Exploration in Deep Reinforcement Learning

Exploration in Deep Reinforcement Learning
Bachelor-Thesis von Markus Semmler aus Rüsselsheim
Tag der Einreichung:

1. Gutachten: Ph.D. Riad Akrour
2. Gutachten: Prof. Dr. Jan Peters
3. Gutachten:

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Exploration in Deep Reinforcement Learning
Exploration in Deep Reinforcement Learning

Vorgelegte Bachelor-Thesis von Markus Semmler aus Rüsselsheim

1. Gutachten: Ph.D. Riad Akrour
2. Gutachten: Prof. Dr. Jan Peters
3. Gutachten:

Tag der Einreichung:

For my Family and Friends

Erklärung zur Bachelor-Thesis

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

Darmstadt, den 23. Oktober 2017

_____

(Markus Semmler)

Abstract

Deep Learning techniques have become quite popular. Over the past few year they have also been applied to Reinforcement Learning. A main challenge is the Exploration-Exploitation Trade-Off. Very much theoretical work exists, which perform very good on small scale problems. However most of the theoretically interesting topics, can't be scaled easily to arbitrary state spaces, which are present in everyday life. This thesis splits into two parts. The former analyzes and lists various exploration techniques, to get familiar with the parameters or to detect weaknesses. It is then continued by evaluating them on more small scale MDP's. Besides that two new algorithms are proposed, whereas one was also applied to the small MDP's and was the only one to completely solve an instance of Deep Sea Exploration. The second algorithm is not applicable to tabular algorithms and uses regularization combined with DDQN for driving exploration. It was evaluated on MountainCar-v0 using three different regularization techniques. On the task it was capable of reaching a good performance rapidly. The final model successfully converges to one solution, repeating it over and over again and most important to not forget about it's knowledge again.

Contents

Figures and Tables

List of Figures

List of Tables

1 Introduction

Recent advances in AI-Technologies have solved a lot of complex tasks. The Deep Q Network (DQN) architecture was first capable of playing Atari games at a human level [8]. It's actions are based solely on the raw input pixel data - one of the main steps in designing a generally closed perception system. However a prominent example where Deep Q Networks (DQN) fail is Montezuma's Revenge. In this game the player has to find several keys to unlock doors unveiling the way to higher levels. The main reason for the malfunction is insufficient exploration in regions of sparse rewards. As these structures of feedback can also be found in real life tasks, e.g. compare Geo-Caching and Montezuma's Revenge, it plays a crucial role on how the agent gets sense of the environment.

Interesting problems, especially tasks which are easy solvable by humans, suffer from the curse of dimensionality in the sense that the state space size is usually exponential in the number of dimension. It is thus not possible to brute force all possible combinations of actions and save the best. There is increasing need for clever strategies on how to explore the environment and when the agent should explore.

The natural counterpart of Exploration is Exploitation. Both together form one of the fundamental challenges in RL - the Exploration-Exploitation-Trade-Off (EET). If knowledge is obtained about the environment, the agent has to use it in order to receive more points. Logically pure random exploration is not sufficient to get high rewards. Therefore imagine a human who has recently moved to a new city. During the first week he explores the city center taking some randomly chosen routes through it. With this technique he is going to find some spots, which match his personal interests. However if he decides to visit to the best places, he found during the first week. Obviously there might be other places, which are even better, but he doesn't find them, because he stopped exploring the environment.

When the problems scale up, several features are needed for efficient learning. First of all there is the need to get exhaustive knowledge about the environment, without visiting all state-action pairs. The knowledge has to be generalized to unseen examples, which gives rise to the use of neural networks as a function approximator in the closed learning cycle. Although this idea is pretty naive approaches don't perform well or even diverge [8].

In order to solve the Trade-Off several different strategies have been proposed. There are very simplistic ones, e.g. ϵ-Greedy strategies, which were also used in DQN networks. [8]. The problem with these simplistic strategies is that they tend to need exponential many examples of states and actions and thus suffer from the curse of dimensionality as well. There have been interesting approaches, where some of them scale very well and remain computationally tractable and others not.

This thesis gives an overview of old exploration strategies, it then continues by reviewing recently proposed techniques. Therefore a categorization of them was made and presented. The main goal is to leave out all unnecessary details while focusing on the functioning and connection between them. By executing them on small scale MDP's, not only knowledge about the inner workings and how the parameters influence the performance could be obtained, but also on why some problems are harder for some agents and others not. Furthermore two algorithms are proposed, whereas the first simply combines two existing exploration strategies [9] and [2] into one unified strategy. The second was inspired by some Bayesian approximation techniques [10], the main idea is to use an appropriate regularization method for exploring the environment, which get extended by ideas from [2]. A final evaluation on the MountainCar is done and presented.

2 Foundations

2.1 Basics

2.1.1 Markov Decision Process

The theory of Reinforcement Learning (RL) builds up on the concept of a Markov Decision Process (MDP). The idea is to provide a formalism, which inherently contains the information to obtain an optimal strategy. A MDP is defined as a 5-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, d)$, adapted from [11]:

- Set of states $\mathcal{S}$.
- Set of actions $\mathcal{A}$.
- Transition distribution $\mathcal{T}(s'|s,a)$.
- Reward distribution $\mathcal{R}(r|s,a,s')$.
- Starting distribution $d_0(s)$.

Note that the transition distribution is sufficient to describe how the environment reacts to certain actions and how it changes. The incentive of the reward distribution is to provide some feedback, which represents how good it is to execute an action in a state. It can be seen as a definition of the goal. Furthermore the joint distribution of the next state and reward conditioned on the current state and action gets defined as: $p(s', r|s,a) = \mathcal{R}(r|s,a,s')\mathcal{T}(s'|s,a)$.

2.1.2 Reinforcement Learning Problem

The objective of an agent is to select a action in each state such that the overall reward described by the underlying MDP $\mathcal{M}$ is maximized. The definition is adapted from [11]. A policy $\pi(a|s)$ – modeled as a probability distribution over actions conditioned on the current state – is used to specify any behavior for running the task. Throughout this thesis only strategies, which are independent of the current time step, were studied. To ensure convergence in this setting a discount factor $\gamma \in [0,1[$ has to be selected. Assume from now on $a_t \sim \pi(a|s_t)$, s_{t+1}, $r_t \sim p(s', r|s_t, a_t)$ unless other stated. The basic RL problem can then be formalized as inferring π such that the discounted cumulative reward is maximized or more formally:

$$\pi^* = \arg\max_\pi \mathbb{E}_\pi^{\mathcal{M}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 \sim d_0(s) \right]$$

By "Bellman's Principle of Optimality" every π specifies two unique functions $\mathcal{V}_\pi$ and $\mathcal{Q}_\pi$, which can in turn be used to define π. Like in [11] this relation can be expressed as:

$$\mathcal{V}_\pi^{\mathcal{M}}(s) = \mathbb{E}_\pi^{\mathcal{M}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right] = \sum_a \pi(a|s) \mathcal{Q}_\pi^{\mathcal{M}}(s,a) \tag{2.1}$$

$$\mathcal{Q}_\pi^{\mathcal{M}}(s,a) = \mathbb{E}_\pi^{\mathcal{M}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s,\ a_0 = a \right] = \sum_{s',r} p(s', r|s,a) \left(r + \gamma \mathcal{V}_\pi^{\mathcal{M}}(s') \right) \tag{2.2}$$

As the solution is unique the optimal policy can be expressed in terms of them:

$$\pi^* = \arg\max_\pi \mathbb{E}_\pi^{\mathcal{M}} [\mathcal{V}_\pi(s_0) \mid s_0 = s,\ a_0 = a]$$

Note that with Equation 2.1 and Equation 2.2 either one of $\mathcal{V}_\pi$ or $\mathcal{Q}_\pi$ can be expressed in terms of the other. So an optimal policy can be equivalently expressed in terms of $\mathcal{Q}_\pi$.

2.1.3 Neural Action Network

A Neural Action Network is a neural network with a specialized structure dependent on the $\mathcal{M}$, which is well-suited for learning $\mathcal{Q}$-functions. It consists of an activation function f and a inner structure $\text{nl} \in \mathbb{N}^{\text{hl}}$. Furthermore let $\text{nl}(0) = |\mathcal{S}|$ and $\text{nl}(\text{hl}+1) = |\mathcal{A}|$. For $l \in \{1, \dots, \text{hl}+1\}$ weights $W_l \in \mathbb{R}^{\text{nl}(l-1),\text{nl}(l)}$ and biases $b_l \in \mathbb{R}^{\text{nl}(l)}$. The network can then be formally described with $l' \in \{1, \dots, \text{hl}\}$

$$o_0(s) = s \qquad o_{l'}(s) = a(o_{l'-1}(s)W_{l'} + b_{l'})$$

$$o_{\text{hl}+1} = o_{\text{hl}}(s)W_{\text{hl}+1} \qquad \mathcal{Q}(s,a) = o_{\text{hl}+1}(s)[a]$$

A general neural network is a mathematical model dependent on the weights and biases. The idea is to find a set of parameters minimizing the loss between the actual prediction and some predefined target. For more details on neural networks see e.g. [12]. Usually the targets are given in advance, but for Deep Reinforcement Learning (DRL) they get generated on the fly by the underlying $\mathcal{M}$.

2.2 Agents

2.2.1 Tabular $\mathcal{Q}$-Learning (TQL)

$\mathcal{Q}$-Learning was proposed by [13], a more detailed explanation is available in [11]. As the agent operates online with an environment, only one experience (s_t, a_t, r_t, s_{t+1}) can be observed per time step. There is no way for the agent to access the underlying MDP directly. Using the two previously mentioned points 2.2 reduces to:

$$\mathcal{Q}_\pi(s_t, a_t) = r_t + \gamma \max_{a'} \mathcal{Q}_\pi(s_{t+1}, a') \tag{2.3}$$

By transforming Equation 2.3 to an update rule and thereby storing the $\mathcal{Q}$-values in a table $\mathcal{Q}$-Learning can be derived. The formula gets rearranged and a learning rate $\alpha \in [0,1]$ is added. It's concrete value decides how much the $\mathcal{Q}$-value should be moved to the estimated target value. Note for deterministic environments set $\alpha = 1$, as there is no need to slowly approach the target value. However for stochastic environments this is necessary as the mean of a distribution needs to be approximated.

$$\begin{aligned}
\mathcal{Q}_{t+1}(s_t, a_t) &\leftarrow (1-\alpha)\mathcal{Q}_t(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} \mathcal{Q}_t(s_{t+1}, a')) \\
&= \mathcal{Q}_t(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} \mathcal{Q}_t(s_{t+1}, a') - \mathcal{Q}_t(s_t, a_t) \right) \\
&= \mathcal{Q}_t(s_t, a_t) + \alpha \, \text{TD}(s_t, a_t, r_t, s_{t+1})
\end{aligned}$$

All of the agent's knowledge is represented as a table over state and actions – called $\mathcal{Q}$-function. Each experience gets integrated into the $\mathcal{Q}$-function by using the above formula. Intuitively the update rule can be interpreted as a controller driving the value up when positioned underneath the target and vice-versa. An description of the complete procedure is given in algorithm 1.

