervised, i.e. that no labeled data or expert feedback is required to train an agent or evaluate its performance. Hence, the performance is evaluated by considering solely the scored reward. The main challenge is to find an adequate representation of the reward signal and to maximize this reward through the agent's behavior. The state-action reward system is generally modeled as an MDP, while the solution to this MDP problem yields the optimal behavior.

2.2.1 Reward and Return

In RL the reward signal provides feedback for the agent, how good it is too choose a certain action given from the current state. While each action returns a reward r, we usually want to know the cumulative reward, denoted by R, obtained by following a policy π. A policy is formally described as $\pi(a|s)$, i.e. the policy maps each state input to a probability of choosing a possible action from this state [1]. The cumulative reward, or return, in its simplest form is the sum of all rewards encountered when following a policy:

$$R_t = \sum_{t=0}^{\infty} r_{t+1} \tag{2.1}$$

It becomes clear that an infinite sequence of (positive) rewards would also lead to an infinite return. Hence, a discount factor γ is used in RL similarly to γ introduced in the MDP definition. The cumulative discounted reward is then computed as follows:

$$R_t = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{2.2}$$

2.2.2 Value Functions and Policies

The performance of RL agents is evaluated w.r.t. the obtained reward. To evaluate the benefit of being in state s_0 over being in state s_1 values can be assigned to these states to indicate their usefulness to achieving some goal, e.g. maximal rewards. Evaluation can be either performed w.r.t. the state value, i.e. 'how good is it to be in state s_0/s_1', or choosing some action a from state s_0/s_1. While the value function is denoted by $V(s)$, the action-value function is denoted by $Q(s,a)$. Each function maps a state, or state-action combination, to some value that measure performance w.r.t. a desired outcome. Value functions are utilized to evaluate a policy's quality and iteratively improve this policy to ultimately find the (near-)optimal policy $\pi*$.

Value Function

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\pi}[R_t | s_t = s] \tag{2.3}$$

The value function $V^{\pi}(s)$ evaluates how good it is to be in a certain state and following a policy π by considering the expected return (cf. eq. 2.3). R_t represents the expected reward to be observed when executing π starting from a time-dependent state s_t usually until the end of an episode. Note, that the policy π may be stochastic, i.e. there is no deterministic sequence of actions, in which case a probability distribution needs to be kept over all possible state-action combinations and their respective expected returns. Furthermore, the transition function may be stochastic as well, i.e. even when choosing the same action from the same state repeatedly, the agent could still land in different states. If not explicitly stated, policies and environments will be assumed to be deterministic in the following.

Action-Value Function

In contrast to the value function, which determines how good it is to be in state s and executing policy pi, the action value function evaluates how good it is to be in state s, execute action a, and then follow policy π:

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{\pi}[R_t | s_t = s, a_t = a] \tag{2.4}$$

The action-value function is also called Q-function. One algorithm utilizing the Q-function to improve policies will be described in the following section.

2.2.3 Q-learning

Q-learning is one approach to solving MDPs in discrete environments. At its core lies the Q-function, which builds upon the Bellman equation. While, the general goal of RL is to maximize the expected return over a horizon T, MDPs inherit the markov property, which states that the next state $st + 1$ depends solely on the current state s_t. This dependency leads to the value function based on Bellman's equation [5]:

$$Q^{\pi}(s, a) = \mathbb{E}\, R(s, a) + \gamma\, \mathbb{E}\, Q^{\pi}(s', a'), \tag{2.5}$$

where s' and a' stand for the state and an action at time step $t + 1$. Q^{π} then is the expected return considering the immediate reward plus the expected return from executing action a' in s' based on policy π.

$$Q^{*}(s, a) = \mathbb{E}\, R(s, a) + \gamma\, \mathbb{E} \max_{a' \in A} Q^{*}(s', a'). \tag{2.6}$$

To improve the Q-function towards optimality, the policy should choose an action that maximizes the expected return, represented by the Q-function, in the following state, leading to the target for the optimal Q-function in eq. 2.6. The Q-value for state s and action a depends on the immediate reward $R(s, a)$ and the Q-value that is obtained from choosing an optimal action in the following state s'. Rewards are propagated one time step per iteration to approximate the expected return over some horizon, usually from the current state to the end of an episode. Slow reward propagation leads to a slowly converging Q-function, i.e. that meaningful estimates are impossible in early training. How reward propagation can be improved to accelerate learning will be addressed in chapter 4.
The Q-function is updated iteratively by using the current Q-value estimate plus the difference between target and current estimate weighted by some step-size α (see eq. 2.7), where s_t and s_{t+1} represent the current and next state equal to s and s'.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[R(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big] \tag{2.7}$$

Q-learning is an off-policy algorithm, i.e that the value function is optimized during training, while the policy is derived from the Q-function. One such policy for instance is ϵ-greedy, where ϵ is a factor determining with which probability an action is chosen randomly, while the greedy policy (action that maximizes return) is followed with probability $1 - \epsilon$. Traditionally, Q-learning keeps a table of all possible state-action combinations and updates its values iteratively. As this is intractable for larger states spaces, the Q-function can also be approximated using ANNs as will be explained in 2.4. In the following an introduction to ANNs is provided for basic understanding.

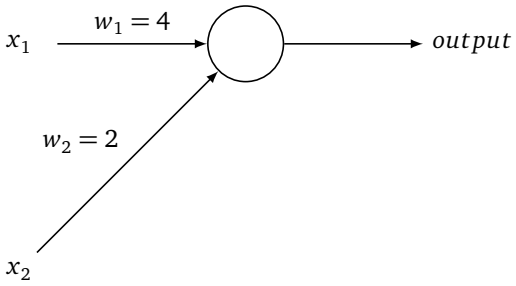


Figure 2.3: Perceptron with 2 inputs and 1 output. The sum of the weights w_i multiplied with the inputs x_i determines whether a perceptron is activated (see equation 2.8).

2.3 Deep Learning

In recent times a lot of attention has been given to ANNs. In general, ANNs serve as non-linear function approximators. In combination with RL, ANNs offer a great opportunity to generalize across different learning tasks due to the possibility of automatic feature extraction. In this section a basic understanding of how ANNs work is given and selected components will be explained.

2.3.1 Neurons

Neural networks are comprised of a set of neurons that are connected in some way. Each neuron has one input or several inputs and one output, while some computation is performed internally. The neuron's output is computed by a nonlinear function of the sum of its inputs. Neurons are generally aggregated into layers (see 2.3.3), while the number of layers describes the depth of a network.
The most basic neuron is a perceptron. Unless explicitly stated, all concepts presented here apply to neurons in general. A perceptron receives some binary input and concludes in a binary output. Each connection between neurons is called an edge and has a weight. In the course of training a network, these weights are optimized to find the most suitable description between input and output of the network. Consider a toy example as illustrated in 2.3. The perceptron computes the weighted sum of its inputs. Hence, if $x_1 = 1$ and $x_2 = 0$ the perceptron's output would be computed as $p_{out} = 4 * 1 + 2 + 0 = 4$. Now, imagine a threshold on the perceptron's output, i.e. that only if its output is greater than this threshold, the final output will be 1, otherwise it's 0 (cf. eq. 2.8). This process of thresholding a neuron's output is called activation function (see 2.3.2), because this function determines when a neuron is activated, i.e. its output is 1. Sometimes the activation of a neuron is also called firing.

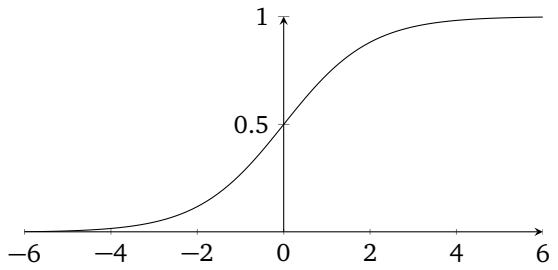
$$output = \begin{cases} 0 & if \sum_j w_j x_j \leq threshold \\ 1 & if \sum_j w_j x_j > threshold \end{cases} \tag{2.8}$$

In general the $threshold$ gets replaced by a $bias = -threshold$. Basically this has the same effect, but now the $bias$ can be incorporated directly into the computation of the $output$ (cf. eq. 2.9). The greater the bias, the greater the probability for the network to output a 1. The network not only trains the weights but also the biases to minimize the error between prediction and correct output. The problem with adjusting these weights and biases is, that a small correction can lead to a big difference in the predicted outcome. This is undesirable, as intuitively, during the learning process, small adjustments in the network should lead to a respectively small change in the output.

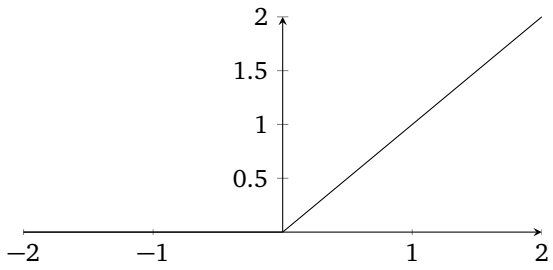
$$output = \begin{cases} 0 & if \sum_j w_j x_j + b \leq 0 \\ 1 & if \sum_j w_j x_j + b > 0 \end{cases} \tag{2.9}$$

2.3.2 Activation Functions

To conquer the problem of small network adjustments resulting in a big difference in output, ANNs utilize activation functions with a smoother transition between the output 0 and 1. Generally, instead of just computing the sum of weighted inputs and comparing the result to some threshold, the weighted sum is passed as input to another function. More complex functions allow real number outputs instead of the threshold binarization.



(a) Illustration of a sigmoid activation function.



(b) Illustration of a rectified linear unit (ReLU) activation function.

Sigmoid Function

One of the most basic activation function is the sigmoid function (see eq. 2.10). The basic computation of a neuron's input (cf. eq. 2.9) is the input to this function, while the output is a non-linear transition on the interval $[0, 1]$.

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-\sum_j w_j x_j - b)} \tag{2.10}$$

Rectified Linear Unit

The ReLU activation function is one of the most frequently used activation functions due to its simplicity and robustness. The output is a simple linear activation that maps all inputs smaller 0 to 0 and preserves all inputs greater or equal to 0 (see eq. 2.11).

$$ReLU(z) = max(z, 0) \tag{2.11}$$

2.3.3 Layers

The aforementioned components are combined to make up layers, which is a more abstract representation of a set of neurons. An ANN architecture is made up of a sequence of layers. Each layer is comprised of a number of neurons (layer size), one or several weighted edges to neurons of the next layer and some activation function between the layers.

Fully Connected Layer

In a fully-connected layer (FL), or dense layer, each neuron of the current layer has a connection to each neuron of the next layer, where one neuron's output is the same for all edges, i.e. all neurons of the following layer receive the same input from one neuron of the previous layer. Furthermore, the input for each neuron of the next layer is a combination of all the outputs of the previous layers' neurons. FLs are among the most basic and widely used layers in deep learning (DL).

Convolutional Layer

When dealing with image inputs it can be helpful to use convolution to reduce the image to a smaller size. The method of convoluting images by sliding a weighted matrix over an image stems from the area of image processing, where convolution is used to create e.g. blurriness or reduce the size of an image by combining nearby pixels to one pixel using a convolutional matrix. While this process leads to information loss, it significantly reduces the state space when considering each pixel-value combination as a possible state. Hence, convolutional layer (CL) enable ANNs to deal with high dimensional input data. The matrix weights in CLs are represented as a convolution kernel that is slided over all pixels in the image to produce a new image of reduced size.
CLs make use of the convolutional process to reduce the size of high dimensional inputs, while automatically extracting features by learning the kernel weights in the network. Utilizing these properties in DRL enables an agent to generalize across vastly different tasks using the same network architecture.

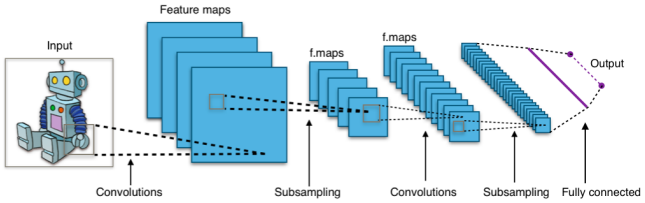
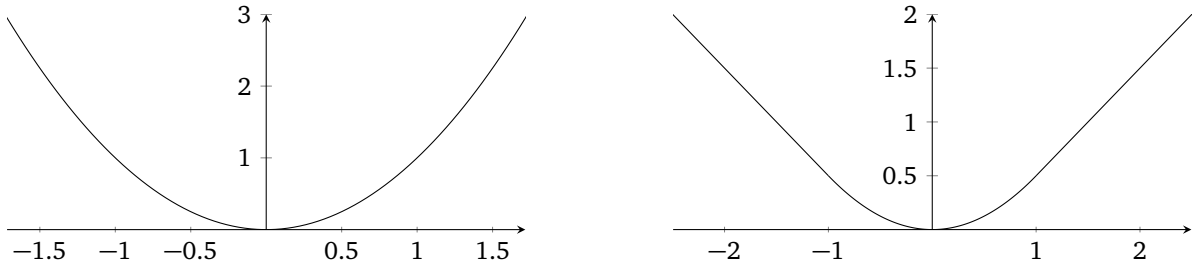


Figure 2.5: Architecture of a typical CNN[1]. Features are extracted from the input image using a series of convolutional and pooling layers (here subsampling). At the end a fully connected layer generates scalar outputs.



(a) Illustration of the squared error loss function, where all inputs are scaled quadratically (see equation 2.12).

(b) Illustration of the Huber loss function with $\delta = 1$, i.e. all values greater δ are scaled linearly (see equation 2.13).

Figure 2.6: Comparison of different loss functions.

Pooling Layer

Pooling layers work very similar to convolutional layers. The main difference is that in contrast to CLs, where a new pixel value is determined by a weighted combination of the original pixels, pooling just takes the minimal or maximum value in the kernel area as output. Pooling layers are used to eliminate redundant information. Even though pooling leads to significant data loss, it helps to prevent pitfalls like overfitting and improves the computational performance through data reduction.
A typical CNN utilizes a series of convolutional layers followed by pooling layers for automatic feature extraction as illustrated in figure 2.5.

2.3.4 Loss Functions

Instead of minimizing the the loss in terms of difference between target value and prediction, ANNs utilize loss functions. A loss function takes in the loss and transforms it in some way to handle different magnitudes of losses.