2.2.2 Deep-$\mathcal{Q}$-Networks (DQN)

Deep-Q-Networks (DQN) were proposed by Mnih et. al., 2015 [8]. The authors utilized two tricks to stabilize the learning process. Each observation $e_t = (s_t, a_t, r_t, s_{t+1})$ gets inserted into the Experience Replay Memory (ERM) $D = (e_1, \dots, e_m)$. As succeeding experiences are highly correlated, a batch $B \subset D$ is sampled from the ERM. Each batch contains pseudo-decorrelated data, which reduces overfitting while training. Also they used a learning and a target network specified by $\mathcal{Q}(s, a|\theta_t)$ and $\mathcal{Q}(s, a|\theta_t^-)$ respectively. They optimized only $\mathcal{Q}(s, a|\theta_t)$ bootstrapping it against the values of $\mathcal{Q}(s, a|\theta_t^-)$, further improving the stability. After a fixed number of steps the weights were copied $\theta^- \leftarrow \theta$. Using a squared error loss for training on a sampled batch effectively optimizes the loss:

Algorithm 1: $\mathcal{Q}$-Learning($\mathcal{M}$, α, γ, π, epochs, steps)

initialize $\mathcal{Q}$ randomly
initialize reward[epochs] $\leftarrow 0$
for $e = 0;\ e < epochs;\ e \leftarrow e + 1$ **do**
  sample $s_0 \sim d_0(s)$
  for $t = 0;\ t < steps;\ t \leftarrow t + 1$ **do**
    sample $a_t \sim \pi(a|s_t)$
    observe $r_t, s_{t+1} \sim \mathcal{M}$
    update $\mathcal{Q}(s_t, a_t) \leftarrow \mathcal{Q}(s_t, a_t) + \alpha\ (r_t + \gamma\ \max_{a'} \mathcal{Q}(s_{t+1}, a') - \mathcal{Q}(s_t, a_t))$
    store res[e] $\leftarrow$ res[e] $+ r_t$
return $\mathcal{Q}$, *res*

$$\mathbb{L} = \mathbb{E}_{e_t \sim \mathbb{U}(D)}\left[\left(r_t + \gamma\ \max_{a'} \mathcal{Q}(s_{t+1}, a'|\theta_t^-) - \mathcal{Q}(s_t, a_t|\theta_t)\right)^2\right]$$

Their network architecture received the state as an input and outputs the $\mathcal{Q}$-values for each action, such that all $\mathcal{Q}$-values could be computed in one forward pass. As $\mathcal{Q}$-Learning suffers from overestimation (Van Hasselt, Guez & Silver, 2016) [14] proposed a modified loss with little amount of extra computational cost, which adds only one forward pass to the training:

$$\mathbb{L} = \mathbb{E}_{e_t \sim \mathbb{U}(D)}\left[\left(r_t + \gamma\ \mathcal{Q}(s_{t+1}, \arg\max_{a'} \mathcal{Q}(s_{t+1}, a'|\theta_t)|\theta_t^-) - \mathcal{Q}(s_t, a_t|\theta_t)\right)^2\right]$$

The stability can be further increased by using Huber-Loss instead of MSE [8]. Moreover all evaluations presented here have used the Huber-Loss. This effectively clips the norm of the gradients to 1, which results in a more stable learning process with outliers. The implementation uses LReLu instead of ReLu activation combined with the initialization scheme described by [15].

3 Exploration

This section presents an overview of Exploration in RL. Some of the techniques are pretty old, e.g. Boltzmann or ϵ-Greedy policies. Nevertheless there are also more recent methods like [9] and [2] and others. In the following text related work was reviewed, methodology summarized and compared to each other. Afterwards the methods were evaluated on the introduced tasks and integrated into a DDQN.

3.1 Difficulties

One of the main challenges in Reinforcement Learning agents is to manage exploration even if no valuable information is received. For example the agent has some observation of the environment, but didn't reached the goal yet. Only from the observation he can't tell if the goal is found. In Montezuma's Revenge one has to collect keys to open doors. However if the agent never experiences how to achieve points, he also cannot learn from it. While this is already problematic in games, this becomes even worse when moving to reality. Osband et. al. [2] calls the skill needed Deep Exploration. Nevertheless the agent has to decide when to explore and when to exploit the knowledge. A basic idea would be to just decrease the exploration over time. However when facing real world tasks it very likely that during a run something in the environment changes, which then needs to be re-explored.

Data Efficiency is another important aspect, especially DQN's suffer from that problem. There have been some approaches, e.g. Priotirized Replay Memory [16] or Hindsight Experience Replay [17]. The former prioritizes experiences by their TD-errors and the latter also gets insight using bad examples. Furthermore there is the need for other techniques, which maybe utilize some knowledge about the model to decide how to increase the performance or de-correlate the data, so that complex models do not tend to overfit. In [3] a interesting idea is presented for improving data efficiency. Nevertheless all of these methods need to be scalable, e.g. so they can be used in real world tasks.

3.2 Deterministic Exploitation

These policies select their actions such that the action value is deterministically. If there is no learning of new knowledge the agent always selects the same action when faced with a state s. The chosen actions are based on the current Q-Function, whereas the policy has to assign the probability 1 to one action for each state, in order to be deterministic. In fact it showed to beneficial if the agent employs a deterministic exploration behavior, because the taken actions are consistent. Several recently proposed techniques like [2] and [9] rely on an underlying Greedy-Policy.

3.2.1 Greedy-Policy

Greedy exploration always takes the best action known to the agent. Using the $\mathcal{Q}$ function the distribution can be expressed as:

$$\mathcal{B}^s_{\mathcal{Q}} = \{a | a = \arg\max_{a^*} \mathcal{Q}(s, a^*)\} \qquad \pi_{\text{greedy}}(a|s, \mathcal{Q}) = \frac{\mathbb{I}\left[a \in \mathcal{B}^s_{\mathcal{Q}}\right]}{|\mathcal{B}^s_{\mathcal{Q}}|}$$

3.3 Exploration by Stochasticity

Stochastic Policies on the other hand are able to select different actions when the agents visits a state s twice - even when there is no newly learned knowledge. All of them have in common that they sample from a non-deterministic probability distribution. The probability values assigned to each action don't have to depend on the values of the Q-Function. Nevertheless it is a good idea to incorporate knowledge into the selection, as this guides the exploration in more interesting parts of the graph.

3.3.1 Random-Policy

Random exploration is a naive strategy and selects always a uniformly sampled action from the distribution:

$$\pi_{\text{rand}}(a|s) = \frac{1}{|\mathcal{A}(s)|}$$

3.3.2 ϵ-Greedy -Policy

This is a rather simplistic exploration strategy. The main idea is that the agent should execute a random action with probability ϵ, whereas for the other $1-\epsilon$ of the cases greedily with respect to the $\mathcal{Q}$ function. With $\epsilon \in]0,1[$ this can be formalized as sampling from:

$$\pi_\epsilon(a|s,\mathcal{Q},\epsilon) = (1-\epsilon)\ \pi_{\text{greedy}}(a|s,\mathcal{Q}) + \epsilon\ \pi_{\text{rand}}(a|s)$$

It can be seen as sampled from a non-continuous function space, bounded by $\pi_{\text{greedy}}(a|s,\mathcal{Q})$ and $\pi_{\text{rand}}(a|s)$.

$$\lim_{\epsilon'\to 0}\left[\pi_\epsilon(a|s,\mathcal{Q},\epsilon')\right] = \pi_{\text{greedy}}(a|s,\mathcal{Q}) \qquad \lim_{\epsilon'\to 1}\left[\pi_\epsilon(a|s,\mathcal{Q},\epsilon')\right] = \pi_{\text{rand}}(a|s)$$

The corresponding continuous version sharing this property can be found in 3.3.3.

3.3.3 Boltzmann-Policy

Boltzmann depends more on the actual value of the $\mathcal{Q}$ function. It uses the value to assign different probabilities to each action using a softmax distribution. There is a temperature β involved which controls this dependence. As in the case of ϵ-Greedy -Policy the action gets sampled from a distribution:

$$\upsilon_\beta(x) = \exp\left(\frac{x}{\beta}\right)$$

$$\pi_{\text{boltz}}(a|s,\mathcal{Q},\beta) = \frac{\upsilon_\beta(\mathcal{Q}(s,a))}{\sum_{a'\in\mathcal{A}(s)} \upsilon_\beta\left(\mathcal{Q}(s,a')\right)}$$

The Amount of Exploration by this policy gets controlled by a temperature $\beta \in]0,\infty[$. Intuitively every Boltzmann exists in a function space bounded by the greedy and random strategy. As the following analysis show as β goes to it's limits, that for each one of them the strategy converges to these policies. For all $\mathcal{Q}$, s and a we derive the limits for $\beta \to g \in \{0,\infty\}$:

$$\text{le}_g^x := \lim_{\beta'\to g}\left[\upsilon_{\beta'}(x)\right] = \exp\left(\lim_{\beta'\to g}\left[\frac{x}{\beta'}\right]\right) \to \begin{cases} \mathbb{I}[x=0] + \text{sgn}(x)\ \infty & \text{if} \quad g = 0 \\ 1 & \text{if} \quad g = \infty \end{cases} \tag{3.1}$$

With that the policy limits with respect to β can be derived:

$$\lim_{\beta' \to 0} \left[\pi_{\text{boltz}}(a|s, \mathcal{Q}, \beta') \right] = \frac{\text{le}_0^{\mathcal{Q}(s,a)}}{\sum_{a' \in \mathcal{A}(s)} \text{le}_0^{\mathcal{Q}(s,a')}} \overset{3.1}{\to} \frac{\text{le}_0^{\mathcal{Q}(s,a)}}{\sum_{a' \in \mathcal{B}^s_{\mathcal{Q}}} \text{le}_0^{\mathcal{Q}(s,a')}} = \frac{\text{le}_0^{\mathcal{Q}(s,a)} \, \text{le}_0^{\mathcal{V}(s)^{-1}}}{|\mathcal{B}^s_{\mathcal{Q}}|}$$

$$= \frac{\text{le}_0^{\mathcal{V}(s)} \, \text{le}_0^{\mathcal{V}(s)^{-1}} = 1}{\left(1 + \mathbb{I}\left[a \notin \mathcal{B}^s_{\mathcal{Q}}\right] \left(\text{le}_0^{\mathcal{V}(s) - Q(s,a)} - 1\right)\right) |\mathcal{B}^s_{\mathcal{Q}}|} \overset{3.1}{\to} \frac{\mathbb{I}\left[a \in \mathcal{B}^s_{\mathcal{Q}}\right]}{|\mathcal{B}^s_{\mathcal{Q}}|} = \pi_{\text{greedy}}(a|s, \mathcal{Q})$$

$$\lim_{\beta' \to \infty} \left[\pi_{\text{boltz}}(a|s, \mathcal{Q}, \beta') \right] = \frac{1}{|\mathcal{A}(s)|} = \pi_{\text{rand}}(a|s)$$

It was shown that Boltzmann converges to either a random or a greedy policy in the limits of β.