Mean Squared Error

Taking the mean squared error (MSE) of a loss means that each loss in a mini-batch is first squared and then the average over the whole mini-batch is computed. How the MSE is computed can be seen in 2.12, where the input to this function is a mini-batch of losses (difference between target and prediction) of size N. Sometimes also the sum of losses is used dropping the division by the number of samples. MSE has the property of amplifying large losses, while losses smaller 1 are diminished, i.e. large errors are punished more severely. Squaring the loss also has the effect of eliminating negative losses, as only the distance between target and prediction matters. To avoid exploding losses the squared loss is usually divided by 2.

$$L_{MSE}(loss) = \frac{1}{N}\sum_{n=0}^{N} loss_n^2 \tag{2.12}$$

[1] Image created by Aphex34, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45679374

Huber Loss

Huber loss offers a smooth approach to squared errors, as it squares errors smaller than or equal to some δ (usually 1.0), while treating errors larger δ linearly (see eq. 2.13). Scaling larger errors linearly avoids exploding losses and prevents optimizations from being dominated by outliers in sample data. An illustration of the behavior of Huber loss compared to squared error can be seen in 2.6.

$$L_{huber}(loss) = \begin{cases} \frac{1}{2} loss^2 & for|loss| \leq \delta \\ \delta(|loss| - \frac{1}{2}\delta), & otherwise \end{cases} \tag{2.13}$$

2.3.5 Optimizers

The parameters of an ANN are updated based on the update rules of an optimizer. While the target determines in which direction the parameters are updated, the optimizer determines how the gradient is computed. There is a vast number of optimizers readily available with common machine learning libraries. The optimizers most frequently used in combination with DQN are explained here.

RMSProp

Root Mean Square Propagation (RMSProp) was never officially published, but only introduced in the 6 lecture of an online course by Geoffrey Hinton [6]. Still, RMSProp is widely used by Deepmind for their DQN algorithms (e.g. [2, 7]). RMSProp differs from normal gradient descent in the way that it divides the learning rate for each weight by "a running average of the magnitudes of recent gradients for that weight" [6]. RMSProp is an extension of rprop, which only considers the signs of a sequence of gradients and increases/decreases the step size when encountering a sequence of gradients with equal/different signs [8]. Hinton mentions that rprop can not be effectively combined with mini-batches and, hence, introduces RMSProp, which keeps a moving average of the squared gradients for each weight (cf. eq. 2.14) [6]. Each gradient is then divided by $\sqrt{MeanSquare(w,t)}$.

$$MeanSquare(w,t) = 0.9 \cdot MeanSquare(w,t-1) + 0.1\left(\frac{\partial E}{\partial w^{(t)}}\right)^2 \tag{2.14}$$

Adam

Adam [9] combines the advantages of two optimization methods, namely Adaptive Gradient Algorithm (AdaGrad) [10] and RMSProp.
Like RMSProp, Adam is an adaptive learning rate method, i.e. the learning rate is calculated individually for each parameter instead of using a general learning rate for updates as in basic gradient descent methods. Adam utilizes two moment vectors to adapt the learning rate accordingly, where the first moment of the gradient is the mean as already seen in RMSProp and the second moment the uncentered variance. Kingma et al. describe the ratio between these two moments as *signal-to-noise* ratio, where the signal is represented by the moving average (eq. 2.15a) and the noise by the variance (eq. 2.15b) [9]. The smaller the *signal-to-noise* ratio the closer the adapted learning rate will be to zero.

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \tag{2.15a}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \tag{2.15b}$$

2.3.6 Common Issues with Artificial Neural Networks

Catastrophic Forgetting

The key to tackling catastrophic forgetting is retaining already learned knowledge. Neural networks tend to forget already attained knowledge once a totally new task is introduced, because old knowledge is overwritten to maximize performance on a new task. Deepmind proposes a solution called Elastic Weight Consolidation (EWC), where connections to tasks grow stronger depending on their importance [11]. Once a new task is learned, these connections to old tasks are not eliminated but only degenerated slowly, where tasks with strong connections stay in the memory. In the Atari setting, the agent is able to learn several games in a row, while an agent without continual learning forgets formerly acquired knowledge as soon as a new game is learned.

Co-adaptation

Neurons in a neural network may learn to detect the same features. This is not intended, as the capability of a neural network can only be maximized, if neurons do not learn the same tasks. Furthermore, co-adaptation may lead to overfitting, because neurons are highly dependent on each other, i.e. bad inputs are propagated to other neurons. One approach to tackling co-adaptation of neurons is to use dropout in hidden layers [12]. In this case neurons are randomly omitted during the training process, which should decrease dependent learning between neurons.

2.4 Deep Reinforcement Learning

Instead of calculating the value function in RL exactly using a tabular approach, it is possible to use an ANN to approximate the value function. This is especially useful, as inputs can be directly passed to the network, without worrying about the feature representation. Furthermore, it is computationally infeasible to find exact solutions for value functions in large state spaces. Using tabular approaches to compute optimal value functions only works for smaller state spaces, as it requires too much memory and does not generalize across states. Approximating value functions using ANNs gained popularity in recent years, as it opened new ways for RL to deal with high-dimensional inputs (cf. [2], [13]). However, DRL also suffers from many issues that make learning generally unstable, e.g. high variance or overestimation. Efforts have been made to conquer these challenges and some proposed solutions are described in chapter 3.

2.4.1 Deep Q-Network

It has been shown by Mnih et al. [2] that feature representations can be learned implicitly by ANNs, which eliminates the need to define features by hand. Hence, the same input representation can be used for fundamentally different learning tasks, while the network automatically learns, which features are most important to solving this task.
In general, a neural network is trained to optimize the value function. In the case of Q-learning this means that $Q(s,a,\theta) \approx Q^*(s,a)$, where Q^* is the optimal value function and θ are the parameters of the neural network used to approximate the optimal value function. The loss is defined as the squared difference between the target, which in the case of Q-learning is given by equation 2.6, and the network's current Q-value prediction of taking action a from state s.

$$L(w) = \mathbb{E}[(\underbrace{r + \gamma * \max_{a'} Q(s', a', \theta)}_{\text{target}} - Q(s,a,\theta))^2]. \tag{2.16}$$

While traditionally the squared difference is used as loss function, other loss functions have been proven to stabilize learning. The Huber loss for instance scales the loss between 0 and δ (usually set to 1) quadratically, while scaling the loss greater δ linearly. Scaling greater differences linearly avoids exploding loss functions and, hence, stabilizes learning. The Q-learning gradient to the DQN loss is:

$$\frac{\partial L}{\partial w} L(w) = \mathbb{E}\left[(r + \gamma * \max_{a'} Q(s', a', \theta) - Q(s,a,\theta)) \frac{\partial Q(s,a,\theta)}{\partial \theta}\right].$$

In its simplest form, gradient descent is used to adjust the function approximation in the direction of the error term. Mnih et al. use the RMSProp optimizer for network updates (see sec. 2.3.5).
The DQN algorithm proposed by Mnih et al. has been evaluated on a variety of Atari games (arcade learning environment (ALE)) and proved to generalize training on games offering fundamentally different challenges using the same network and hyperparameters.

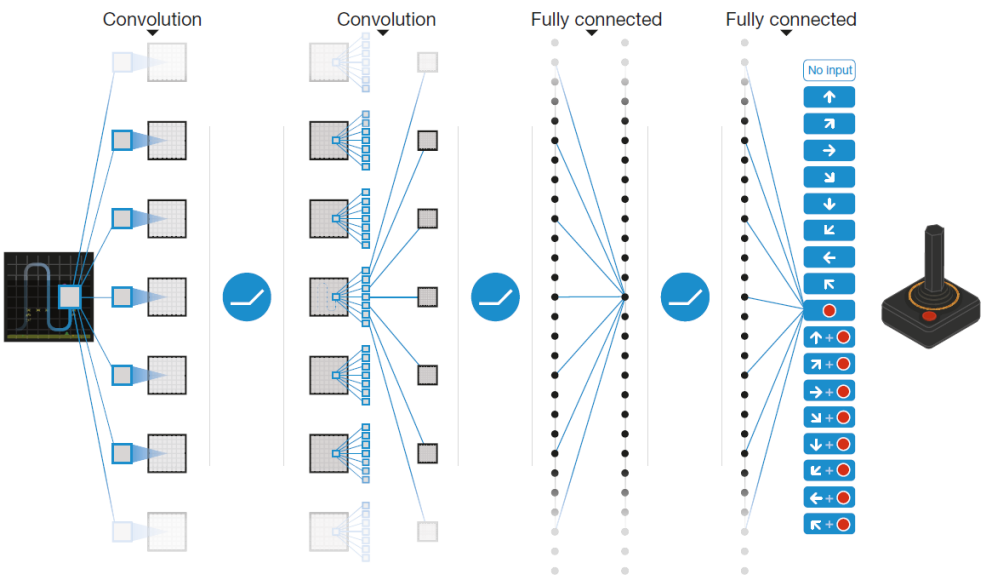


Figure 2.7: ANN architecture used by Mnih et al. in [2] (from [2]).

DQN as proposed by Mnih et al. [2] introduces some modifications to the basic Q-learning algorithm to stabilize learning. The usage of a deep CNN avoids overfitting problems of iteratively updated Q-functions.
Increased stability during training DQN is achieved through two primary modifications:

1. Experience replay randomizes over the data by occasionally sampling experiences from already seen data and updating the Q-function accordingly. Each experience is a tuple of current state, action, next state, and obtained reward. Sampling from replay memory breaks correlation between data, which is usually a problem when learning online from sequences of states. Experience replay makes the training data sort of independently and identically distributed (i.i.d.)

2. The Q-function is adjusted towards a target function $Q_{\theta^-}(s,a)$, where θ^- are periodically updated parameters taken from the learned Q-function. A periodically updated target network leads to decreased oscillation of the action-values during training, because the function used to compute the targets (current objective) stays fixed over a period of timesteps.

The network architecture used by Mnih et al. is illustrated in figure 2.7. The input image coming from the ALE is scaled to 84x84 and the last 4 frames are always stacked to include information about moving objects. Each hidden layer is followed by a ReLU.

2.4.2 Exploration-Exploitation Trade-Off

One of the main challenges in RL is finding a trade-off between exploration and exploitation. While exploration intends to find new states that may lead to high rewards in the long run, exploitation tries to maximize the immediate reward. In practice this means that the agent choses actions that lead to lower rewards, but could lead to potentially interesting states, with the intention to explore the state space. Finding the right balance between exploitation and exploration can be difficult, because the ultimate objective is to maximize the cumulative reward, which makes agents naturally greedily. Especially environments with sparse returns, i.e. returns are only given rarely, it can be difficult for the agent to determine which actions lead to positive returns. First, it is difficult for the agent to make a connection between an action and obtained rewards, if action and effect lie to far apart. Second, an agent may not even get into states with positive rewards, if the state space is not sufficiently explored and rewards are sparse. One naive exploration strategy is ϵ-greedy, where an action is chosen randomly with probability ϵ. ϵ is usually annealed during the training process, with the intent of forcing the agent to explore (randomly) at the beginning and acting greedily after the state space is sufficiently explored. In sparse reward environments this approach may never lead to states with a reward and, hence, makes it impossible for the agent to learn useful trajectories. More sophisticated exploration approaches will be discussed in chapter 3.
The objective of this thesis is to evaluate directed approaches to exploration by limiting the search space to meaningful states. As the state space is too vast to be explored completely, the agent needs some directive to explore promising areas.

Using value bounds to limit the search space may lead to faster exploration, especially in early stages of training, where agents usually act almost only randomly.

3 Related Work

In this chapter work related to DQN, value bound optimization and possible approaches at exploration are presented. First, several promising approaches building on DQN are explained in section 3.1. Possible methods to utilize value bounds in combination with POMDPs and DRL are explored in section 3.2. In section 3.3 different approaches tackling exploration in RL and DRL are introduced. Section 3.4 focuses on works specifically related to OT and BDQN.

3.1 DQN Variants

3.1.1 Double DQN

Hasselt et al. show that DQN, and DRL algorithms in general, have problems with overestimating action values [7]. Overestimation may not incur a negative effect under certain conditions, for instance, if the overestimation is distributed uniformly, i.e. all actions are overestimated by the same amount. However, as shown by Hasselt et al. empirically overestimation can lead to worse policies. To decrease overestimation Double DQN adapts the idea of evaluating and choosing an action on different parameters as proposed in [14]. As DQN already uses two sets of parameters, one for the online network and one for the target network, double DQN simply uses the online network parameter θ_t to choose the optimal policy while evaluating its value using the more consistent target network parameter θ_t^-. The resulting modification to calculate Y for updating θ_{t+1} is:

$$Y_t^{DoubleDQN} \equiv r_{t+1} + \gamma Q(s_{t+1}, \underset{a}{\operatorname{argmax}} Q(s_{t+1}, a; \theta_t), \theta_t^-).$$

Eventually double DQN minimizes overestimation compared to DQN and achieves better results in selecting optimal policies.

3.1.2 Bootstrapped DQN

BDQN is a DQN algorithm that modifies the original network configuration by using several network heads, where each head approximates a different Q-function. The main goal of this approach is to improve exploration by initializing each episode with a randomly selected head and following its policy. The approach is bootstrapped in the sense that it uses samples from the replay memory to train several network heads, while each head is trained on a subsample of the sampled data using a random mask sampled from some distribution. Each head is initialized randomly and trained against its own target network inducing diversity besides the bootstrapped subsamples. For more information on BDQN see chapter 4.

3.1.3 Asynchronous Methods for Deep Reinforcement Learning

Mnih et al. propose an approach that eliminates the need for experience replay [15], which is intended to minimize the correlation between data in time, by using several actor-learners with different exploration strategies. Multiple actor-learners each have their own copy of the same environment to interact with but optimize the same parameters. Updates to the online network parameters are made in parallel by using the experiences of all actor-learners. This approach is applied to several learning algorithms, namely Sarsa, Q-learning, and a form of actor-critic, while the asynchronous advantage actor-critic (A3C) proved to achieve the best results. Furthermore, A3C modifies the objective function by an entropy regularization term for improved exploration:

$$\nabla_{\theta'} \log \pi(a_t|s_t; \theta')(R_t - V(s_t; \theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t; \theta')),$$

where H is the entropy and the hyperparameter β defines the influence of the entropy regularization term.
The different exploration strategies are all ϵ-greedy, while the ϵ is periodically sampled from some distribution. Each actor-learner runs in a different thread on a multi-core CPU, instead of the usual approach to train a Q-network on GPU as in [2]. According to Mnih et al. results inferior to [2] and [7] are achieved on much shorter training time and less computing power.

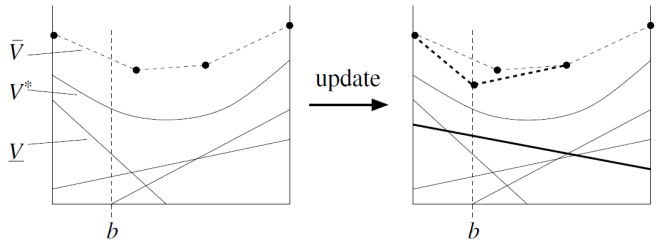


Figure 3.1: Updating the piecewise linear value bounds at belief b (from [3]).

3.1.4 Observe and Look Further: Achieving Consistent Performance on Atari

Several of the main challenges in Reinforcement Learning, especially Q-learning, include fast reward propagation, efficient exploration, and variance in reward signals. To address the issue of variance in reward signals, reward clipping is introduced in [2]. However, this approach has some obvious downfalls as it introduces information loss. If rewards are clipped to $[-1, 1]$, no difference is made between different magnitudes of rewards outside of the clipping bounds. In the Atari game BOWLING for instance, hitting one pin or all of them would result in the same reward signal for the learning algorithm. In [16], reward clipping is substituted by a function that reduces the scale of the action-value function to minimize information loss and still stabilize training.
The planning horizon in Q-learning is mainly defined by its discount factor, where a value closer to 1 results in a larger horizon. Hence, in [16] the discount factor of the Q-function is set to 0.999, while it is 0.99 in e.g. [2, 7]. According to Pohlen et al. the aforementioned modifications lead to significant performance increases. Especially on the Atari game MONTEZUMA'S REVENGE, which requires sophisticated exploration to be solved, rewards could be significantly increased. However, it should be noted that some increase in performance in [16] can be attributed to the use of imitation learning based on expert demonstrations.

3.1.5 Rainbow

Hessel et al. combine some of the most notable extensions to the Q-function and analyze their combined performance on Atari games [17]. By combining prioritized double DQN (DDQN), experience replay, dueling network multi-step learning, distributional RL and noisy nets, performance on Atari games is improved drastically. Furthermore, instead of RMSProp, Adam is used to optimize the network parameters, as it is less sensitive to the choice of learning rate according to Hessel et al [17].

3.2 Optimizing Value Functions using Value Bounds

3.2.1 HSVI - Heuristic Search Value Iteration

Heuristic Search Value Iteration (HSVI) introduced by Smith et al. in [3] maintains lower and upper bounds to approximate the optimal value function in POMDPs. The idea behind this algorithm is to navigate through the search space by selecting the next action based on the current upper value bound. In contrast to an exhaustive search, the effective search space is therefore limited to a subset of the true search space. Hence, HSVI focuses on the beliefs that are most probable to be reached while traversing the POMDP - this is called a point-based method.
Smith et al. show that, although the update of the value function bounds is governed by the highest lower and upper bound of all possible actions, only selecting the next action with the greatest upper bound leads to convergence. The search space is explored as long as the excess is positive. Excess is the difference between the current width of the value bounds interval and the approximation accuracy we want to achieve. The interval function for the current belief b contains the upper $\overline{V}$ and lower bound $\underline{V}$, while its width is defined as $width(\hat{V}(b)) = \overline{V} - \underline{V}$. Figure 3.1 illustrates how piecewise linear value bounds are updated at belief b, while $V*$ represents the optimal value function to be determined. $\underline{V}$ is initialized using the blind policy method, i.e. that all value functions for always selecting the same action are computed [18]. The lower bound is given as the highest value out of all blind policy value functions. $\overline{V}$ is initialized using the solution of the underlying MDP problem. The optimal value function of this MDP is equal to the upper bound [19].
The tolerance of approximating the optimal solution is determined by parameter $\epsilon\gamma^{-t}$, where ϵ is the convergence tolerance of the algorithm and γ the discount factor over time. Therefore, the requirements on uncertainty are looser in

deeper nodes, as the former term becomes larger with greater t. Although HSVI shows convergence on larger POMDPs, the computation of the upper bounds, which is needed to choose actions heuristically, creates major overhead.

3.2.2 FSVI - Forward Search Value Iteration for POMDPS

Another point-based method for POMDP solving is FSVI [20]. Eliminating the need for an upper bound to approximate the optimal value function, this algorithm converges significantly faster than HSVI. By traversing the underlying MDP, which can be solved easier than the POMDP itself, and the belief space together, this algorithm navigates towards rewards or goal points. When such a goal point is reached, a backup in reversed order is performed to update only the value of states in the given trajectory. A goal point can also be defined as a certain amount of reward or number of actions executed. Note, that the current MDP state s of the agent is only available during simulation while it is not during policy execution. The authors point out the limited exploration of the algorithm which is required when executing long sequences of actions to obtain rewards. ϵ-greedy exploration is proposed to solve this problem, however, this strategy may not lead to meaningful trajectories before the algorithm converges.

3.2.3 Optimality Tightening

To address the problem of time complexity in DQN, He et al. use value bounds to achieve faster convergence and, thus, faster learning [21]. To enforce upper and lower bounds a penalty method is used that extends the basic DQN loss function (eq. 2.16) by an additional penalty term. This penalty term includes the priorly computed value bounds and a penalty coefficient λ that scales the influence of the value bounds on the loss function.
To compute the value bounds preceding and succeeding states in relation to the current state are utilized. These state sequences are readily provided by the replay memory. Through empirical evaluation, it is shown that this approach significantly outperforms the original DQN and DDQN on most games in the ALE while training for a fraction of the frames DQN was trained for. A more detailed explanation on OT is provided in section 4.1.

3.3 Intrinsic Motivation and Count-based Exploration

Count-based exploration approaches use the state-visitation count information to direct exploration in potentially unexplored states. This approach follows the optimistic idea that less visited states are potentially informative, i.e. the fewer times a state has been visited during exploration, the more there is still to be learned from this state. States that have been visited rarely get assigned a higher priority to be visited in the future by adding a bonus to their actual value. There are several issues that prevent a simple, effective implementation in DRL, of which some have been tackled by the following papers.

3.3.1 #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning

Tang et al. present an exploration approach that combines classical state visitation counts with state hashing [22]. To deal with continuous state spaces and achieve generalization over similar states, Tang et al. use a hash function, namely SimHash, that projects the input state onto a vector of binary values. Generalization is especially essential because of sparse reward feedback and a typical one-time encounter of states in RL.
Tang et al. learn the hash function using an autoencoder, while the state is binarized by rounding the sigmoid function output to the nearest binary value. Therefore, state visitation counts are not counted for each state, but for each possible binary state representation of dimension k, where k defines the granularity, or degree of generalization, of the hash function. This generalization resolves the problem for visitation counts of continuous state spaces. The total reward is then calculated as follows:

$$r^{+}(s) = \frac{\beta}{\sqrt{n(\phi(s))}}.$$

β represents the bonus coefficient, while $n(\dot{})$ is the state visitation count table which is initialized with zeros. Each time a state is visited, the binary representation in the table, resulting from the hash function $\phi(s)$, is increased by one. Therefore, the bonus reward gets smaller for frequently visited similar states, encouraging the agent to visit less explored states.

3.3.2 VIME: Variational Information Maximizing Exploration

In contrast to random exploration heuristics like ϵ-greedy, VIME directs exploration in the search space by exploring states that result in maximum information gain, which leads to better efficiency [23]. Maximizing information gain is here defined as taking actions that result in large updates to the dynamic model distribution represented by a Bayesian neural network.
To quantify surprise in the model a weighted intrinsic reward for exploring a state is added to the regular reward signal provided by the environment. The hyperparameter representing the weight determines how curious the algorithm is. Intrinsic reward is based on the variation between the posterior distribution after executing action a_t ending in state s_{t+1} and before (measured by Kullback-Leibler divergence), while the posterior considers the history of taken actions and visited states ξ_t up to the current state s_t. As calculating the posterior $p(\theta|\xi)$ is intractable an approximation is used. However, this approach is unsuitable for model-free learning, as it relies on learning a dynamic model and comparing its distribution over time.

3.3.3 Unifying Count-Based Exploration and Intrinsic Motivation

Following a similar approach as Houthooft et al. [23], the algorithm proposed by Bellemare et al. [24] uses intrinsic motivation to direct exploration efficiently. A measure coined *prediction gain* relates pseudo count and information gain. First, a probability distribution $p_n(s)$ before and $p'_n(s)$ after observing an occurrence of state s over the state space Ξ is defined. The distributions are based on the pseudo-count function $\hat{N}_n(s)$, the number of occurrences of state s in a given sequence, and a pseudo-count total $\hat{n}$, the total number of states in a given sequence:

$$p_n(x) = \frac{\hat{N}_n(s)}{\hat{n}} \qquad p'_n(x) = \frac{\hat{N}_n(s)+1}{\hat{n}+1}.$$

Prediction gain is then calculated by taking the difference of the logarithmic distributions:

$$PG_n(x) := \log p'_n(s) - \log p_n(s).$$

Whenever p is learning-positive, prediction gain is nonnegative.
By applying prediction gain as exploration bonus, Bellemare et al. achieve significant performance enhancements in combination with DQN [2] compared to the original DQN algorithm. Especially in the Atari game Montezuma's Revenge, exploration could be heavily improved.

3.4 OT and BDQN Related Work

3.4.1 Episodic Backward Updates

In a similar fashion as in [21], Lee et al. utilize consecutive sequences of states to update the value function [25]. In this setting, a whole episode is sampled from the replay memory to update the network. Lee et al. mention the problem of correlation between successive states, however, they later show empirically that this issue is not predominant. The primary contributions are a temporary target table storing Q-values for the currently sampled sequence and a β factor that weighs target values against target Q-values from this table. The algorithm starts at the end of a sequence and uses backpropagation to update the target values sequentially. The vector of target values for this sequence is then used to update the network.
The main advantage of episodic backward updates (EBU) is a fast reward propagation, which is especially important in environments with sparse reward signals. An evaluation on 49 games of the ALE shows promising results. Lee et al. show that when training for only 10M frames, EBU significantly outperforms basic DQN, some extensions of DQN and even OT.
It should be mentioned, that the OT reference values used by Lee et al. do not coincide with the originally reported results in [21]. This discrepancy may be attributed to Lee et al. performing their own experiments, using the code published by He at al. (https://github.com/ShibiHe/Q-Optimality-Tightening), to obtain reference values for OT.

3.4.2 UCB Exploration via Q-Ensembles

Chen et al. present two algorithms, both of which build upon the idea of BDQN [26]. The first algorithm coined *Ensemble Voting* basically just shows the benefits of BDQN by implementing one network with several heads as output. However,

instead of choosing one head at random per episode, the best action is chosen by majority vote, i.e. the greedy action suggested by most heads will be taken as the best action. Empirical results show that this algorithm already performs better on the ALE than the original BDQN.
The second algorithm combines BDQN with an upper confidence bound (UCB) calculation. Given the output of K heads of the neural network, the UCB is calculated by adding the empirical standard deviation $\tilde{\sigma}(s_t, a)$ of $\{Q_k(s_t, a)\}_{k=1}^{K}$ to the empirical mean $\tilde{\mu}(s_t, a)$ of $\{Q_k(s_t, a)\}_{k=1}^{K}$. The action that maximizes the following equation is considered the best action:

$$a_t \in \underset{a}{\operatorname{argmax}}\, \tilde{\mu}(s_t, a) + \lambda \cdot \tilde{\sigma}(s_t, a),$$

where λ is a hyperparameter determining the weight of the standard deviation. Similar to *Ensemble Voting*, all heads are used to compute the best action. One of the main ideas of BDQN is to increase exploration by selecting heads at random and use the selected strategy for the whole episode. Although this approach works well in early learning, its late performance does not significantly improve over vanilla DQN. *UCB Exploration* shows better late results as well as a steeper learning curve.

4 Bootstrapped Optimality Tightened DQN

In the first part of this chapter, a thorough explanation of OT and BDQN is provided as this work mainly builds upon the idea of using value bounds in the context of bootstrapped DQN. In section 4.3 the main contribution of this work is described in detail.

The base algorithm for all modifications is DQN, which is a Q-learning algorithm based on neural network approximations. OT uses information about state sequences to calculate value bounds which are then enforced by modifying the basic DQN loss function by a penalty term. The bounds can be calculated exactly by using immediate reward information from the replay memory, i.e. information about past interactions with the environment. BDQN on the other hand utilizes the bootstrap principle to maximize the sample efficiency from replay memory by training a network with several outputs, where each head is trained on the same data but initialized randomly and trained against a distinct target network. Both approaches have shown to improve the performance of a DQN agent significantly, especially in spare reward environments which require sophisticated exploration to find useful states.

OT is a way to propagate rewards faster, which is helpful when using DQN, where the accuracy of approximations relies on reward propagation. Reward propagation in DQN is inherently slow in DQN due to iterative improvements of the Q-function. Finding useful states in sparse reward environments is still difficult as exploration relies solely on ϵ-greedy. Especially if OT directs exploration into unfruitful areas of the state space, it is difficult to find useful policies. Using BDQN several Q-functions can be approximated, which increases the probability of DQN actually finding fruitful areas in the state space. BDQN in combination with OT could therefore resolve the issue of OT running into dead end areas of the state space. On the other hand, all network heads may learn similar policies, if random initialization does not introduce enough diversity into the learning process. In this case, the network would simply overfit a suboptimal solution, if most network heads learn the same suboptimal policy. In another scenario, one network head learning a good policy may not dominate over several network heads learning a worse policy, as at the start of each episode the policy of one head is selected at random, i.e. the probability of choosing a 'good' head would still be low. Currently there is no evaluation of a network heads learned policy in DQN, as value functions are optimized directly without evaluating learned policies.

Still, using OT in combination with BDQN should lead to more accurate predictions, i.e. faster learning and less oscillation, compared to solely using BDQN. Eventually, more consistent pedictions should also lead to smaller confidence intervals. However, BDQN may introduce more oscillation into the learning process than just using OT, due to different policies being selected potentially each episode. The degree of difference between the policies hereby determines the oscillation. The expectation of a combined approach is that higher rewards should be obtained faster, but policies may still converge to a suboptimal solution in some cases, where a majority of the heads learn suboptimal policies.

4.1 Optimality Tightening

The main contribution of He et al. is a DQN-agent that explores more efficiently by using value bounds to optimize the value function. These value bounds are calculated based on a sequence of surrounding observations given a current state. The bounds are enforced by a penalty function that extends the basic DQN loss function 2.16.

Value bounds are natural restrictions to a function, i.e. that values taken on by this function should not exceed the given bounds. Lower and upper bounds to a function give restrictions in both directions. The main idea is to limit the actual search space, by categorically excluding unreasonable values, and increase the estimation precision. One way to calculate these bounds in combination with DQN is proposed in [21].