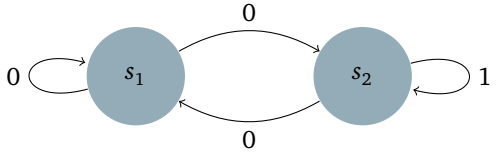
3.4 Exploration by Optimality

The core idea is to generally think of all possibilities optimal, reducing this view when experienced with real rewards. One common idea would be to initialize the $\mathcal{Q}$-function to values such that it is biased to find a solution for the underlying $\mathcal{M}$ rapidly. One of these techniques is presented in the next subsection.

3.4.1 Optimistic Initialization (OI)

The idea is to initialize the $\mathcal{Q}$-values to very high values, which directs the agent into regions were it wasn't that often. In the experimental section the values are initialized to $\mathcal{Q}_0 \sim \mathbb{U}(\max_{a,s} \mathcal{Q}^*(s,a) + l, \max_{a,s} \mathcal{Q}^*(s,a) + l + w)$. This method is easily applicable to TQL (subsection 2.2.1). In fact [18] showed that if for all s, a the assumption $\mathcal{Q}_0(s,a) \geq \max_{s',a'} \mathcal{Q}^*(s',a')$ holds convergence for TQL can be guaranteed, whereas sampling extremely high $\mathcal{Q}_0$ values results in a delayed learning process. Working values are somehow restricted to a bounded interval. Additionally they are highly task dependent and if determined limit the transferability to other tasks. In real world task it is very unlikely to know an approximation of the optimal $\mathcal{Q}^*$-function prior to solving the task. To mitigate for this Machado et. al [19] proposed a domain-independent technique based on normalizing the rewards by the first seen non-zero reward, which approximately shifts the mean of the $\mathcal{Q}$-function to zero. Nevertheless it is possible to construct a problem where this technique fails, e.g. to design it such that the first seen reward is very small compared to the rest.

As TQL can't be scaled to arbitrarily sized tasks, function approximations needs to be integrated. In the linear case it is possible to derive an initialization in parameter space which is also optimistic in $\mathcal{Q}$-space, but for deep neural architectures there is no straightforward way to find a proper parameter vector without hindering the performance or solving the task. However [19] can be applied under these constraints, because it alters only the rewards.



(a) $\mathcal{M}_1$ returns a reward when staying in s_2

$\mathcal{Q}$	L	R
s_1	$\mathcal{Q}(s_1, R) - 1 = 98$	$\mathcal{Q}(s_2, R) - 1 = 99$
s_2	$\mathcal{Q}(s_1, R) - 1 = 98$	$\frac{1}{1-\gamma} = 100$

(b) Optimal $\mathcal{Q}$-function for MDP $\mathcal{M}_1$

Figure 3.1: Simple MDP with Optimal $\mathcal{Q}$-values.

(a) Simple MDP is used to give an intuition on the behavior of optimist initialization. (b) Shows the optimal $\mathcal{Q}$-function. It can be seen that going right in 1 has the highest value. From that all other values which are dependent

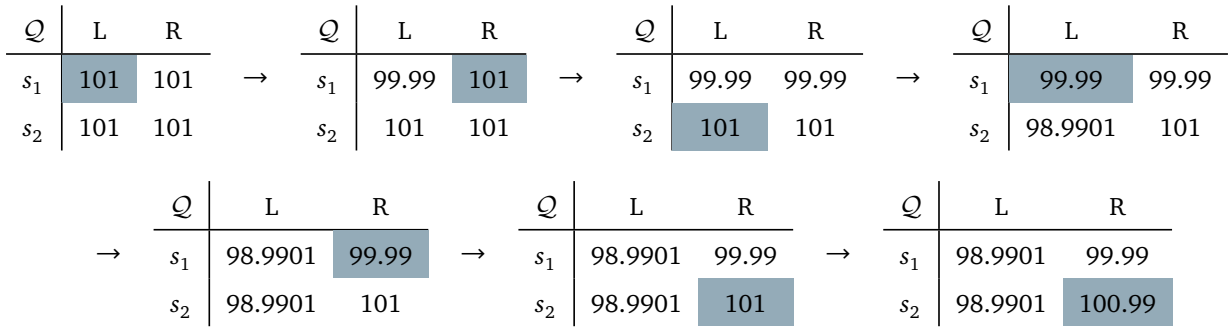


Figure 3.2: Exploration Scheme explores All State-Action Pairs.

To see intuitively why optimistic initialization works consider a simple MDP $\mathcal{M}_1$ from Figure 3.1a. The optimal $\mathcal{Q}$-Value Function can be calculated by hand for $\gamma = 0.99$, see Figure 3.1b. If the agent gets optimistically initialized, e.g. all values with 101. Assume L gets prioritized over R when they have equal $\mathcal{Q}$ values for a state. As long all values are optimistic the agent tries the best action reduces it's value accordingly until a different action appears better. In doing so all state-action pairs are executed at least once.

3.5 Exploration by Intrinsic Motivation

These techniques heavily rely on a process called reward shaping. Humans also receive some form of intrinsic motivation, which subconsciously guides the agent into other regions. Such intrinsic rewards can be based, e.g. on novelty, empowerment or surprise, usually they are added to the present external reward using the framework proposed by Chentanez, Barto & Singh [20].

3.5.1 Reward Shaping vs. Action Selection

When receiving a reward bonus it can be used to either learn a slightly modified $\mathcal{Q}$-function or to use it only for action selection. If a reward bonus is meant it can be used in both ways. The difference lies in the time to forget a bonus, whereas the first one actively learns the modified target resulting in a delayed learning process, the second only influences the current action selection, and is thus not integrated into the knowledge and remembered by the agent. From now a bonus is denoted as $r^+ = r^+(s,a)$. For reward shaping this results in $r^*(s,a) = r(s,a) + r^+(s,a)$, whereas for action selection the bonus is utilized like $a_t = \arg\max_a Q(s_t,a) + r^+(s_t,a)$

3.5.2 Upper-Confidence-Bound (UCB) Exploration

The most prominent example of Intrinsic Motivation is the Upper-Confidence-Bound Exploration. This strategy relies on a state-action count. The core idea is to base the action selection on the upper bound of an estimated interval of plausible $\mathcal{Q}$-values. Instead of remembering the mean and the variance for each value, this is practically handled by reward shaping. This is based on a count, how often a certain state-action-pair was already taken. For small scale problems this can be realized by a table. In order to scale it to arbitrary big tasks the techniques using density estimation described in 3.5.3 can be used.

Assume for now that $N_t(s,a)$ - count of state-action pair - is accessible at each time-step t. UCB is integrated by giving the reward bonus assuming $t = \sum_{s,a} N_t(s,a)$:

$$r^+ = \sqrt{\frac{p\log(t)}{N_t(s,a)}}$$

For action selection all available actions are taken into account. When all of them were executed the same number of times, the reward bonus has also same value and a constant doesn't influences the action selection.

3.5.3 Pseudo-Count Exploration

In [9] novelty is integrated into the RL-framework by using a so called Pseudo-Count (PC) $\hat{N}_t(s,a)$. The PC is proportional to the real count, e.g. $\hat{N}_t(s,a) \propto N_t(s,a)$. This concept can also be used in subsection 3.5.2. The formulas from[9]

get reviewed in this section. Therefore assume that a density model $p_t(s,a)$ is accessible. Furthermore let $p'_t(s,a) = p_t(s,a|s_t,a_t)$ be the recoding probability, when the current state-action pair was seen again. Using that the PC can be obtained by solving the linear equation system:

$$p_t(s,a) = \frac{\hat{N}_t(s,a)}{t} \qquad p'_t(s,a) = \frac{\hat{N}_t(s,a)+1}{t+1}$$

All over the paper they assume that p is learning-positive. For a definition see [9]. The interpretation of these rules is that the model is consistent when the PC is reasonable as well, e.g it can't be negative or a division by null has to be avoided. Once solved results the above system in:

$$\hat{N}_t(s,a) = \frac{p_t(s,a)(1-p'_t(s,a))}{p'_t(s,a)-p_t(s,a)}$$

The count-based exploration bonus from the paper can then be written as:

$$r^+ = \frac{\beta}{\sqrt{\hat{N}_t(s,a)}}$$

The type of density model depends on the problem. For small scale problems an empiric count-based model can be used. Pixel CNN [21] or PixelRNN [22] is suitable for image-based state data. Different discrete binary-based auto-regressive methods like NADE [23] it's extension DeepNADE [24] or MADE [25] are possible, but they suffer from a lack of scalability as the number of input neurons is logarithmic to the number of different states. A continuous version is available, e.g. in RNADE [26]

3.5.4 InfoGain Exploration [1]

This is based on subsection 3.6.1. The idea is to give a reward bonus for uncertain state-action pairs. This uses a $\mathcal{Q}$-Ensemble $\mathcal{Q}_t^k(s,a)$ for $k \in \{1,\dots,K\}$. Using a temperature value β the following distribution is controlled and defined like in [1]:

$$h_{t,\beta}^k(a|s) = \frac{\nu_\beta\left(\mathcal{Q}_t^k(s,a)\right)}{\sum_{a'\in\mathcal{A}(s)} \nu_\beta\left(\mathcal{Q}_t^k(s,a')\right)}$$

For calculating the quantity simply average over the distributions $h_{t,\beta}^k(a|s)$. The so used average softmax distribution is then used to calculate the KL-divergence from each head's distribution to the average.

$$h_{t,\beta}^{\text{avg}}(a|s) = \frac{1}{K}\sum_{k=1}^{K} h_{t,\beta}^k(a|s)$$

$$b_\beta(s) = \frac{1}{K}\sum_{k=1}^{K} \mathcal{KL}\left[h_{t,\beta}^k \| h_{t,\beta}^{\text{avg}}\right]$$

$$r^+ = \rho\, b_\beta(s)$$

In the original paper it gets used with the technique from subsection 3.6.2 together. Additionally the particular choice of ρ also depends on the problem structure. One disadvantage is that it is s

3.6 Exploration by Posterior Sampling

For large scale problems it is intractable to sample from the posterior. However recent advances proposed some very effective techniques to approximate this posterior sampling. As this uncertainty naturally drives exploration, it is a promising way to go. In the following useful techniques are explained roughly. The main principle of Posterior Sampling is given by:

$$p(\mathcal{Q}|\mathcal{D}) = \int_{\mathcal{M}} p(\mathcal{Q}^*|\mathcal{M}, s, a, \mathcal{D}) p(\mathcal{M}|\mathcal{D}) d\mathcal{M}$$

However this is only tractable for small state spaces as a distribution over all possible MDP's needs to be maintained. Techniques exist for Multi-Arm Bandits, therefore see also the tutorial on Thompson Sampling by Russo, Roy, Kazerouni et. al. [27]. The approaches from the next sections very roughly approximate sampling from $p(\mathcal{Q}|\mathcal{D})$ or the value of $\mathbb{E}_{\mathcal{Q}\sim p}[\mathcal{Q}]$ by maintaining a set of $\mathcal{Q}$ functions.

3.6.1 Bootstrapped DQN (BDQN) [2]

This technique depends on two parameters. The first is K which specify the number of independent copies to use, and secondly $H \leq K$ which says for how much heads one single sample should be used. Both of them control the level of exploration, but as K introduces some parameters or at least additional forward runs, the parameter can't be chosen arbitrary high. H can be interpreted to hold the variance between two heads higher, however experiments in [2] and this thesis show that $H = K$ works pretty well. However they might be some other problems, where it is better to introduce a smaller H.

This technique approximates sampling from $p(\mathcal{Q}|\mathcal{D})$ through an ensemble of models. It is first explained for the naive tabular case like also introduced in [3]. Therefore simply K different $\mathcal{Q}$-functions are sampled at the beginning, for each $k \in \{1, \dots, K\}$ initialize a random initial $\mathcal{Q}$-function.

$$\mathcal{Q}_0^k \sim \mathbb{U}(\max_{a,s} \mathcal{Q}^*(s,a) + l, \max_{a,s} \mathcal{Q}^*(s,a) + l + w)$$

One head $c \sim \mathbb{U}(\{1, \dots, K\})$ is sampled prior to each episode. The agent then follows the induced policy greedily for one episode collecting experiences from the environment. In each step the current experience is taken to bootstrap H random models with this example. Not that each head is always bootstrapped against itself, to ensure the consistency to it's target values. Over time all policies should converge to one policy, when this happens the agent stops exploring. On each epoch the optimized objective can be expressed with $z_t \in Z_t \sim \mathbb{U}(\{Z' | Z' \subseteq \{1, \dots, K\} \wedge |Z'| = H\})$ as:

$$\mathbb{L} = \mathbb{E}_{e_t \sim \mathbb{U}(D); z \in Z_t} \left[\left(r_t + \gamma \max_{a'} \mathcal{Q}_t^z(s_{t+1}, a') - \mathcal{Q}_t^z(s_t, a_t) \right)^2 \right] \tag{3.2}$$

In the paper they used a shared neural network connected to K independent heads. In the paper [2] they claim that initializing the weights randomly suffices to maintain the variance in exploration. Additionally they use Double-Deep-Q-Learning described in subsection 2.2.2 to account for the overestimation bias.

3.6.2 UCB Ensemble Exploration (UCBE) [1]

This technique extends the approach described in the previous section. It integrates UCB Exploration with the Bootstrapped DQN. It does so by taking the mean and variance over the subset of head instead of relying on one randomly sampled head like in subsection 3.6.1. The formulas were adopted from the paper [1]:

$$a_t \in \arg\max \{\hat{\mu}(s_t, a) + \lambda \hat{\sigma}(s_t, a)\}$$

$$\hat{\mu}_{\mathcal{Q}_t}(s, a) = \frac{1}{K} \sum_{k=1}^{K} Q_t^k(s, a)$$

$$\hat{\sigma}_{\mathcal{Q}_t}(s, a) = \frac{1}{K} \sum_{k=1}^{K} \left(Q_t^k(s, a) - \hat{\mu}_{\mathcal{Q}_t}(s, a)\right)^2$$

The connection to normal UCB Exploration is not directly clear from subsection 3.5.2. Instead of approximating the variance by using multiple copies, the reward is shaped and reduced over time, which indirectly causes the learned $\mathcal{Q}$-value to be located around the upper confidence bound.

3.6.3 Shared Learning using $\mathcal{Q}$-Ensembles [3]

Last but not least we describe an approach which uses transfer learning between the heads. The idea from Menon & Ravindran [3] consists in using the best head for making the action selection for the bootstrap. Nevertheless all heads are bootstrapped against themselves to ensure convergence. In this thesis the technique is naively adopted to the tabular case. After a specific number of steps, e.g. 500 the agent determines which head performs best by finding the maximum over all heads for the current state s_t and action a_t. However as this doesn't take the history into account, there might be some room for improvement.

$$k^* = \arg \max_{k \in \{1, \ldots, K\}} Q_t^k(s_t, a_t)$$

With the previous approximated best head the loss can be rewritten as.

$$\mathbb{L} = \mathbb{E}_{e_t \sim \mathbb{U}(D);z \in Z_t} \left[\left(r_t + \gamma \, \mathcal{Q}_t^z(s_{t+1}, \arg\max_{a'} \mathcal{Q}_t^{k*}(s_{t+1}, a')) - \mathcal{Q}_t^z(s_t, a_t) \right)^2 \right]$$

Note that the action selection is not based on the head itself anymore like in DDQN, but instead on the best known. The main incentive of the authors was to transfer the best available knowledge to the other heads. It is a reasonable approach, but a different criterion for selecting the best head might yield better results. For example it could be based on the history by using a right aligned moving average.

4 Proposed Algorithms

4.1 Tabular Bootstrapped Pseudo-Count

In the previous section two methods were presented. First of all there was the intrinsically motivated Pseudo-Count Exploration (PCE) and second the Bootstrapped DDQN (B-DDQN). Throughout this paper we use the naive tabular version for evaluating Bootstrapped DDQN. Both approaches have their strengths and weaknesses. PCE is capable of remembering which state-action pairs weren't investigated that often and B-DDQN is able to represent uncertainty in a computationally efficient manner. For some tasks B-DDQN has problems to explore everything - at least in the tabular case - it then converges to a suboptimal solution. The idea behind B-DDQN is to try to guess a good approximation of the distribution of $\mathcal{Q}$-functions by using multiple initializations. This approximation gets worse when all $\mathcal{Q}$-functions are converging to the same value. For standard $\mathcal{Q}$-Learning this happens even faster. See subsubsection 6.1.2.4 for an evaluation on how many heads should use one sample for learning. It is argued that it is sometimes important to keep the variance in the models. One way of achieving this is to set $H < K$, whereas in the original paper [2] was stated that random initialization already suffices.
Another approach would be to attach a density model to the Bootstrapped, or equivalently B-DDQN. This should guide the agent in not so well known directions, whereas the different heads ensure that there is some variance inbetween them. Obviously when $H = K$ it makes only sense to attach one density model to the agent, as all heads experience the same information. In algorithm 2 the proposed combination can be seen. See subsubsection 6.1.2.1 for an evaluation on small MDP's.

Algorithm 2: Tabular Bootstrapped Pseudo-Count ($\mathcal{M}$, α, γ, π, H, K, β, epochs, steps)

let $i \in \{1, \dots, K\}$
set reward[epochs] $= 0$, $t = 0$
initialize $\mathcal{Q}_i(s,a)$ randomly
initialize $N_i(s,a) = 0$

for $e = 0$; $e < epochs$; $e \leftarrow e + 1$ **do**
  sample $s_0 \sim d_0(s)$
  sample $k \sim \mathbb{U}(\{1, \dots, K\})$
  set done $= 0$
  for $t' = 0$; $[t' < steps \wedge \neg done]$; $t', t \overset{+}{\leftarrow} 1$ **do**
    get $a_t \in \arg\max_{a'} \mathcal{Q}_t^k(s,a)$
    observe $r_t, s_{t+1} \sim \mathcal{M}$
    sample $h_1, \dots, h_H \sim \mathbb{U}(\{1, \dots, K\})$
    for $j = 0$; $j < H$; $j \leftarrow j \overset{+}{\leftarrow} 1$ **do**
      set $N_j(s_t, a_t) \leftarrow N_j(s_t, a_t) + 1$
      set $r^* = r_t + \beta / \sqrt{N_i(s_t, a_t)}$
      update $\mathcal{Q}_j(s_t, a_t) \leftarrow \mathcal{Q}_j(s_t, a_t) + \alpha \left(r^* + \gamma \max_{a'} \mathcal{Q}_j(s_{t+1}, a') - \mathcal{Q}_j(s_t, a_t) \right)$
    store res[e] $\leftarrow$ res[e] $+ r_t$
return $\mathcal{Q}$, *res*

when $H < K$ is set, a different density model for each head is needed, while this is easily feasible for the tabular case as the parameters only double, it remains challenging for large state spaces. Using separate neural density models for each head would increase the computational costs drastically. Another idea would be to condition the density model on the number of the head, using only one model, but this would connect the densities to each other. Inspired by Bootstrapped

DQN [2] one could also use a shared network with independent heads, where the intuition is that some information of densities can be reused by the shared model as the heads are exploring one region. Nevertheless currently no techniques aiming for that problem was proposed.