Value bounds in OT are calculated by making use of the Markovian principle, which says that the next state solely depends on the current state, not all preceding states. This principle is also reflected in the Q-function, where updates to the Q-function are based on the Bellman equation, an iterative approach that optimizes w.r.t. current and next state information. Using this iterative update, reward information is propagated through sequences of states. The problem of Q-learning is that rewards are propagated slowly (one timestep per iteration). To tackle this issue, He et al. propose fast reward propagation by tightening constraints on the value function.

$$Q^*(s_j, a_j) = r_j + \gamma \max_a Q^*(s_{j+1}, a) = r_j + \gamma \max_a \Big[r_{j+1} + \gamma \max_{a'} \big[r_{j+2} + \gamma \max_{\tilde{a}} Q^*(s_{j+3}, \tilde{a}) \big] \Big] \tag{4.1}$$

According to He et al., due to the update rule of the Q-function, equation 4.1 holds for the optimal Q-function. This sequence is arbitrarily extendable until the end of an episode has been reached. Using reward information over longer sequences leads to faster reward propagation for each iterative function update.

$$Q^*(s_j, a_j) = r_j + \gamma \max_a Q^*(s_{j+1}, a) \geq \cdots \geq \sum_{i=0}^{k} \gamma^i r_{j+i} + \gamma^{k+1} Q^*(s_{j+k+1}, a) = L^*_{j,k}. \tag{4.2}$$

According to He et al., the lower bound for the current state s_j can be calculated by considering reward information over a sequence of transitions of length k_{ot} as in equation 4.2. In this equation, j is the position of the current sample in the replay memory, which is required to determine state sequences, and k_{ot} is the currently considered range limited by the maximally considered range K_{ot}. The tightest lower bound is then $L_j^{max} = \max_{k_{ot} \in \{1,\dots,K_{ot}\}} L_{j,k_{ot}}$, which is the maximal value of $L_{j,k_{ot}}$ considering all possible transition lengths $1 \leq k_{ot} \leq K_{ot}$ starting from sample position j.

$$U^*_{j,k_{ot}} = \gamma^{-k_{ot}-1} Q^*(s_{j-k_{ot}-1}, a_{j-k-1}) - \sum_{i=0}^{k_{ot}} \gamma^{i-k-1} r_{j-k_{ot}-1+i} \geq Q^*(s_j, a_j) \tag{4.3}$$

In a similar way, the upper bound is calculated using the backward sequence (see eq. 4.3). In contrast to the lower bound, which calculates Q-values based on optimal action selection using the current network parameters, the Q-values for the upper bound are determined by the action sampled from memory. As the optimal Q-function $Q*$ is unknown, He at al. suggest to use Q-function given by the target network, which uses the delayed online network parameters θ^-, to compute the upper and lower bounds. Hence, the same network parameter are used to compute the value bounds for OT and the target for the DQN loss.

$$\min_\theta \sum_{(s_j, a_j, r_j, s_{j+1}) \in \mathcal{B}} \Big[(Q(s_j, a_j; \theta_t) - y_t)^2 + \lambda (L_j^{max} - Q(s_j, a_j; \theta_t))_+^2 + \lambda (Q(s_j, a_j; \theta_t) - U_j^{min})_+^2 \Big], \tag{4.4}$$

He at al. suggest to solve the constraint optimization problem, given by the value bounds, by extending the DQN loss function with a penalty term as in equation 4.4. The first part of the equation is the basic DQN loss function, while the latter two terms determine the penalties inflicted by breaking the lower and upper bounds. In case no bound is violated, the penalty terms will be zero due to a rectifier function (+) that zeros all values lying within the region determined by the lower and upper bound. The λ penalty coefficient determines how strong bound violations are penalized. He at al. set this parameter to a constant value, but they also propose a scenario, where λ could annealed over time.
Agent interactions with the environment are stored as experiences in a replay memory $\mathcal{D}$. To update the parameters θ of the ANN, minibatches of experiences $\mathcal{B}$ are sampled from replay memory, which is called experience replay (cf. algorithm 1). The main difference between basic DQN and OT DQN is, that instead of calculating the gradients based solely on the difference between target and prediction (eq. 2.16), the loss function is extended by a bound penalty (eq. 4.4).

Output: Parameters θ of a Q-function
Initialize θ randomly, set $\theta^- = \theta$
for *each episode* **do**
  Initialize s_1
  for *step* $t = 1, \dots$ *until end of episode* **do**
    Choose action a_t according to ϵ-greedy strategy
    Observe reward r_t and next state s_{t+1}
    Store the tuple $(s_t; a_t; r_t; \cdot; s_{t+1})$ in replay memory $\mathcal{D}$
    Sample a minibatch of tuples $\mathcal{B} = \{(s_j; a_j; r_j; R_j; s_{j+1})\}$ from replay memory $\mathcal{D}$
    Update θ with one gradient step of cost function given in Eq. 4.4
    Reset $\theta^- = \theta$ every C steps

Algorithm 1: Optimality Tightening

He at al. also incorporate the discounted cumulative return R of an episode into the lower bound calculation. However, it is not stated how R is used in this calculation and, thus, its usage is disregarded in this work.
Compared to DQN some additional hyperparameters have to be defined for OT DQN. He at al. suggest hyperparameter values that were coarsely tuned on a subset of Atari games. In experiments performed by He et al. the penalty coefficient is set to $\lambda = 4$, while the maximally considered transition length for calculating the bounds is set to $K_{ot} = 4$. The same constant penalty coefficient is used to weight the lower and upper bound penalties. The possibility of annealing the penalty coefficient over time is mentioned by He et al. but not evaluated.

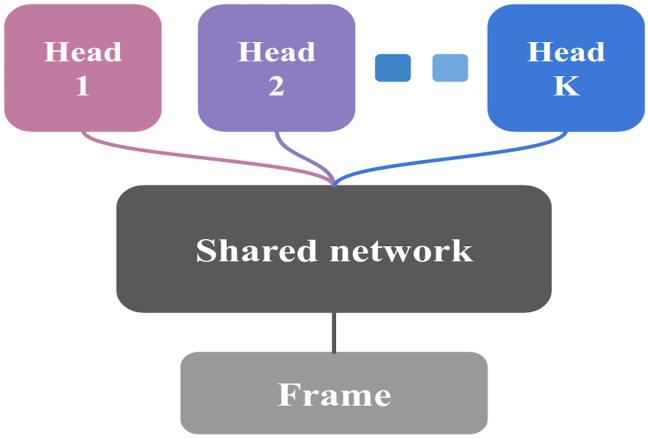


Figure 4.1: Abstract network architecture for bootstrapped DQN using one main network and several heads that each output a different Q-function (from [4]).

4.2 Bootstrapped DQN

Osband et al. suggest a DQN architecture that incorporates several heads to approximate different Q-functions. According to the Osband et al. using different Q-functions should deliver a more sophisticated approach at exploration than simply relying on ϵ-greedy policies, while maximizing the sample efficiency through bootstrapping.

Bootstrapping tries to maximize the efficiency of a dataset by generating (bootstrapping) new data from its samples. Osband et al. apply this principle to DQN by first generating data through interactions with an environment and then using subsets of this data to train different network outputs. The agent uses one shared framework, i.e. one environment to interact with and one dataset, in the form of experiences, that is saved in a replay memory. Furthermore, the agent uses a shared network, as illustrated in figure 4.1), which is split into different network heads. Each head approximates a different Q-function depending on the data it is trained on.

While the shared network is trained on all of the data sampled from replay memory, each head is trained on a subset of the sampled data. To determine which head is trained on which part of the data a Bernoulli mask $w_1, ..., w_K \; Ber(p)$ is sampled for each generated experience. Here, K stands for the number of heads and p is the parametrization of the Bernoulli distribution, where $p = 0.5$ equals double-or-nothing bootstrap and $p = 1$ is ensemble with no bootstrapping (cf. [4]). The number of iterations required by the algorithm increases with decreasing p, as more iterations are needed to train the network if the subset differs for each head. Osband et al. noticed during experiments that the choice of p does not significantly affect the performance of the agent. Therefore, Osband et al. choose $p = 1$ for their experiments to improve computational efficiency. Setting $p = 1$ basically reduces BDQN to an ensemble method, where all heads are trained on all the data sampled from replay memory. Osband et al. mention that training all heads on all samples also increases the sample efficiency, because no samples are disregarded in network updates.

Interestingly, even without bootstrapping the resulting Q-functions differ enough to induce diversity in exploration. Osband et al. give two reasons to explain this phenomenon.

1. All heads are randomly initialized introducing some randomness among the heads from the start.

2. Each head is updated towards its own randomly initialized target network.

Sparse reward environments are difficult to solve in DRL, because even when receiving rewards, it is difficult to find the causing action(s). To make exploration deeper Osband et al. propose to use the different network heads by selecting one head at random per episode and following its policy for the whole episode, i.e. until the agent reaches a terminal state or exceeds some step limit (see algorithm 2).

4.3 Bootstrapped Optimality Tightening

This section explains BOT, the main contribution of this work, in detail. The proposed method combines value bound tightening from OT with a bootstrap network as in BDQN to potentially unite the benefits of both approaches.

Input: Value function networks Q with K_b outputs $\{Q_{k_b}\}_{k_b=1}^{K_b}$. Masking distribution M.
Let $\mathcal{D}$ be a replay memory storing experience for training.
for *each episode* **do**
  Obtain initial state from environment s_0
  Pick a value function to act using $k_b \sim \text{Uniform}\{1, ..., K_b\}$
  for *step* $t = 1, ...$ *until end of episode* **do**
    Pick an action according to $a_t \in \text{argmax}_a Q_{k_b(s_t,a)}$
    Receive state s_{t+1} and reward r_t from environment, having taken action a_t
    Sample bootstrap mask $m_t \sim M$
    Add $(s_t, a_t, r_{t+1}, s_{t+1}, m_t)$ to replay memory $\mathcal{D}$

Algorithm 2: Bootstrapped DQN

4.3.1 Theory

BDQN tackles the problem of deep exploration and employs a more sophisticated exploration strategy than relying simply on ϵ-greedy for exploration. OT on the other hand uses value bound penalties to encourage fast reward propagation. By combining OT and BDQN a diverse, deep and fast exploration should be achieved.
In general, OT and BDQN are readily combinable. As in BDQN, a network using several heads to approximate different Q-function is implemented. Each of these heads is trained against its own target network. Instead of using the basic Q-function as in BDQN, the extended Q-function 4.4 proposed by He et al. will be used to update the network parameters. One immediate downside to this approach is a significant increase in computational requirements, as value bounds have to be calculated explicitly for each network head.

$$\min_{\theta_{k_b}} \sum_{(s_j,a_j,r_j,s_{j+1})\in\mathcal{B}} \Big[(Q_{k_b}(s_j,a_j;\theta_{t,k_b}) - y_{j,k_b})^2 + \lambda (L_j^{max} - Q_{k_b}(s_j,a_j;\theta_{t,k_b}))_+^2 + \lambda (Q_{k_b}(s_j,a_j;\theta_{t,k_b}) - U_j^{min})_+^2 \Big]. \tag{4.5}$$

The OT loss function given by eq. 4.4 is modified as shown in eq. 4.5 to accommodate the usage of several network heads. The loss is calculated for each head k_b based on the output of this head Q_{k_b} against a target y_{j,k_b} based on the head's own target network prediction. The parameters of each head are then updated by one gradient step using the designated optimization method. The upper and lower bounds are calculated by querying the periodically updated target network as in [21]. Note, that j is the position of a sample in the replay memory, which is needed to determine preceding and succeeding states within an episode to calculate the value bounds. The index j can be understood as the center within a sequence of states, where the center state s_j is used to calculate the predicted Q-value of the online network, while the target Q-value is calculated based on reward r_j and state s_{j+1}. Value bounds are always calculated w.r.t. the center state s_j.

$$y_{j,k_b} \leftarrow r_j + \gamma \max_a Q_{k_b}(s_{j+1}, \text{argmax}_a Q_{k_b}(s_{j+1}, a; \theta_t), \theta^-) \tag{4.6}$$

BDQN uses double DQN to calculate its targets, while He et al. mention that double DQN could result in further performance improvements, but it is not actually used for OT experiments in the paper. As double DQN has proven to solve overestimation problems, it will be used in this work unless stated otherwise. Furthermore, double DQN should lead to more accurate value bounds, which is essential to OT's bound tightening approach. The individual target for each is calculated as in equation 4.6. For double DQN the online network is used for optimal action selection, while the target network is used to calculate the q-value.
Initially, Osband et al. describe BDQN as a bootstrap method that trains each head on different data. However, in the Atari experiments performed by Osband et al. no bootstrapping is used and BDQN is reduced to an ensemble method instead. Based on the observations of Osband et al., each head is trained on all minibatch samples, i.e. data will be shared fully amongst all heads, in BOT. However, the use of an individual λ value for each head will be investigated. The idea is that better uncertainty estimates can be made through the use of different penalty weights, as this method should lead to a more diverse exploration. Furthermore, λ might be less sensitive to the choice of parameter value as it covers a wider range of values. A more detailed description of this approach is given in 4.3.2.
Similarly to OT and BDQN a minibatch of experiences is sampled periodically from the replay memory (see algorithm 3). Value bounds for the sampled experiences are then calculated for all network heads using the respective Q-value output of each head. The online network parameters are updated with one gradient step using equation 4.5. Figure 4.2 illustrates schematically how value bounds are incorporated into the DQN architecture. According to He et al. gradients

Input: Value function networks Q with K outputs $\{Q_{k_b}\}_{k_b=1}^{K_b}$.
Let $\mathcal{D}$ be a replay memory storing experience for training.
foreach *episode* **do**
  Obtain initial state from environment s_0
  Pick a value function to act using $k_b \sim \text{Uniform}\{1, ..., K_b\}$
  for *step* $t = 1, ...$ *until end of episode* **do**
    Pick an action according to $a_t \in \text{argmax}_a Q_{k_b}(s_t, a)$ or randomly based on ϵ
    Receive state s_{t+1} and reward r_t from environment, having taken action a_t
    Add $(s_t, a_t, r_{t+1}, s_{t+1})$ to replay memory $\mathcal{D}$
    if *Update network* **then**
      sample a minibatch of tuples $\mathcal{B} = \{(s_j; a_j; r_j; s_{j+1})\}$ from replay memory $\mathcal{D}$
      foreach *Network head* Q_{k_b} **do**
        Calculate lower and upper bounds for head Q_{k_b} using Eq. 4.2 and 4.3 respectively
        Update parameters θ_{k_b} of head Q_{k_b} with one gradient step of cost function given in Eq. 4.5

Algorithm 3: Bootstrapped Optimality Tightening

are rescaled after applying value bounds to maintain the gradient magnitude compared to the DQN version [21]. In chapter 5 the performance of methods with and without gradient rescaling is evaluated.