4.2 Regularized Bootstrapped DDQN Exploration

This section is only relevant to Bootstrapped DDQN and it's extensions. The methods were evaluated on the MountainCar-v0 task. Instead of using a set of neural networks, only subnetworks from a shared one are used. Finally it gets combined with subsection 3.5.4 and subsection 3.6.2.

4.2.1 Regularization

In supervised learning regularization techniques are used to reduce the magnitude of all weights. As the model is kept simple it is inclined to learn a more general representation. A binary mask is used in the implementation to sample a model. Let $m_l^i \in \mathbb{B}^{\mathrm{nl}(l)}$ be a mask for $i \in \{1,\dots,K\}$. By determining the expectation over masks sampled from a Bernoulli with probability φ, the expectation over the posterior can be approximated.

$$\mathbb{E}_{\mathcal{Q}\sim p_t(\mathcal{Q}|\mathcal{D})}[\mathcal{Q}] \approx \mathbb{E}_{m_l\sim \mathrm{Bern}(\mathrm{nl}(l),\varphi)}[\mathcal{Q}(s,a|\theta_t,m)] \approx \frac{1}{K}\sum_{i=1}^{K}\big[\mathcal{Q}(s,a|\theta_t,m^i)\big]$$

For this chapter assume all definitions from subsection 2.1.3 also depend on the head $i \in \{1,\dots,K\}$. All submodels share the same weights, but their output is influenced by the associated mask.

$$o_0^i(s) = s \qquad o_{\mathrm{hl}+1}^i = o_{\mathrm{hl}}^i(s)^T W_{\mathrm{hl}+1} \qquad \mathcal{Q}(s,a|m^i) = o_{\mathrm{hl}+1}^i(s)[a]$$

The recursive rule is given when the techniques are presented. Note that the mask size represents an upper bound, because not all techniques utilize every mask entry, e.g. Zoneout. As each element is sampled independently simply not using some weights doesn't make a difference.

In the end all members of the ensemble should optimally converge to the same solution. This goal can only be achieved when the weights are small and represent the $\mathcal{Q}$-function appropriately. When initialized randomly at the beginning, there are weights which influence the output more than others. If a subnetwork is sampled, this obviously removes or strengthens some characteristics of the policy. During the first episodes this results in a lot of variance and thus a lot of exploration. Moreover as the game progresses the difference between the outputs by two masks gets smaller, which results in less exploratory behavior at latter steps. Additionally the agent tends to generalize better, because overfitting is reduced. However due to the reduction in model representability it could also hinder the learning process.

4.2.1.1 Dropout [4]

Dropout was proposed by Srivastava, Hinto, & Krizhevsky et. al [4]. It can be integrated into neural network in an computationally efficient manner. The recursive formula describing this approach is:

$$o_{l'}^i(s) = m_{l'}^i \; a(o_{l-1}^i(s)^T W_{l'} + b_{l'})$$

The mask effectively decides for each layer which neurons are participating and which are currently disabled - set to zero. In [5] it is said, that setting an output to zero leads to sudden changes in the distribution, which makes it hard for recurrent neural networks to converge. Although in this thesis a feed forward network is used, it is generally a good idea for increasing stability to slightly modify a control policy.

Algorithm 3: Regularized-UCB DDQN (R-UCB-DDQN)

let $i \in \{1,\dots,K\}$, $h \in \{1,\dots,H\}$
sample weights θ_0, θ_0^- for neural network randomly
initialize replay memory $\mathcal{D}$ with size N
set reward[epochs] $= 0$, $t = 0$

for $e = 0$; $e < epochs$; $e \leftarrow e + 1$ **do**

  sample $s_0 \sim d_0(s)$
  sample $m_l^i \sim \text{Bern}(\text{nl}(l), \varphi)$ for $i \in \{1,\dots,K\}$
  set done $= 0$

  for $t' = 0$; $[t' < steps \wedge \neg done]$; $t', t \overset{+}{\leftarrow} 1$ **do**

    calculate $\hat{\mu}_{\mathcal{Q}_t} = (\sum_{k'=1}^{K} \mathcal{Q}(s_t, a_t|\theta_t, m^{k'}))/K$
    calculate $\hat{\sigma}_{\mathcal{Q}_t} = (\sum_{k'=1}^{K} (\mathcal{Q}(s_t, a_t|\theta_t, m^{k'}) - \hat{\mu}_{\mathcal{Q}_t})^2)/K$
    get $a_t \in \arg\max_{a'} \hat{\mu}_{\mathcal{Q}_t} + \rho\hat{\sigma}_{\mathcal{Q}_t}$

    observe $r_t, s_{t+1}, \text{done} \sim \mathcal{M}$
    store $(s_t, a_t, r_t, s_{t+1}, \text{done})$ in $\mathcal{D}$
    sample minibatch $(s_j, a_j, r_j, s_{j+1}, d_j)$ from $\mathcal{D}$

    sample $\hat{m}_l^h \sim \text{Bern}(\text{nl}(l), \varphi)$ for $h \in \{1,\dots,H\}$

    set $y_j^h = \begin{cases} r_j & \text{if} \quad d_j = 1 \\ r_j + \gamma\,\mathcal{Q}(s_{t+1}, \arg\max\limits_{a'} \mathcal{Q}(s_{t+1}, a'|\theta_t, \hat{m}^h)|\theta_t^-, \hat{m}^h) & \text{else} \quad d_j = 0 \end{cases}$

    perform gradient descent on $\sum_{h',j} \left(y_j^{h'} - \mathcal{Q}(s_t, a_t|\theta_t, \hat{m}^{h'})\right)^2$
    store res$[e] \leftarrow$ res$[e] + r_t$

return $\mathcal{Q}$, *res*

4.2.1.2 Zoneout [5]

Zoneout was proposed by Krueger, Maharaj & Kramár et. al. [5]. The difference to Dropout is that it bypasses a neuron, instead of discarding the output value. As the authors said this can be seen as generalization of Dropout. Note that the following condition has to hold $\forall l_1, l_2 \colon \text{nl}(l_1) = \text{nl}(l_2)$ for applying this technique. To make models comparable this assumption is used for all three types of regularization. Formally this can be described as:

$$o_{l'}^i(s) = m_{l'}^i\, a(o_{l-1}^i(s)^T W_{l'} + b_{l'}) + (1 - m_{l'}^i)\, o_{l-1}(s)$$

A network using this method needs to account for these random structural changes. It might learn to produce the same distribution for neurons positioned behind each other. Once this robustness is achieved all submodels learned similar representations. However it might be that there is inherent to the problem or network no solution, such that the outputs of all ensemble members are more or less the same.

4.2.1.3 Shakeout [6]

It was proposed by Kang, Li, Tao et.al [6]. Instead of reconnecting or setting some connections to zero like the previous two methods it uses the mask, to apply the following change to the architecture:

$$o_{l'}^i(s) = a(o_{l-1}^i(s)^T (W_{l'}\, m_{l'}^i + q\ \text{sgn}(W_{l'})\, (m_{l'}^i - 1)) + b_{l'})$$

For each neuron, it is decided to either take the input values or use a constant with the same sign as the corresponding weights. When $q = 0$ the connection between Shakeout and Dropout is seen, although it doesn't reduce to Dropout. For $q > 0$ this method effectively rescales a weight w by $|w|/q$, so the network learns to be invariant to weight rescaling. If all weights fulfill $|w| = q$, all masks obviously produce the same output.

4.2.2 Regularized-UCB DDQN (R-UCB-DDQN)

This section combines subsection 3.6.1, subsection 3.6.2 with subsection 4.2.1. In the beginning of each episode a set of masks m_l^i with $i \in \{1, \dots, K\}$ is sampled, which serve as the heads already familiar from Bootstrapped DQN. The agent calculates all $\mathcal{Q}(s|m^i)$ using them to derive the average and variance among them. For action selection the upper confidence bound of this interval is taken, whereas the size can be modified with ρ like in subsection 3.6.2. After receiving the experience a different set of masks $\hat{m}_l^j$ with $j \in \{1, \dots, H\}$ is sampled, which gets for training on the sampled batch of the ERM. Like in all related methods each head is bootstrapped against its target network using the same mask. Formally the following objective is minimized:

$$\mathbb{L} = \mathbb{E}_{m_l \sim \text{Bern}(\text{nl}(l),\varphi)} \left[\left(r_t + \gamma \, \mathcal{Q}(s_{t+1}, \arg\max_{a'} \mathcal{Q}(s_{t+1}, a' | \theta_t^-, m) | \theta_t, m) - \mathcal{Q}(s_t, a_t | \theta_t, m) \right)^2 \right]$$

All parts were combined and presented as an algorithm in algorithm 3. The main difference consists in using an ensemble of masks to simulate an ensemble of networks. However due to really sampling new masks there are $2^{\mathbb{1}^T nl}$ unique ones. Nevertheless φ imposes a prior on the masks, so some of them receive a approximately zero probability.

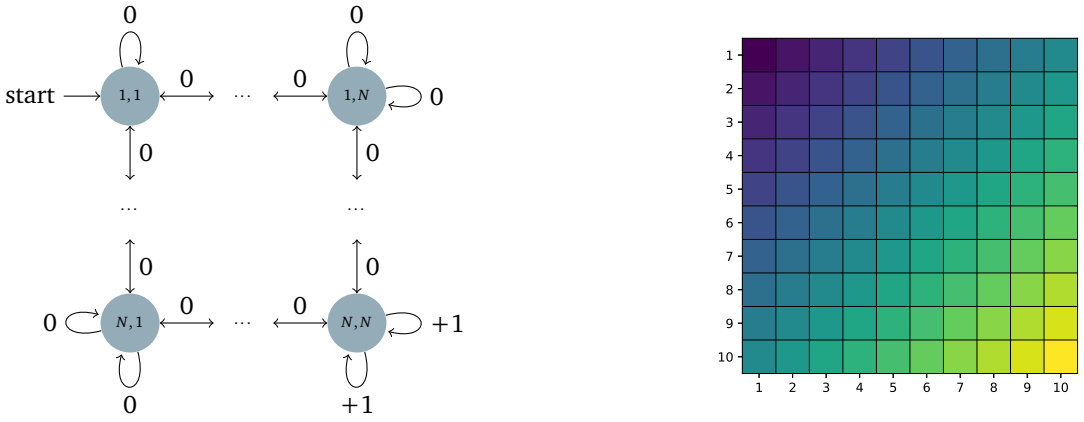
5 Tasks

5.1 Deterministic MDP's

Five different problems were examined and explained in detail. For each of them the problem structure was described shortly, by supplying a plot of the MDP. Furthermore a plot of the optimal $\mathcal{V}^*$-function was added, to help understanding the relationship between problem and $\mathcal{V}^*$-function. The episode length Ψ of each problem is given in terms of N. Another important characteristic is the size of the state $|\mathcal{S}|$ and action space $|\mathcal{A}|$, especially for TQL subsection 2.2.1 where the size of the table can be expressed as $|\mathcal{Q}| = |\mathcal{S}||\mathcal{A}|$. Both spaces were defined for each problem in terms of N.

5.2 Grid World

The first introduced problem is a basic Grid World $\mathcal{M}_G$ with $\mathcal{S} = \{1,\dots,N\}^2$ and $\mathcal{A} = \{l,u,r,d\}$. At the beginning of an episode the agent starts at $s_0 = (1,1)$ and is allowed to freely move in all four cardinal directions. Only executing r or d in (N,N) yields a reward of $+1$, whereas all other transitions output a non-informative zero. Every episode gets executed for $\Psi = 2N$ steps. A optimal performing strategy needs only $2(N-1)$ steps to reach (N,N), hence the maximum achievable reward values 2. A graph describing all details of the MDP is given in Figure 5.1a. The $\mathcal{Q}$-space sizes $|\mathcal{Q}| = N^2/4$.



(a) MDP of Grid World

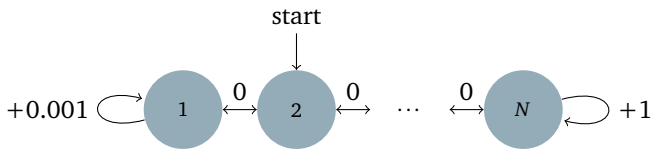
(b) Optimal $\mathcal{V}^*$-function (N=10)

Figure 5.1: MDP and Optimal $\mathcal{V}^*$-function for Grid World

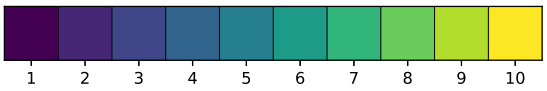
(a) MDP for Grid World for arbitrary N. In every state four actions left, top, right, and down are available. Each of them moves the agent to the next state in the respective direction. If he walks against a wall, the action is performed, but the state is not altered. For executing right or down in (N,N) a reward of $+1$ is granted, whereas for all remaining transitions no reward is given. (b) $\mathcal{V}^*$ gives a smooth gradient originating from $(0,0)$ to (N,N). Note that the diagonals have the same $\mathcal{V}$-value, because there is no advantage when starting from a same-valued field.

5.3 Exploration Chain [2]

The Exploration Chain $\mathcal{M}_E$ was proposed by I. Osband et. al. [2]. A MDP is of this task is presented in Figure 5.2a, it has the spaces $\mathcal{S} = \{1,\dots,N\}$ and $\mathcal{A} = \{l,r\}$. The initial starting state is 2. Each episode runs for about $\Psi = N+9$ steps. On the left end of the chain an attracting reward of $+0.001$ is given for going left in state 1. On the right side the highest reward of $+1$ can be achieved for going right in state N. See Figure 5.2b for the $\mathcal{V}^*$-function. The $\mathcal{Q}$-space size is $|\mathcal{Q}| = 2N$. By using only random actions it is very unlikely to reach N. Once the attractor is discovered it might incline an agent to investigate more the left part of the graph instead of the valuable right side. This thesis also investigates a slightly modified exploration chain proposed by [3]. This environment simply adds from each state a shortcut to state 1 with a reward of -10 and modifies the remaining rewards. To see an exact description see [3]



(a) MDP of Exploration Chain



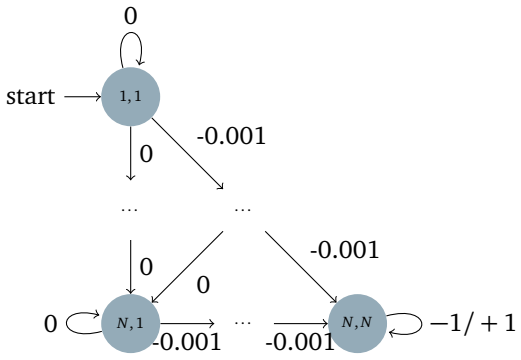
(b) Optimal $\mathcal{V}^*$-function (N=10)

Figure 5.2: MDP and Optimal $\mathcal{V}^*$-function for Exploration Chain

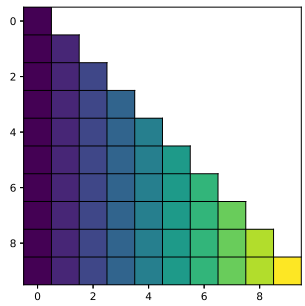
(a) The MDP of Exploration Chain is basically a deterministic chain, where the agent always starts at state 2. For big N the agent usually encounters the reward of +0.001 at the left end first. If the agent acts too greedily, he won't explore the right graph and thus doesn't know about the highest achievable reward. (b) For an example size of $N = 10$ the optimal $\mathcal{V}$-function was generated. Notice how the peak is positioned at the right end, however if the agent doesn't explore thoroughly enough his learned representation is flipped.

5.4 Deep-Sea-Exploration [7]

The Deep-Sea-Exploration $\mathcal{M}_D$ with $\mathcal{S} = \{(x, y) : x - y \leq 0\}$ and $\mathcal{A} = \{l, r\}$ is adopted from I. Osband et. al. [7]. Each episode lasts for $\Psi = N$ steps. It is an environment where the MDP is sampled stochastically but stays fixed during the agent explores it. Hence the problem effectively splits into a trivial $\mathcal{M}_{D1}$ and a non-trivial $\mathcal{M}_{D2}$ one, whereas their problem structure can be best imagined as a step like in Figure 5.5. Initially the agent starts at $(0, 0)$. During each step the agent has to choose between left and right. Both actions also move the agent one field down. Going right generally gives a reward of -0.001, whereas going left returns nothing. Depending on the sampled version either a reward of -1 or $+1$ can be received for moving right in (N, N). As a consequence they also have different optimal value functions $\mathcal{V}^*_{\mathcal{M}_{D1}}$ and $\mathcal{V}^*_{\mathcal{M}_{D2}}$, see Figure 5.3b. The space size is given by $|\mathcal{Q}| = N(N + 1)$.



(a) MDP of Deep-Sea-Exploration



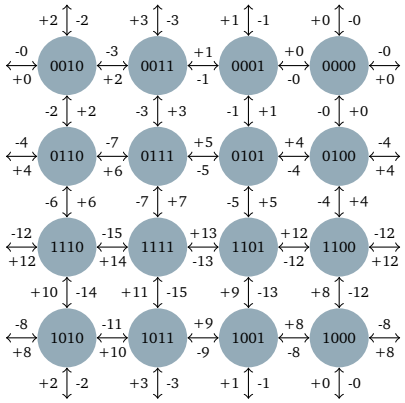
(b) Optimal $\mathcal{V}^*_{\mathcal{M}_{D2}}$-function (N = 10)

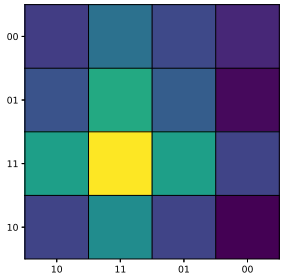
Figure 5.3: MDP and Optimal $\mathcal{V}^*$-function for Deep-Sea-Exploration

(a) The MDP of Deep-Sea-Exploration is given. Notice both rewards for staying in (N, N). By choosing one for an environment in prior this effectively factors into two tasks $\mathcal{M}_{D1}$ and $\mathcal{M}_{D2}$. A strategy is optimal for $\mathcal{M}_{D1}$, if left is chosen in each state. A strategy is optimal for $\mathcal{M}_{D2}$, if right is chosen in each state. (b) Only the +1 version is shown, because $\mathcal{V}^*_{\mathcal{M}_{D1}}$ has zero value everywhere. One can clearly see that it doesn't matter from which row inside of a column c the agent starts. If two different agents have the same number of steps remaining neither on of them has an advantage over the other, e.g. starting from any field of c and executing $N - x + 1$ times right, results in finishing with the highest achievable reward in $N - x + 1$ steps.

5.5 Binary Flip

The Binary Flip environment $\mathcal{M}_B$ with $\mathcal{S} = \{1, \ldots, 2^N\}$ and $\mathcal{A} = \{1, \ldots, N\}$ is a task where bits have to be flipped to achieve a high reward. Each episode lasts $\Psi = 8 * N$ steps. The state gets represented as a binary state vector $s \in \mathbb{B}^N$ with N bits and val(s) being the respective decimal number. At the beginning of an episode the state is initialized to 0. During each step the agent has to select one bit to flip. Assuming that action i was selected, the next state is calculated as $s - (2s - 1)\vec{e}_i$, and the reward as $\text{sgn}(2s_i - 1)\,\text{val}((1 - \vec{e}_i)s)$. A plotted MDP as well as the optimal $\mathcal{V}^*$-function for $N = 4$ is given in Figure 5.4. For this specific problem size both can be plotted in a natural way. The space size is $|\mathcal{Q}| = N2^N$.





(a) MDP of Binary-Flip-Environment ($N = 4$)

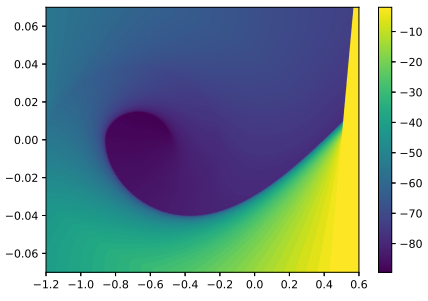
(b) Optimal $\mathcal{V}^*$-function of Binary Flip Environment (N=4)

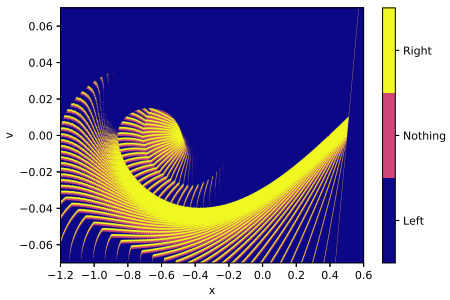
Figure 5.4: MDP and Optimal $\mathcal{V}^*$-function for Binary Flip Environment

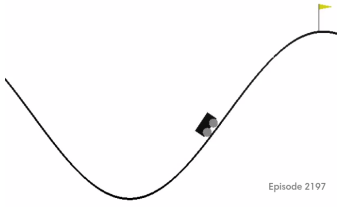
(a) The MDP for $N = 4$ was selected, because it can be displayed easily using a gray code. Each episode starts from state 0. From every node 4 actions are available, all transitions can be taken in both ways. However the received reward depends also on the direction a transition is used. As noticeable in the graph the rewards are relatively even balanced. The achievable rewards per episode lie in [-92, 92], whereas achieving the lowest reward is as difficult as receiving the positive reward. When an agent is initialized randomly it is very likely that he achieves some reward around 0. Due to this reason the plots in the evaluation usually start from 0.5. (b) The optimal $\mathcal{V}$-function states that state 15 is the most promising. But to really increase the rewards the agent has to traverse a subgraph, whereas the optimal policy repeats the same cycle over and over.