4.3.2 Implementation

The implementation of BOT is based on the DDQN algorithm with OT and BDQN as extensions. The ANN required to approximate the K different Q-functions is implemented using the open source machine learning library *TensorFlow*. The *Gym* toolkit is used as evaluation platform.
As suggested by He et al. in [21] λ is annealed over time, similarly to the decaying random exploration factor ϵ. Enforcing higher penalties for violating bounds while the state space is less explored should lead to more accurate Q-functions in early training due to faster reward propagation. Especially in early training Q-value estimates are usually unreliable, as the state space is insufficiently explored. With increased exploration of the state space, the penalties should be lower to loosen the bounds once the Q-function is able to make more accurate estimates and the state space is sufficiently explored. The penalty weight will be annealed linearly from λ_{start} to λ_{end} during training. The exact $lambda$ values and their annealing length will be discussed in 5, as these values differ depending on the environment that is evaluated on.
Concerning the BDQN part of the implementation, the number of network heads is set to $K = 10$ as Osband et al. suggest that more than 10 heads do not lead to a significant performance increase and thus would only result in increased computational requirements [4]. In the Atari evaluation in section 5.2 only 5 heads are used due to the required training time. While Osband et al. mention that either K different neural networks or one neural network with K outputs can be used to approximate the different Q-functions, they use one network with K heads. The BOT implementation also makes use of one network with K different outputs.
Using different heads for calculating the value bounds opens up some opportunities for a more diverse exploration. The effect of using different OT penalty weights $\lambda_{1:K}$ for different heads is investigated as a modified version of BOT, namely PM-BOT, in chapter 5. The penalty weights could be assigned at random for instance, however, this could introduce too much randomness into the learning process and may lead to unstable results. One possible approach is to assign the penalty weights based on an equally spaced interval between λ_{min} and λ_{max}. More precisely, based on the penalty coefficients used for BOT, the λ range for PM-BOT is defined as $[0, \lambda_{max} - \lambda_{max}/K_b]$ with a step size of λ_{max}/K_b.

4.3.3 Network Architecture

For the Atari evaluation, the neural network structure to approximate the Q-functions is the same as in [4], which is basically the same as in [21] with the modification that the network is split into several heads after the convolutional layers. Both network architectures are based on the original DQN architecture suggested by Minh et al. in [2]. Depending on the environment that is evaluated on different network architectures are used. Atari games utilize convolutional layers with raw pixel images as input. The *Gym* toolkit offers a series of environments that can be used to evaluate RL algorithms. Most of these algorithms have a state space represented by a scalar or a vector of scalar, i.e. a different network architecture has to be used for compatibility.
The BOT network architecture for Atari games is similar to the architecture proposed by Osband et al. in [4], which is similar to the model used in [2]. The network consists of 3 CL, followed by 2 FL:

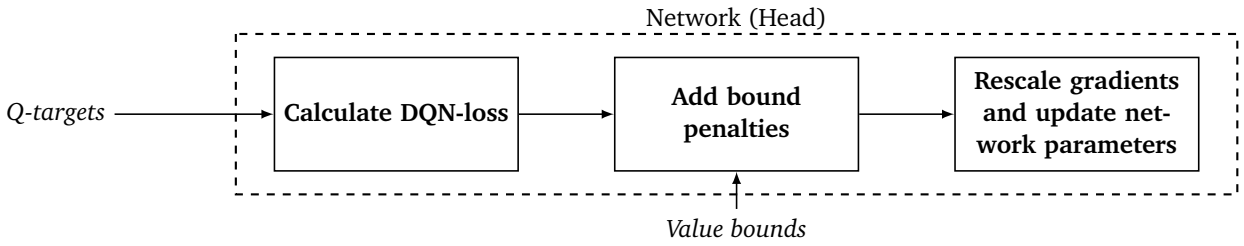


Figure 4.2: Illustration of how value bounds are incorporated into the DQN architecture. First, the basic loss between the target and Q-value prediction is calculated. The penalties for breaking value bounds are then added to the basic loss. After the penalties have been applied, gradients are rescaled, i.e. if one bound is broken the final error will be divided by 2 and in case of two broken bounds by 3.

- CL 1 (hidden): 32 filters of size 8x8, stride 4x4; Receives pixel image as input
- CL 2 (hidden): 64 filters of size 4x4, stride 2x2
- CL 3 (hidden): 64 filters of size 3x3, stride 1x1
- FL 1 (hidden): 512 neurons; Receives flattened input from CL 3
- FL 2: Outputs one value for each action available in the environment (no activation function)

Except for the output layer, all layers are followed by a ReLU activation function. The network is split into K heads after the convolutional part, where each head receives the same input. Each head consists of two FL, where the last layer in each head approximates one Q-function. The weights in all layers are initialized randomly using the Xavier uniform initializer, where samples are drawn from a uniform distribution within a limit that considers the number of units in a layer. The biases in all layers are initialized to 0.1.
For evaluation on classic control environments a different network architecture is used:

- FL 1 (hidden): Receives input in the form of a scalar vector
- FL 2 (hidden)
- FL 3: Outputs one value for each action available in the environment (no activation function)

The number of neurons per layer depends on the environment (see chapter 5 for details). Initialization is performed similarly to the Atari architecture and each layer (except for the output layer) is followed by a ReLU activation function. The network is split into K heads after the first FL, i.e. each head consists of the same number of layers as in the Atari setting.

5 Experiments

In this chapter experiments are conducted on a diverse set of environments using the BOT and PM-BOT algorithm introduced in chapter 4. The performance of both algorithms is compared against the baselines DQN, BDQN and OT. All experiments are performed in environments provided by OpenAI's Gym (https://gym.openai.com/). Gym is a toolkit for evaluating the performance of RL algorithms.

5.1 Evaluation on Classic Control, Toy Text and Box2D Environments

Eventually, evaluation is performed in the ALE. However, due to the required training time on Atari games, first evaluations are performed on simpler environments to verify the algorithms sanity.
As BOT is based on DQN, it can only deal with discrete action spaces. Possible environments satisfying this requirement are CartPole-v1, LunarLander-v2, FrozenLake-v0, FrozenLake8x8-v0 and Taxi-v2, which are included in Gym's classic control, Box2D and toy text suites. The environments are chosen based on their diverse objectives and partially for their requirements on exploration.

5.1.1 Environment Objectives

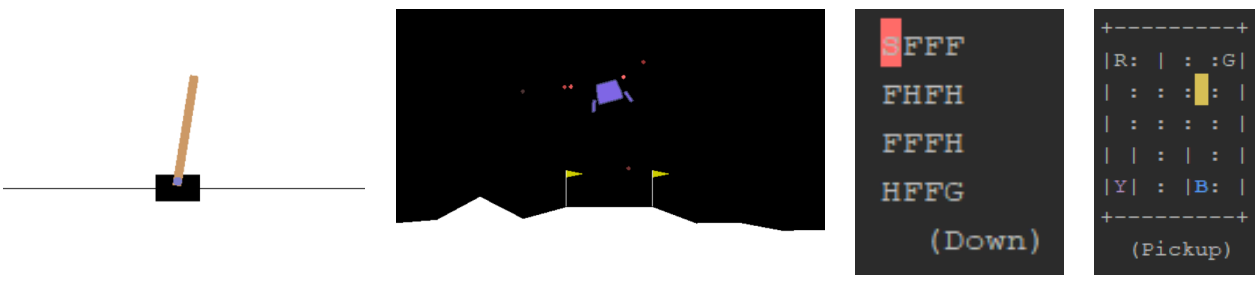
In the following, a brief explanation on the environments used for the first evaluation is provided.

CartPole-v1

CartPole-v1 is the more complex variant of the CartPole-v0 environment. The only difference is that the pole has to be balanced over a longer time. In the following all environments are referred to by their names without the version appendix.
CartPole is one of the more straightforward RL tasks. A pole is attached to a cart, while the cart is movable in horizontal direction on a rail. The pole starts in upright position and the goal is to balance it as long as possible. This can be achieved by moving the cart along the rail to balance the pole. Should the pole be in a position, where it is impossible to stabilize it again, the episode will be terminated. The agent can perform two different actions: apply force to move cart left/right. The environment provided by gym is based on the cart-pole problem described in [27].

LunarLander-v2

In LunarLander a spaceship has to be landed on the moon. The goal is to land between two flags on the ground. To achieve this the lander has to touch the ground softly, i.e. at low speed. While the flags are always at the same position, the landscape is initialized randomly at the start of each episode. The lander can be steered using thrusters at the sides and the bottom. Each time the main thruster, i.e. bottom thruster, is fired a small negative reward signal is returned. If the lander gets closer to the landing pad a positive reward signal is received and a negative one if it moves away from



CartPole-v1 LunarLander-v2 FrozenLake-v0 Taxi-v2

Figure 5.1: Illustration of the environments used in the first experiment.

Environment	Hidden Layer Size	Learning Rate	θ^- Update Frequency	Pre-train Steps	ϵ Annealing Steps
FrozenLake	[64, 32]	0.001	10	200	1000
FrozenLake8x8	[128, 64]	0.001	10	200	10000
CartPole	[64, 32]	0.001	100	200	10000
LunarLander	[256, 128]	0.005	500	500	20000
Taxi	[64, 32]	0.001	100	2000	50000

Table 5.1: DQN specific hyperparameter values for experiments conducted in control and toy environments of the Gym toolkit. Parameters were tuned coarsely using the DQN algorithm.

it. Getting the lander save to ground will issue a large positive reward signal, crashing it a large negative reward signal. There is no timestep limit. Each episode ends with a crash or safe landing.
The main challenge in this environment is to train the lander to land in as few episodes as possible. This environment is especially suitable for evaluating exploration, because the agent has no information about the orientation of the lander. However, the lander can only be safely landed on its legs. Hence, a sufficient deep exploration is needed to connect firing thrusters to actually landing on the landing pad. Furthermore, the agent has the option to do nothing, which would probably give a positive reward, because it gets closer to the landing pad. Firing the main thruster on the other hand gives a negative reward signal at first, but is essential to get the lander safely to ground in the long run.

FrozenLake

Both FrozenLake and FrozenLake8x8 basically present the same challenge. The agent has to cross a frozen lake and reach the goal which is always at the same position. Each position on the field is either safe (S), frozen (F), hole(H) or the goal (G). The agent starts on the safe position in the top left corner and has to find a safe path to the goal in the bottom right corner of the grid. Walking onto a hole makes the agent plummet to its death. Frozen parts are safe, but have a certain probability of the agent being stuck on the previous position, instead of transporting the agent to the next position. Due to this circumstance, the agent has to learn how to deal with uncertainty, as transitions between states are non-deterministic. The only difference between FrozenLake and FrozenLake8x8 is the size of the lake which is either 16 positions (4x4 grid) or 64 positions (8x8 grid) with the latter one resulting in a larger state space.

Taxi

Exploration is crucial to solve the Taxi environment, where the agent has to pick up passengers and transport them from one location to another. Rewards are received for successful drop-offs, which means that the agent has to establish some meaningful relationship between picking up passengers and dropping them off somewhere else with a possibly long sequence of moving actions in between. Furthermore, the agent receives a negative reward of -1 for every timestep and -10 for illegal drop-offs or pick-ups. The agent has 6 actions to chose from: move north/east/south/west, pick-up, and drop-off.

5.1.2 Experimental Setup

To evaluate the performance of the different algorithms, the reward obtained per episode during training is documented. As rewards per episode show a high variance, the average over the last 100 episodes is taken as measure of performance. DQN is also evaluated and serves as a baseline. However, the proposed algorithm BOT should ideally also perform better than OT and BDQN.
A neural network with two hidden layers, as described in section 4.3.3, is utilized to approximate the Q-function in this experiment. The input to this network is a state space in vector form and a scalar action. The network outputs a Q-value approximating the Q-function for each state-action combination.
An overview of all DQN specific hyperparameter values used in the different environments is presented in table 5.1, while further hyperparameter values related to BDQN, OT and BOT are displayed in table 5.2. All algorithms use the Adam optimizer with environment-dependent learning rate and an epsilon (Adam) value of 10^-8 to update the network parameters. The random exploration ϵ-value is linearly annealed from 1.0 to 0.01 over a varying number of steps. For each network update a batch size of 128 is used in all experiments. The number of environment steps between updating

Environment	λ Annealing Steps	λ_{max}	λ_{min}	K_{ot} (OT Range)	K_b (Network Heads)
FrozenLake	20000	2.0	0.8	4	10
FrozenLake8x8	20000	2.0	0.8	4	10
CartPole	10000	2.0	0.8	4	10
LunarLander	50000	2.0	0.8	4	10
Taxi	50000	2.0	0.8	4	10

Table 5.2: DQN variant hyperparameter values for experiments conducted in control and toy environments of the Gym toolkit. Parameters were tuned coarsely using the BDQN, OT and BOT algorithms.

the delayed target network parameters θ^- with the online network parameters θ varies depending on the environment. The maximum size of the replay memory is limited to 10^5 experiences for all environments, while the discount factor γ to compute the target values is set to 0.99.

5.1.3 Results

The performance of all algorithms in a first evaluation is visualized in figure 5.2. Performance is evaluated by considering the mean reward over the last 100 episodes as given by equation 5.1, where R_t is the total reward for episode t. OT, PM-BOT and BOT show solid performance in most environments. Considering its performance BOT and PM-BOT is usually between OT and BDQN. Both algorithms show strong early performance in CartPole, but fail to converge to an optimal solution (as all other algorithms). While BDQN diverges in LunarLander and CartPole, which results in an inability to solve these tasks properly, it performs well in the FrozenLake and Taxi environments. OT performs well in most tasks, but shows deficits in performance in both FrozenLake8x8 and Taxi. BOT shows solid overall performance, but has problems to converge to an optimal solutions.

$$\bar{R}_T = \frac{1}{T-t} \sum_{t=max(0,t-100)}^{T} R_t \tag{5.1}$$

Task specific optimization of the number of heads may lead to better results with all bootstrapping methods, as this parameter was not tuned during evaluation. Furthermore, it should be noted that the rewards documented here where observed during training, which means that no ensemble policy was used to choose actions while training with bootstrapping algorithms. To evaluate the true performance of the bootstrapping algorithms, actions need to be chosen based on a majority vote between all network head predictions, as is done when evaluating on Atari environments.
Osband et al. suggest that agents focusing on exploration, i.e. gaining high rewards as fast as possible, should actually be evaluated using the cumulative episode reward [4], i.e. the sum of rewards over all past episodes. The performance of all agents considering the cumulative reward can be seen in figure 5.3. The cumulative reward is computed by adding up the 100 episode mean rewards (cf. 5.2) of all preceding episodes up until the current episode as in eq. 5.2. Using cumulative reward as a measure of performance, the advantage of OT variants over vanilla DQN becomes more obvious. As can be seen, early performance of OT variants is usually better than with other methods. In FrozenLake8x8 BDQN performs best, while all methods show no significant performance difference in FrozenLake.

$$CR_T = \sum_{t=0}^{T} \bar{R}_t \tag{5.2}$$

He et al. actually scale the gradient after applying the value bound penalty to be comparable to the gradient without penalty [21]. As scaling the gradients could stabilize training and help in converging to an optimal solution, the results with rescaled gradients can be seen in 5.4.
Rescaling the gradient leads to better results, especially in early training, in LunarLander for all OT based methods. However, OT and BOT have problems solving the Taxi environment, while PM-BOT finds an optimal solution in Taxi and shows among the best performance in almost all environments. glsbot and PM-BOT still fail to converge to an optimal solution in LunarLander. Considering the cumulative reward PM-BOT shows either best performance or close to best performance making it the best method overall when evaluating obtained rewards in the set of environments presented here.