5.6 MountainCar-v0

A more complex environment is MountainCar-v0 with $\mathcal{S} = [-1.2, 0.6] \times [-0.07, 0.07]$ and $\mathcal{A} = \{l, n, r\}$ from OpenAI Gym [28]. The goal is to get a car uphill such that it reaches a goal position. An image of the environment can be seen in Figure 5.5c. The agent receives a reward of -1 for each state which is not the goal. Whenever the goal is reached the episode is over. So the agent has absolutely no clue where the goal is unless he reaches it occasionally. A state $s = [x, v] \in \mathcal{S}$ contains information about the current position and velocity, which is sufficient to develop a model representing a control policy. However for example using ϵ-Greedy DDQN takes some time steps to even find the goal. Sparse rewards are a challenge in real life as well, because there is not always an immediate feedback. Most often one has to decide by themselves, which subset of possible solutions are worth examining more deeply. Nevertheless this is a relatively simple task, but it gets used later one to test some algorithmic methods. The optimal $\mathcal{V}^*$-function is displayed in Figure 5.5a







(a) Optimal $\mathcal{V}$-function

(b) Optimal Policy

(c) Screenshot from [28]

Figure 5.5: Optimal $\mathcal{V}^*$-function, Policy and Screenshot for MountainCar

(a) The optimal $\mathcal{V}^*$ for MountainCar-v0 is shown. It was obtained by discretizing the continuous state space into a grid and using value iteration. (b) A discrete action policy was generated from the $\mathcal{Q}$-function. Most of the time either left or right is chosen instead of doing nothing. (c) A screenshot of the visualization was obtained from the rendering by OpenAI-Gym.

6 Experiments

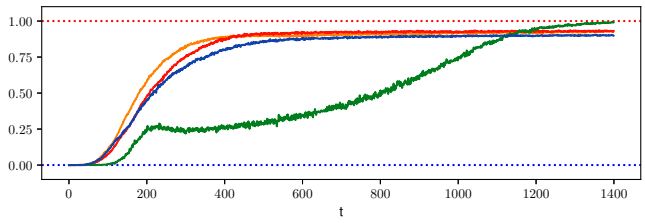
6.1 Deterministic MDP's

All hyper parameters were found by a grid search, the intervals are given in the corresponding sections. Additionally every reward curve was normalized to $[0, 1]$. The average of multiple runs along with it's variance was plotted together with the optimal (1) and minimal reward (0). Note that the visible y-range was sometimes reduced to ensure that all details can be observed.

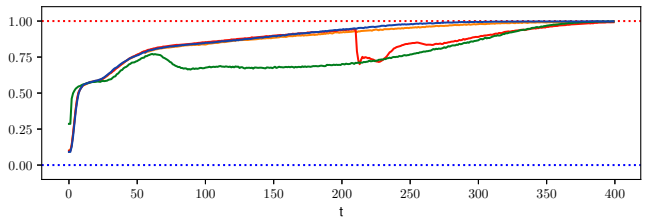
6.1.1 Old Exploration Strategies

In the following a comparison of old exploration strategies on MDPs is given, which includes Boltzmann, ϵ-Greedy , OI and UCB. The different runs on the problems are shown in Figure 6.1.

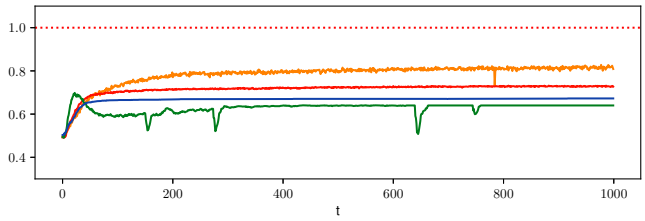
Boltzmann and ϵ-Greedy were performing equally well, but in (e) ϵ-Greedy converges to a lower value than the others. Action selection by ϵ-Greedy is inherently abrupt, because the best is prioritized and all others are taken uniformly. Since this is the reward received during training, one decision could influence the cumulative reward of the current episode.



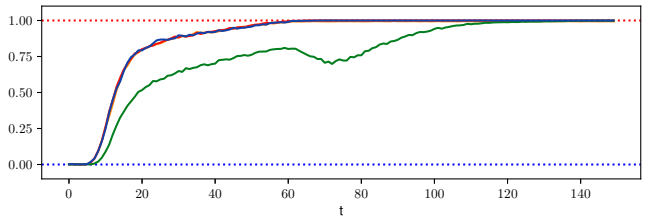
(a) Grid World with (N=10)



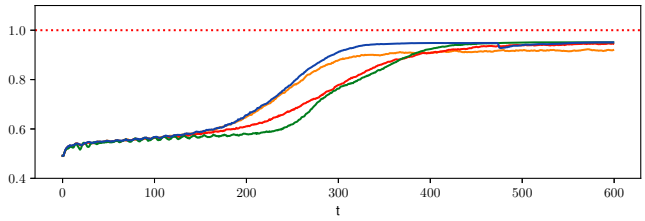
(b) Shared Chain with (N=33)



(c) Deep Sea Exploration with (N=10)



(d) Exploration Chain with (N=50)



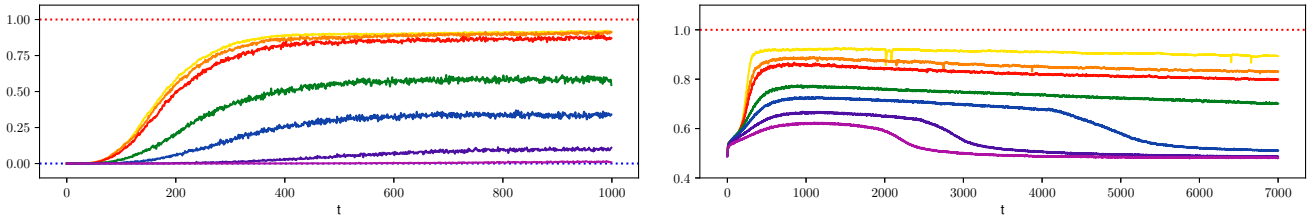
(e) Binary Flip with (N=6)

	ϵ-Greedy	Boltzmann	OI	UCB
$\mathcal{M}_G$	0.975 ± 0.008	0.952 ± 0.010	$\mathbf{1.000 \pm 0.000}$	0.912 ± 0.014
$\mathcal{M}_B$	0.927 ± 0.003	$\mathbf{0.965 \pm 0.001}$	0.952 ± 0.002	0.961 ± 0.001
$\mathcal{M}_S$	0.997 ± 0.001	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$
$\mathcal{M}_E$	0.997 ± 0.001	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$
$\mathcal{M}_D$	$\mathbf{0.840 \pm 0.018}$	0.749 ± 0.022	0.698 ± 0.023	0.673 ± 0.024

(f) Maximal achieved average reward for strategies

Figure 6.1: Evaluation of Old Exploration Strategies

(f) Color mapping and used parameters. All parameters were determined by grid search. The x-axis represents the elapsed episodes, whereas the y-axis shows the normalized achieved rewards per episode - this is the same for all plots in this thesis. Note (c) and (e) have a smaller plot range, so they can be better distinguished. For (d) the policies which achieved the maximum reward during their run are highlighted. In three cases all models of OI converged to the optimum, whereas Boltzmann and UCB achieved this only in two cases.



(a) Grid World (N=10)

(b) Binary Flip (N=6)

Figure 6.2: Different ϵ for ϵ-Greedy

The runs were executed for 7000 episodes, but only the characteristic intervals are given. For (a) there is also no change in performance ranking after the 800th episode. Note that in (b) all strategies, including $\epsilon = 0.005$ and $\epsilon = 0.01$ first increase in performance, but once peaked continuously decrease. For four of five problems $\epsilon = 0.001$ performs best over all episodes, whereas for Deep Sea Exploration $\epsilon = 0.05$ gets awarded significantly higher.

Boltzmann however uses it's knowledge to prioritize the actions based on their $\mathcal{Q}$ value using a softmax distribution. This effect twists for (c), because for this environment it is better to try out things at random instead of sampling actions by defining a distribution of the $\mathcal{Q}$-value. UCB has a equally well exploration performance on the problems, whereas in (b) and (e) it performs better in the shown time frame. OI is swinging in the beginning of (e) due to the environment's structure, where the reward is drastically changed by small changes in the reward. Due to the initialization a part of them tries the same actions at different time steps hence the periodic swinging.