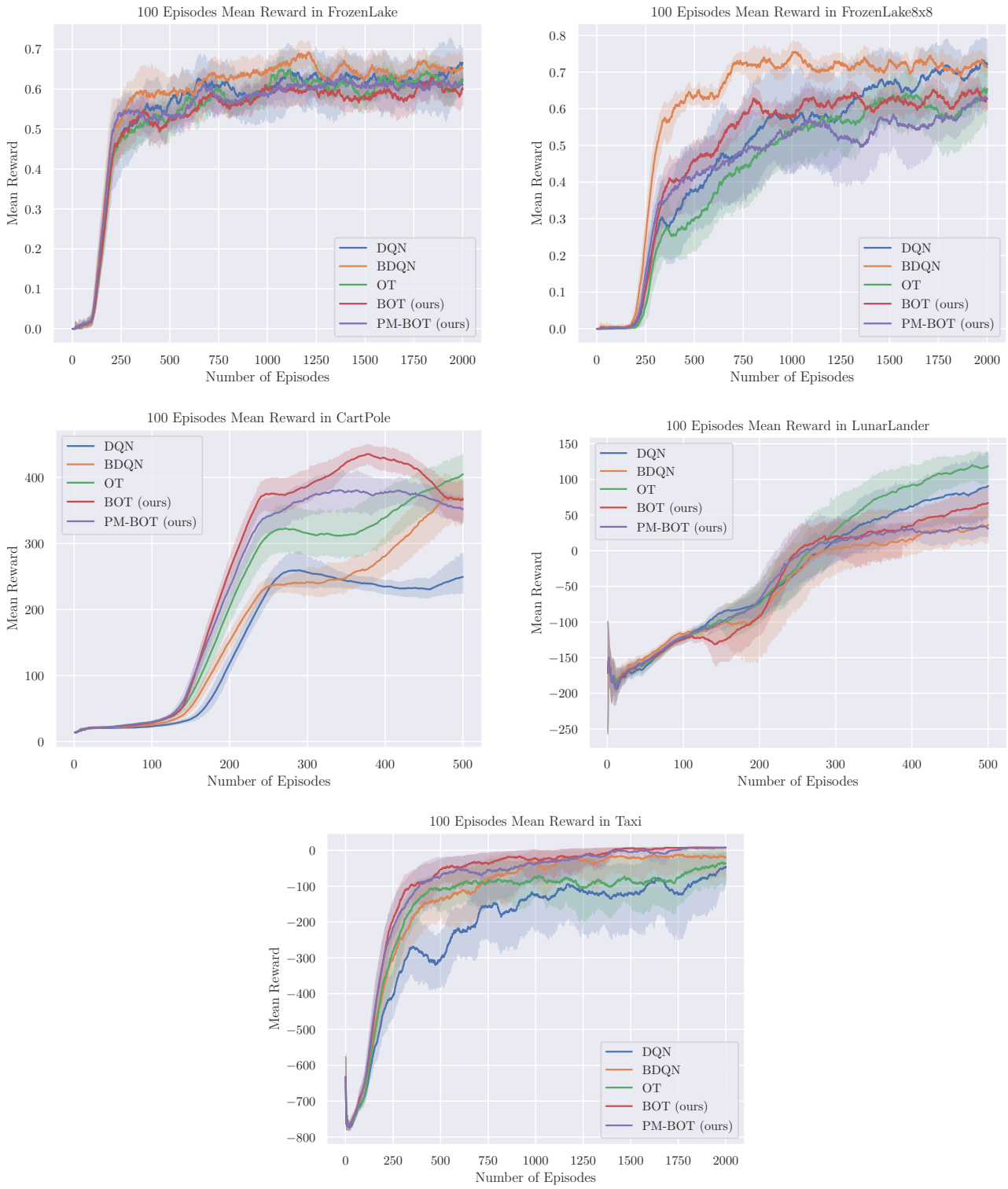


Figure 5.2: The results shown here are obtained using OT variants *without* gradient rescaling after applying penalty bounds. It shows the mean total episode reward over the last 100 episodes during training given by equation 5.1. Each experiment (environment/algorithm) was run ten times with different random seeds. The results shown here are the averages over these ten experiments. The shaded are around the curves displays the confidence interval with a confidence level of 95 %.

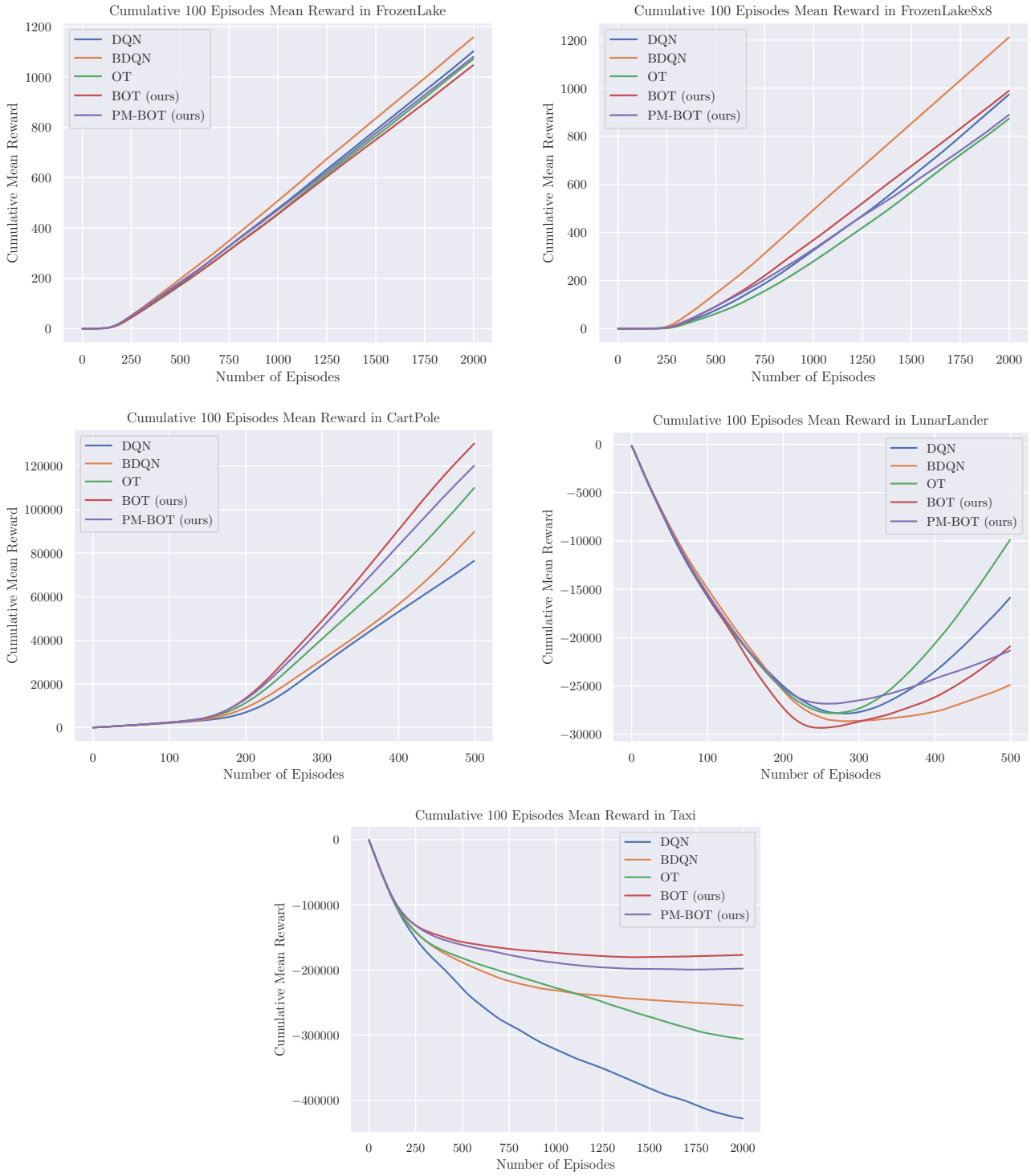


Figure 5.3: The results shown here are obtained using OT variants *without* gradient rescaling after applying penalty bounds. It shows the cumulative reward obtained when summing up the mean rewards from figure 5.2 until the current episode as described in equation 5.2. Each experiment (environment/algorithm) was run ten times with different random seeds. The results shown here are the averages over these ten experiments.

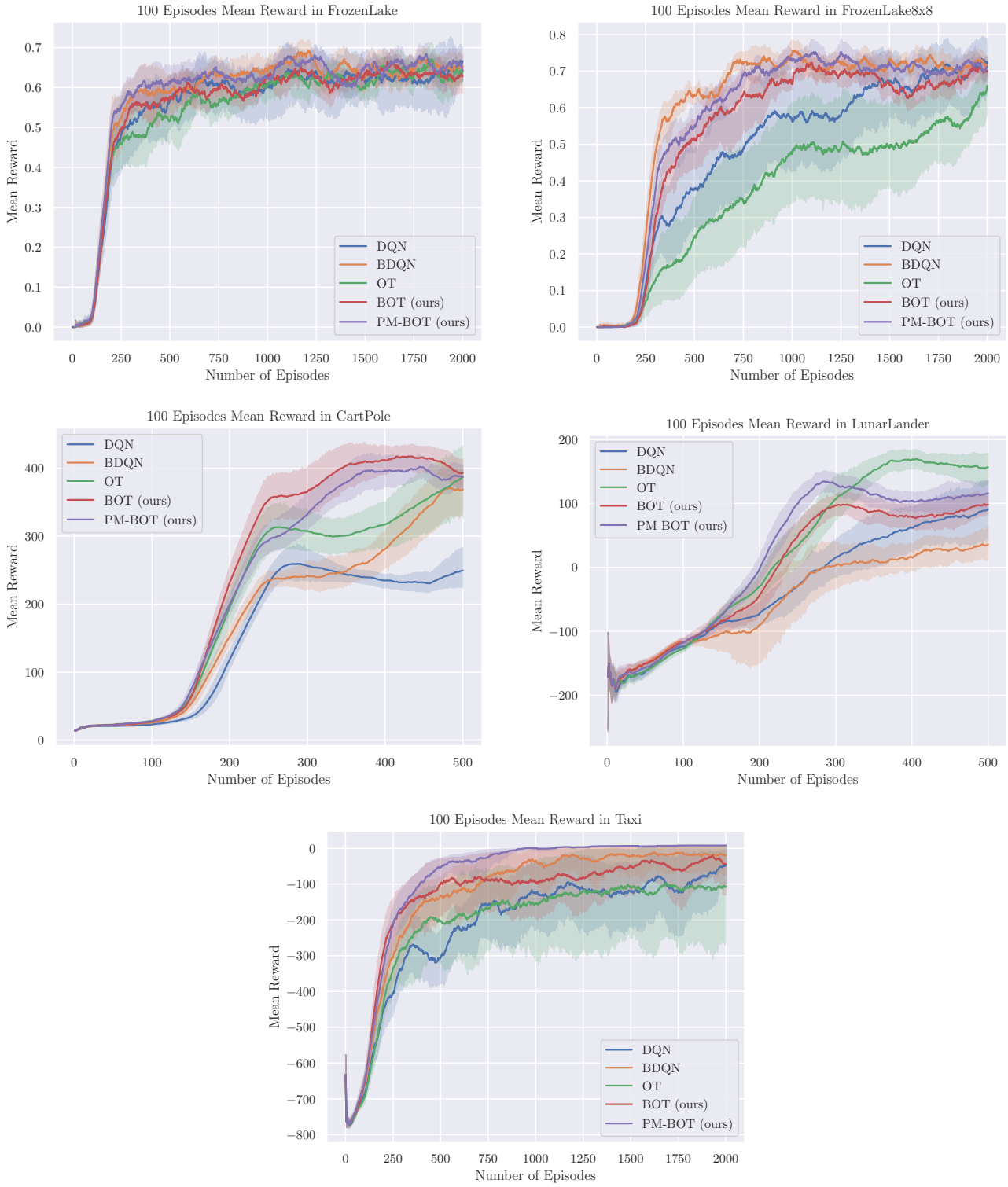


Figure 5.4: The results shown here are obtained using OT variants *with* gradient rescaling after applying penalty bounds. It shows the mean total episode reward over the last 100 episodes during training given by equation 5.1. Each experiment (environment/algorithm) was run ten times with different random seeds. The results shown here are the averages over these ten experiments. The shaded area around the curves displays the confidence interval with a confidence level of 95 %.

The confidence interval with a confidence level of 95 % is plotted as a shaded area around the mean rewards in figure 5.2 and 5.4. 10 random seed runs are used to compute the confidence intervals. The width of the interval visualizes over which area 95 % percent of the rewards are distributed, i.e. the smaller the interval the more consistent are the results for a number of random seed runs. Also, the interval indicates how prone the experiments are to different choices of random seeds. As can be seen in figure 5.2 and 5.4 the confidence intervals show no significant difference between different methods in CartPole and LunarLander. However, in FrozenLake8x8 and Taxi the confidence intervals for PM-BOT and BOT are significantly smaller compared to OT, which shows the benefit of using BDQN in some settings.
BOT and PM-BOT show strong early performance but then plateau on a suboptimal solution in CartPole and LunarLander. Looking at the time around which the curves level out, it seems the performance stops once the penalty coefficient λ is annealed. The assumption is, that there is a correlation between a degrading λ and the stagnant learning curve. Furthermore, the issue of leveling out performance seems to occur with all bootstrapped based methods. To investigate the issue of performance drops with bootstrap methods in later training, BOT and PM-BOT are changed to output only one Q-function once λ is annealed, reducing it to regular OT in later training. However, as is shown in figure 5.6, reducing the bootstrap heads in BOT and PM-BOT does not significantly increase obtained rewards in CartPole and LunarLander, while the performance in FrozenLake8x8 is significantly worse compared to the results reported in figure 5.4. For this experiment all OT methods use gradient rescaling as in the experiments reported in figure 5.4. It may be crucial to find the right moment in training to reduce bootstrap methods to one head, which could not be investigated any further due to time constraints and will be left to future work. More fine tuning on the λ parameter and the annealing length for λ could lead to better results in general. Eventually, it would be better to choose λ based on the the current learning situation and environment, i.e. the value of λ should be adjusted based on the agent's performance.

Evaluation

The mean Q-value estimates during training are calculated by taking the mean over all predicted Q-values per episode following the current policy π (see eq. 5.3), where T in this context is the length of an episode. As can be seen in figure 5.7, BOT and PM-BOT generally give smoother Q-value estimates with less sudden changes. In CartPole for instance, BDQN and DQN show a strong rise in Q-value estimates around episode 200, which then decreases again in later episodes for DQN, while BDQN overestimates the Q-value throughout training. All OT based methods show a smoother increase in Q-value estimates in CartPole. Surprisingly OT shows better performance without gradient rescaling in FrozenLake8x8, while the Q-value estimates suggest significantly less overestimation with gradient rescaling.

$$\bar{Q}^{\pi} = \frac{1}{T}\sum_{t=0}^{T} Q^{\pi} \tag{5.3}$$

Overall OT variants show a smoother learning curve, which is reflected in the Q-value estimations, where values are generally lower with less sudden jumps compared to DQN and BDQN.
A t-test is performed to test how statistically significant the proposed methods are compared to the best observed results. A (two-sided) t-test basically tests how much the average value differs between two independent samples. The null hypothesis states that two independent samples have identical average values. The hypothesis is then either rejected or not rejected based on a predefined threshold (usually 5 %). Whenever the outcome of a t-test, which is represented by the p-value, is higher than this threshold the null hypothesis can not be rejected, i.e. there seems to be no statistical significant difference between two samples. If the p-value is smaller than the threshold the null hypothesis is rejected, i.e. there is a statistical significant difference between two samples.
The results from section 5.1.3 are used as input for the t-test. The first sample input to the t-test is the maximum observed mean reward for each environment considering all methods (eq. 5.4a), while the second input is the maximum mean reward per method (eq. 5.4b). The t-test is performed for the OT variants without rescaling the gradient after applying penalty bounds and with rescaling (see table 5.3 for results). Although no null hypothesis is rejected, it can be seen that all OT based methods have the highest similarity to the best reported results due to a higher p-value. OT has a higher p-value than PM-BOT without gradient rescaling, but PM-BOT reports the highest p-value with gradient rescaling. The same behavior can be observed between BOT and OT in the opposite direction. When considering both cases, with and without gradient rescaling, PM-BOT shows the highest similarity to the best reported results on average.

$$R_{max}(env) = max(\bar{R}_{env,all-methods}) \tag{5.4a}$$

$$R_{max,method}(env, method) = max(\bar{R}_{env,method}) \tag{5.4b}$$

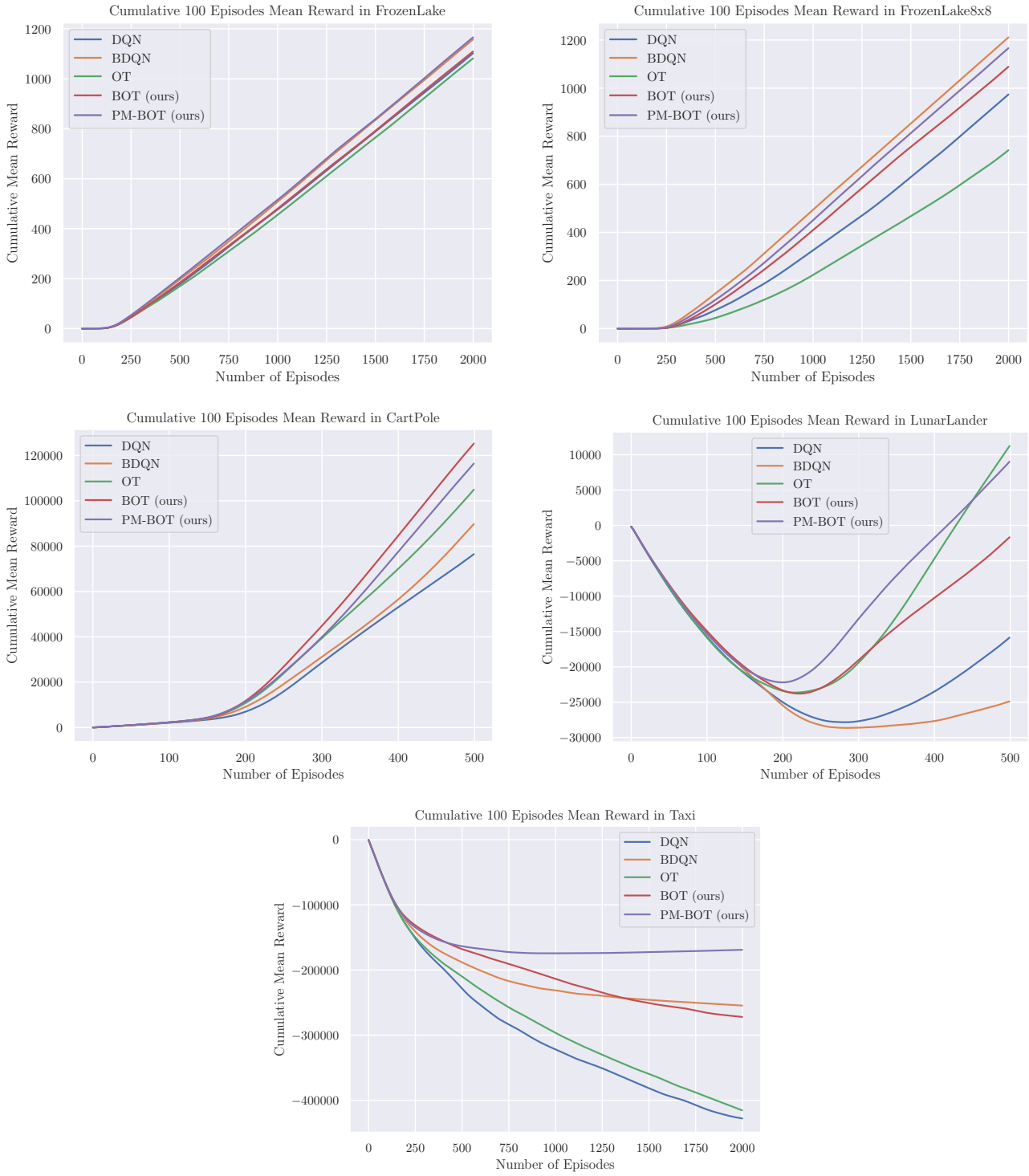


Figure 5.5: The results shown here are obtained using OT variants *with* gradient rescaling after applying penalty bounds. It shows the cumulative reward obtained when summing up the mean rewards from figure 5.4 until the current episode as described in equation 5.2. Each experiment (environment/algorithm) was run ten times with different random seeds. The results shown here are the averages over these ten experiments.

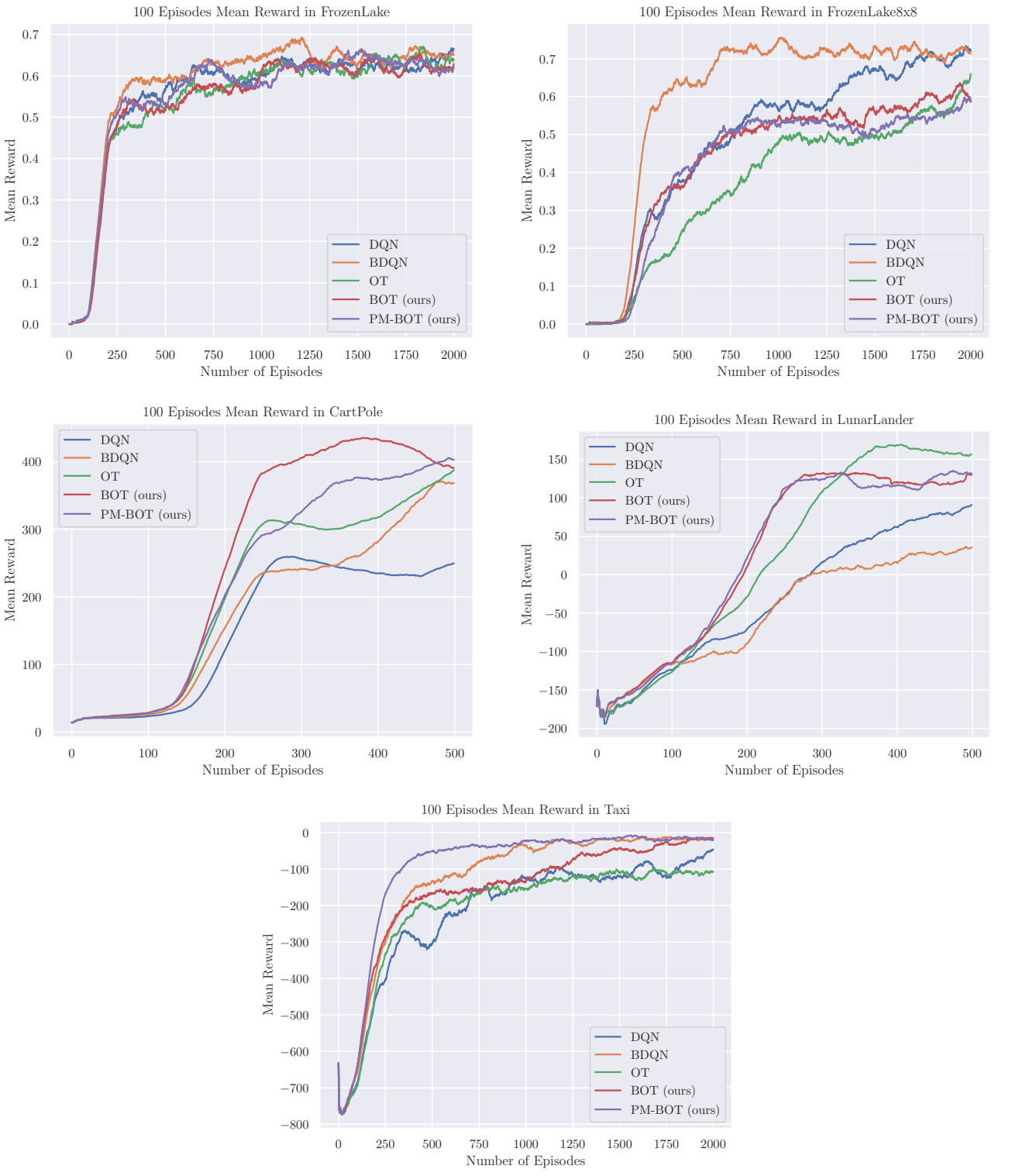


Figure 5.6: The results shown here are obtained using OT variants *with* gradient rescaling, where for BOT and PM-BOT the number of approximated Q-functions is reduced to 1 once λ is annealed to λ_{min}. It shows the mean total episode reward over the last 100 episodes during training given by equation 5.1. Each experiment (environment/algorithm) was run ten times with different random seeds. The results shown here are the averages over these ten experiments.

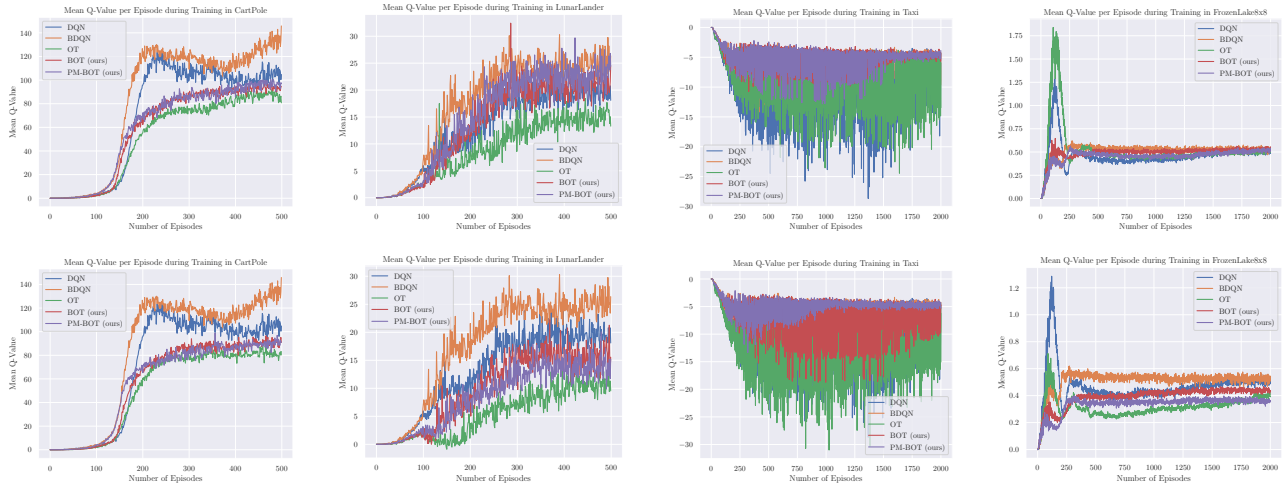


Figure 5.7: Mean predicted Q-values for all methods without gradient rescaling (top) and with gradient rescaling (bottom) after applying bound penalties during training. These values are calculated by taking the mean over all Q-value predictions (see equation 5.3) for actually chosen actions per episode. Again, the mean is taken over ten experiments with different random seeds.

	DQN	BDQN	OT	BOT	PM-BOT
OT Methods without Gradient Rescaling	62.0 % (+)	77.1 % (+)	90.4 % (+)	93.1 % (+)	87.0 % (+)
OT Methods with Gradient Rescaling	57.1 % (+)	72.6 % (+)	82.3 % (+)	80.1 % (+)	93.1 % (+)

Table 5.3: This table shows the p-values (in percent) for t-tests between the best mean reward of each method against the best observed mean reward overall. The t-test uses the mean rewards for OT methods without gradient rescaling (figure 5.2) and the mean rewards for OT methods with gradient rescaling (figure 5.4) as input data. A threshold of 5 % is assumed to test the statistical significance between two samples. '-' means the null hypothesis is rejected, '+' means the null hypothesis can not be rejected.

5.1.4 Preliminary Discussion

The proposed algorithms BOT and PM-BOT show solid overall performance in a first evaluation. Although OT obtains higher final rewards in LunarLander, it has problems to solve FrozenLake8x8 and Taxi. As is reflected in the performance, BOT and PM-BOT successfully combine the benefits of OT and BDQN as they overcome issues the base algorithms have with solving some of the environments.

5.2 Evaluation on Atari Games

The ALE provides a number of games which offer a vast amount of different tasks to solve. Developing one agent to perform well on many of these games, given the same hyperparameters and network structure, has been a challenging task and still offers lots of room for improvement. Deepmind has laid a foundation and fueled competition among researches to set new baselines in this domain when they achieved human-level performance on Atari using a DQN agent [2].
OpenAI's Gym provides a wrapper for the ALE, which makes it easier to interact with the environments. Furthermore, OpenAI provides an additional wrapper for Atari to emulate the configuration used by Deepmind, which is used for evaluation here.

5.2.1 Exploration Challenge

Rewards in Atari games are usually sparse, i.e. there are long sequences of actions without a reward other than 0, due to the nature of the game objectives, where rewards are only given when following a long sequence of adequate actions. This inevitably leads to a challenge of correlating actions to rewards, especially in Q-learning, where rewards are propagated slowly. One of the most challenging games for instance is Montezuma's Revenge, where positive rewards are only returned after a series of dependent quests have been solved.
Atari games are a good testbed to evaluate an agent's ability to explore. While the basic DQN agent relies primarily on ϵ-greedy exploration, many attempts have been made to develop a more sophisticated exploration strategy. The proposed algorithms focus on fast reward propagation and a diverse exploration through multiple network heads, which should make them suitable to tackle exploration problems in the ALE.

5.2.2 Experimental Setup

Machado et al. summarize the most common evaluation procedures on the ALE [28]. The evaluation presented here follows the evaluation and training procedure of [2] as is done by [21] and [4] as well. The Atari wrapper provided by OpenAI offers some functionality to emulate the desired behavior:

- Rescale inputs: Images (states) observed in Atari games are rescaled to size 84x84, which is required to feed the inputs to the convolutional network (equal width and height) and cuts unnecessary information at the top and bottom of the screen. The images are also converted to gray-scale.
- Stacked frames: The last 4 frames in an observation sequence are stacked to include motion information. The stacked state sequence is fed to the network as input.
- Frameskip: The agent actually only observes every k-th frame, while skipping frames in between by repeating the last executed action k times.
- Episodic life: Loosing a life marks the termination of an episode. While the game is not actually reset on life loss, this information is saved in the replay buffer to let the agent know that loosing lives is bad. The Q-value returned by the network will be set to 0 once the agent transitions into a terminal state.
- Fire to reset: Some environments require the agent to additionally execute an action to start a new episode.
- 30 no-op start: Each episode starts with a random number (between 0 and 30) of 'none' operations to randomize the start state. 30 no-op also helps to generalize, as Atari games itself are otherwise deterministic.

Furthermore, all pixel values are normalized from $[0, 255]$ to $[0, 1]$ before being fed to the network. The scale of the reward signal returned by different environments in the ALE can differ by magnitudes. A naive approach to deal with vastly different magnitudes of rewards is to clip them to $[-1, 1]$ as is done by many agents [2, 21, 7]. Although clipping

Learning Rate	θ^- Update Frequency	Pre-train Steps	ϵ Annealing Steps	γ
0.000065	10000	25000	250000	0.99
λ Annealing Steps	λ_{max}	λ_{min}	K_{ot}	K_b
500000	8.0	0.0	4	5

Table 5.4: List of hyperparameter values for the Atari experiment.

rewards leads to information loss, as no difference between different scales of rewards outside these bounds is made, it has proven to stabilize learning. Note, that there is a difference between agent steps and actual environment frames due to the frameskip. While the training time is measured in environment frames (frame skip · agent steps), parameters are assigned w.r.t. agent steps.

Agent Parameters

The convolutional network architecture is described in 4.3.3. Instead of 10 heads, as in the first experiment, only 5 network heads will be used in this experiment to reduce the computational time. Each head increases the computation time significantly, because value bounds need to be calculated explicitly for each head. The hyperparameter values for all environments are the same, as is common when evaluating on the ALE. A list of hyperparameter values for this experiment is given in table 5.4. Instead of using the RMSProp optimizer as in [2], the Adam optimizer with the same learning rate will be used as was done in the Rainbow implementation by Deepmind [17]. The agent follows an ϵ-greedy policy with linear annealing from 1.0 to 0.01. The size of the replay memory is limited to 1M experiences. Huber loss is used as loss function similarly to [2].

To save on training time 8 frames will be skipped between each agent interaction with the environment, while [2] skips only 4 frames. During evaluation on Atari games it was observed that when using OT based algorithms in combination with a frame skip of 8, the performance does not decrease significantly and in some environments even higher rewards are obtained. One possible explanation for this behavior is that with a higher frame skip, the frames delivered to the algorithm for training are further apart and, hence, stacked frames may contain more information. Note, however, that a higher frame skip also leads to more information loss in between observable frames, because certain behaviors are lost in skipped frames. As OT makes use of state sequences to calculate value bounds, this increase in information of stacked frames may lead to more accurate value bounds. Furthermore, a higher frame skip drastically decreases computational costs considering training time in terms of environment frames, as network parameters are updated every n agent steps and every agent step skips 8 frames instead of 4. Therefore, around double the frames can be observed in the same training time.

5.2.3 Results

The results reported here follow the common 30 no-op evaluation procedure introduced by Deepmind in [2]. An agent's performance is evaluated after each epoch by playing for 30 episodes with a maximum length of 18000 environment frames per episode, which corresponds to 2250 agent steps in the case of 8 skipped frames between agent interactions. ϵ is set to 0.05 during evaluation and for all bootstrap methods the action predicted by most heads is chosen as best action. There are no experiences saved to the replay memory during evaluation. Similarly to [21] the agent is trained for 250000 environment frames per epoch, which corresponds to 31250 agent steps due to a frame skip of 8. Due to limited time, the computation limit of one day on the cluster computer and the extensive computation costs of OT methods, the agent is only trained for 4M environment frames, or 16 epochs, in total. The results are reported here for completeness, but it's difficult to make them comparable to the results reported in [21, 4] due to vastly different training lengths.

The results in figure 5.8 show that BOT performs best on average among the evaluated methods. PM-BOT does not learn a useful policy in Breakout and DoubleDunk, while it performs best in Frostbite. Only 8 environments were evaluated due to time constraints, while evaluation in the ALE is usually performed on 49 games. In VideoPinball and ChopperCommand none of the evaluated methods performs well. Due to the vast difference in objectives between Atari games, it is difficult to find one method that performs well in all environments.

Table 5.5 shows the results in terms of best no-op evaluation over all epochs and both random seeds. These values are usually used for comparison when evaluating on the ALE. However, due to the small amount of frames used for training, it is difficult to make the results comparable to results reported in [2, 4, 21].

Figure 5.8: 30 no-op evaluation results for selected Atari games. Evaluation is performed after each epoch by running the agent for 30 episodes and documenting the average over all episode rewards. The reported results are the average over 2 random seed runs.

	OT (4M)	BOT (4M)	PM-BOT (4M)	DQN (200M)	BDQN (200M)	OT (10M)
Atlantis	65543.33	74283.33	62896.67	85641	994500	316766.67
Breakout	22.03	27.73	3.47	401.2	855	229.79
ChopperCommand	946.67	836.67	773.33	6687	4100.0	6360
DoubleDunk	-20	-15.2	-21.6	-18.1	3	-10.07
Frostbite	590	841.33	1335.67	328.3	2181.4	3974.11
Krull	6144.87	7315.97	6748.77	3805	8627.9	9461.1
Qbert	2225	3869.17	3677.5	10596	15092.7	12355
VideoPinball	11189.97	21523.23	11516.77	42684	811610	74873.2

Table 5.5: Maximal 30 no-op scores for selected Atari games (highest score per epoch observed during evaluation). The scores on the right hand side of the table are the officially reported results from the corresponding papers. The scores on the left hand side are results obtained using the implementation of this thesis (2 random seed runs). The number of environment frames used for training are given in parenthesis.

5.2.4 Preliminary Discussion

The ALE offers many environments with vastly different objectives. The two proposed methods BOT and PM-BOT are compared against OT, which these methods build on. Among the compared methods BOT shows the most stable performance on the evaluated Atari games. The evaluation on the ALE offers some challenges, including extremely long training times, which limits possible evaluations that can be performed in the scope of this thesis.

6 Discussion

In this thesis methods to improve exploration in DRL have been explored. Value bounds can be used in combination with DRL for fast reward propagation and, thus, faster learning, especially in early training. He et al. have shown that even complex environments in the ALE can be solved in feasible time [21]. Based on the method proposed by He at al., which uses value bounds in combination with DQN, an extension was developed to combine value bounds with bootstrapped DQN. While OT focuses on fast reward propagation, BDQN induces more diversity into the exploration strategy by approximating multiple value functions.
It has been shown that the proposed methods perform well in several RL environments in early training, while struggling to converge to an optimal solution in some cases. BOT and PM-BOT can overcome some of the downfalls of OT and BDQN. While OT and BDQN perform well on a selection of environments, BOT and PM-BOT perform well on most environments. However, as has been shown in evaluations, the proposed methods have problems to converge to an optimal solution in several cases. The issue of an almost abrupt stop in learning was investigated but could not be resolved. Cumulative rewards were documented and show promising results, which is primarily attributable to a strong early performance of the proposed methods, although a suboptimal performance in later stages still persists. Concerning the consistent performance w.r.t. different random seed runs, it has been shown that BOT and PM-BOT either show around the same or better consistency depending on the environment.
Evaluation on the ALE posed a challenge due to time constraints and the extensive computation time required to solve these environments. Calculating the value bounds exactly already requires a lot more computational resources than DQN, as the network needs to be queried for Q-values for all state sequences required to calculate the bounds. Calculating the bounds for all network heads when combined with BDQN results in an even higher required computation time. Hence, BOT and PM-BOT were evaluated on significantly fewer frames as common for ALE experiments. In this small experiment BOT performed better than OT, while PM-BOT had stability issues in several environments.
While the early performance of the proposed methods is mostly better than with other evaluated methods, BOT and PM-BOT still have problems converging to an optimal solution in some environments. In the scope of this thesis it was not possible to find the cause of this issue. Hence, investigating converge problems of the proposed methods will be left to possible future work.
This thesis laid a foundation for a possible combination of value bounded optimization with bootstrapped DQN. The proposed method was implemented and evaluated w.r.t. its performance in several environments. It has been shown, that a combination offers potential benefits in terms of early learning, but also has stability issues in later training, which need to be resolved to achieve good performance consistently.

6.1 Outlook

The proposed methods still pose several issues to be solved. The problem of time complexity may be tackled by calculating value bounds exactly in early training and approximate them using for instance a neural network once bounds can be approximated with sufficient precision. Value bounds are especially important in early training and significantly improve performance in this stage. Hence, it is maybe sufficient to use value bounds only in early training, while reducing the algorithm to basic DQN or BDQN once the penalty coefficient is annealed.
A more sophisticated approach at determining the penalty coefficient could be used. When adjusting the value of the penalty weight based on the agent's Q-value estimates, or some other measure, performance may increase further. Especially in Atari games, where hyperparameters are consistent across all environment, an adaptive penalty weight may lead to substantially better results.
The issue of early converge to an suboptimal policy was investigated, but could not be resolved. Reducing BOT and PM-BOT to agents with one Q-function output may resolve the issue of an early plateau in performance, but it is not directly obvious when the method should be reduced to a normal OT method. Developing a more sophisticated approach to reducing the number of heads during training will be left to future work.
Several methods that use value bounds for optimality tightening were investigated, but only one of them was readily combinable with DRL. It would be interesting to investigate, how other optimality tightening approaches could be combined with DRL. Furthermore, value bound tightening could be combined with an algorithm for continuous environments (actor-critic) like for instance the A3C algorithm.

Bibliography

[1] R. S. Sutton, *Reinforcement learning*. Adaptive computation and machine learning, Cambridge, MA: The MIT Press, second edition ed., 2018. Literaturverzeichnis: Seite [481]-518.

[2] 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.

[3] T. Smith and R. G. Simmons, "Heuristic search value iteration for pomdps," vol. abs/1207.4166, 2012.

[4] I. Osband, C. Blundell, A. Pritzel, and B. V. Roy, "Deep exploration via bootstrapped DQN," *CoRR*, vol. abs/1602.04621, 2016.

[5] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[6] G. Hinton, "Neural networks for machine learning," 2012. Available at `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

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

[8] M. Riedmiller and H. Braun, "Rprop - a fast adaptive learning algorithm," 1992.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[10] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, July 2011.

[11] J. Kirkpatrick, R. Pascanu, N. C. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *CoRR*, vol. abs/1612.00796, 2016.

[12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[14] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 2613–2621, Curran Associates, Inc., 2010.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[16] T. Pohlen, B. Piot, T. Hester, M. G. Azar, D. Horgan, D. Budden, G. Barth-Maron, H. van Hasselt, J. Quan, M. Večerík, M. Hessel, R. Munos, and O. Pietquin, "Observe and look further: Achieving consistent performance on atari," 2018.

[17] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *CoRR*, vol. abs/1710.02298, 2017.

[18] M. Hauskrecht, "Incremental methods for computing bounds in partially observable markov decision processes," in *AAAI/IAAI*, 1997.

[19] M. Hauskrecht, "Value-function approximations for partially observable markov decision processes," *CoRR*, vol. abs/1106.0234, 2011.

[20] G. Shani, R. I. Brafman, and S. E. Shimony, "Forward search value iteration for pomdps.," in *IJCAI*, pp. 2619–2624, 2007.

[21] F. S. He, Y. Liu, A. G. Schwing, and J. Peng, “Learning to play in a day: Faster deep reinforcement learning by optimality tightening,” 2016.

[22] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, “#exploration: A study of count-based exploration for deep reinforcement learning,” *CoRR*, vol. abs/1611.04717, 2016.

[23] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, “VIME: Variational Information Maximizing Exploration,” *arXiv e-prints*, p. arXiv:1605.09674, May 2016.

[24] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, “Unifying count-based exploration and intrinsic motivation,” vol. abs/1606.01868, 2016.

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

[26] R. Y. Chen, S. Sidor, P. Abbeel, and J. Schulman, “Ucb exploration via q-ensembles,” 2017.

[27] A. G. Barto, R. S. Sutton, and C. W. Anderson, “Neuronlike adaptive elements that can solve difficult learning control problems,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, pp. 834–846, Sept. 1983.

[28] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. J. Hausknecht, and M. Bowling, “Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents,” *CoRR*, vol. abs/1709.06009, 2017.

[29] I. Osband, B. Van Roy, and Z. Wen, “Generalization and Exploration via Randomized Value Functions,” *ArXiv e-prints*, Feb. 2014.

[30] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” 2012.

[31] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling network architectures for deep reinforcement learning,” 2015.

[32] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” 2015.