6.1.1.1 ϵ-Greedy-Policy

	ϵ=0.001	ϵ=0.005	ϵ=0.01	ϵ=0.05	ϵ=0.1	ϵ=0.2	ϵ=0.3
$\mathcal{M}_G$	**0.975 ± 0.008**	0.956 ± 0.010	0.923 ± 0.013	0.630 ± 0.023	0.380 ± 0.023	0.122 ± 0.016	0.038 ± 0.009
$\mathcal{M}_B$	**0.927 ± 0.003**	0.891 ± 0.004	0.867 ± 0.004	0.775 ± 0.003	0.728 ± 0.003	0.669 ± 0.003	0.625 ± 0.003
$\mathcal{M}_S$	**0.997 ± 0.001**	0.977 ± 0.004	0.949 ± 0.006	0.785 ± 0.009	0.674 ± 0.007	0.586 ± 0.003	0.548 ± 0.002
$\mathcal{M}_E$	**0.997 ± 0.001**	0.978 ± 0.003	0.955 ± 0.005	0.751 ± 0.011	0.502 ± 0.014	0.153 ± 0.011	0.030 ± 0.005
$\mathcal{M}_D$	0.794 ± 0.020	0.829 ± 0.019	0.823 ± 0.019	**0.840 ± 0.018**	0.802 ± 0.020	0.766 ± 0.021	0.735 ± 0.022

Table 6.1: Best Achieved Average Reward for ϵ-Greedy

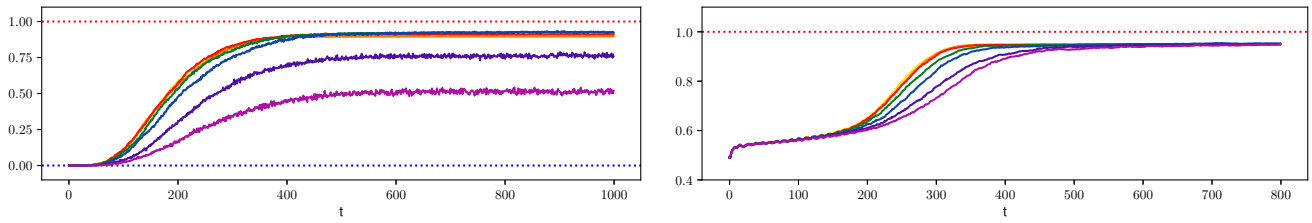
Different values for ϵ were tested and the results are presented in Figure 6.2. The plot for Grid World in Figure 6.2a shows that smaller values of ϵ achieve a better performance during training, because they often use the best action and are thus acting more safely. However if the parameter is too small it takes some time to explore the graph, unless the initialization was luckily. If ϵ is to small or not also depends on the problem structure. It might happen that it's performance first increases and afterwards decreases, like in Figure 6.2b. When the agent is following the policy, then even if he learned the optimal strategy he won't reach the full reward on average, because in ϵ cases a random action is taken.

6.1.1.2 Boltzmann-Policy

	β=0.001	β=0.005	β=0.01	β=0.05	β=0.1	β=0.2	β=0.3
$\mathcal{M}_G$	0.895 ± 0.015	0.899 ± 0.015	0.917 ± 0.014	0.935 ± 0.012	**0.952 ± 0.010**	0.797 ± 0.017	0.543 ± 0.022
$\mathcal{M}_B$	0.958 ± 0.002	0.949 ± 0.002	0.949 ± 0.002	0.959 ± 0.002	0.960 ± 0.001	**0.965 ± 0.001**	0.964 ± 0.001
$\mathcal{M}_S$	**1.000 ± 0.000**	**1.000 ± 0.000**	0.999 ± 0.001	0.886 ± 0.007	0.810 ± 0.006	0.669 ± 0.009	0.503 ± 0.010
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	0.653 ± 0.024	0.647 ± 0.024	0.650 ± 0.024	0.662 ± 0.024	0.695 ± 0.023	0.725 ± 0.022	**0.749 ± 0.022**

Table 6.2: Best Achieved Average Reward for Boltzmann

Evaluations show that the abstract behavior on these simplistic problems is comparable to that of ϵ-Greedy due to the stochastic action selection, view Figure 6.3a and Figure 6.2a. However it differs in the sense that it's strategy depends on the $\mathcal{Q}$-value, which creates different probabilities for actions in any state and if an action is better it will be automatically sampled more often. Ideally the parameter should be decreased over time so that it first explores and



(a) Grid World (N=10) **(b)** Binary Flip (N=6)

Figure 6.3: Different β for Boltzmann

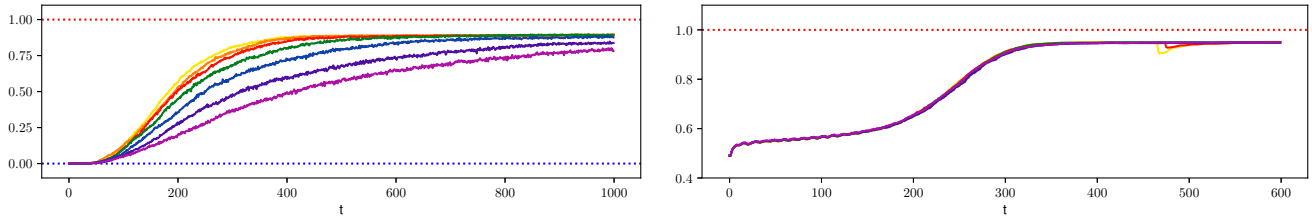
The runs were executed for 7000 episodes. For (a) there is also no change in performance ranking after the 800th episode. The characteristics are very similar to that of Figure 6.2, with the difference that the point of convergence not only depends on the hyper parameter, but also on the optimal $\mathcal{Q}$-function .

if enough information about the environment is collected starts to exploit his knowledge more often. The same can be applied to ϵ-Greedy . In robotics it is important to only slightly change the policy per step, because e.g. a walking robot might fall if he tries out something completely different than the episodes before.

6.1.1.3 UCB

	p=0.001	p=0.005	p=0.01	p=0.05	p=0.3	p=1	p=10
$\mathcal{M}_G$	0.891 ± 0.016	0.900 ± 0.015	0.900 ± 0.015	$\mathbf{0.912 \pm 0.014}$	0.897 ± 0.015	0.877 ± 0.017	0.243 ± 0.022
$\mathcal{M}_B$	0.961 ± 0.001	0.955 ± 0.002	$\mathbf{0.961 \pm 0.001}$	0.950 ± 0.002	0.951 ± 0.002	0.957 ± 0.001	0.960 ± 0.002
$\mathcal{M}_S$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	0.999 ± 0.001	0.856 ± 0.008	0.602 ± 0.002
$\mathcal{M}_E$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$	$\mathbf{1.000 \pm 0.000}$
$\mathcal{M}_D$	0.667 ± 0.024	0.643 ± 0.024	0.651 ± 0.024	$\mathbf{0.673 \pm 0.024}$	0.663 ± 0.024	0.663 ± 0.024	0.653 ± 0.024

Table 6.3: Best Achieved Average Reward for UCB

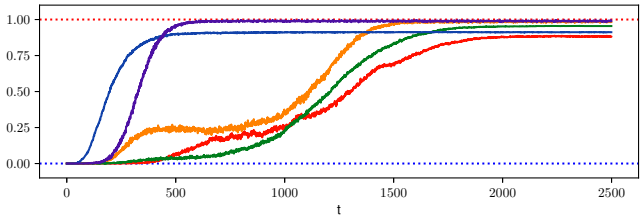


(a) Grid World (N=10) **(b)** Binary Flip (N=6)

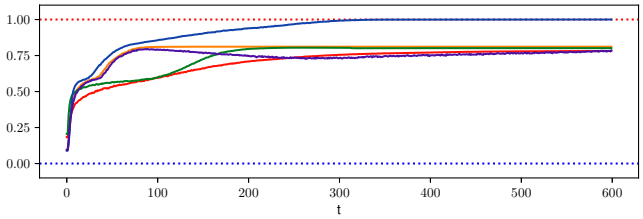
Figure 6.4: Different p for UCB

The runs were executed for 7000 episodes and averaged over 1500 models. (a) p strongly influences the performance, whereas smaller p receive better rewards during the first episodes, but latter on a higher parameter p receives more points. (b) Notice how the parameter doesn't really change the convergence.

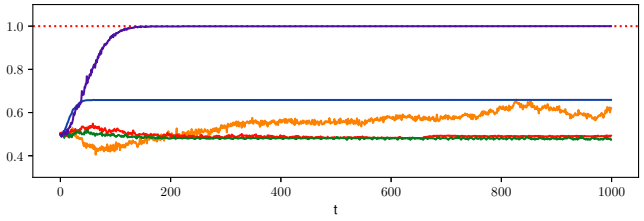
UCB uses a density $N(s,a)$ for reward shaping, which gets used to infer a bonus, which decreases with the number how often the agent visited that state-action pair. So the agent is inclined to move to states which weren't seen that often. In Figure 6.4a p influences the convergence drastically. Although the highest cumulative reward was achieved with $p = 0.05$, by using $p = 0.001$ he learned better at the beginning. For small values the agent approximates only a small upper-bound $\mathcal{Q}$, whereas high values can be interpreted as assuming that the confidence interval is very big. Even though there is quite a lot difference in the runs of Figure 6.4a, it's value doesn't really influence the behavior for Binary Flip.
Moreover in Table 6.3 it can be seen that the parameter doesn't have such a big influence on what is maximally achieved during learning. Especially for the Exploration Chain all tested parameter values guide the agent to converge to the optimum. Nevertheless there are task and parameter pairs, which don't work together well, e.g. Grid World and $p = 10$. Take a look at subsubsection 6.1.2.1 for a recent comparable approach.



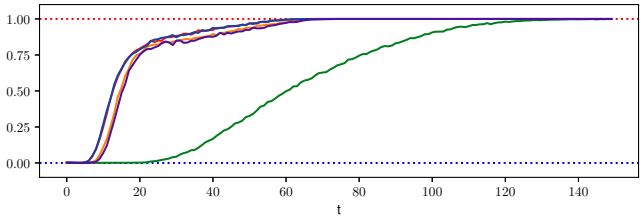
(a) Grid World with (N=10)



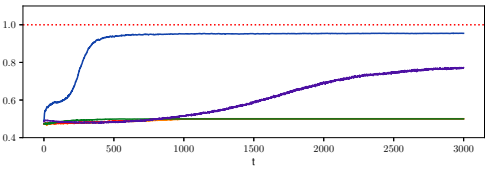
(b) Shared Chain with (N=33)



(c) Deep Sea Exploration with (N=10)



(d) Exploration Chain with (N=50)



(e) Binary Flip with (N=6)

	B	S-B	UCB-IG	PCE	PC-B
$\mathcal{M}_G$	0.994 ± 0.004	0.891 ± 0.016	0.957 ± 0.010	0.915 ± 0.014	**0.999 ± 0.002**
$\mathcal{M}_B$	0.501 ± 0.001	0.500 ± 0.000	0.500 ± 0.001	**0.956 ± 0.002**	0.819 ± 0.004
$\mathcal{M}_S$	0.811 ± 0.000	0.782 ± 0.003	0.805 ± 0.003	**1.000 ± 0.000**	0.878 ± 0.005
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	0.751 ± 0.022	0.549 ± 0.025	0.519 ± 0.025	0.659 ± 0.024	**1.000 ± 0.000**

(f) Maximal achieved average reward per episode

Figure 6.5: Evaluation of New Exploration Strategies

Shows the recent exploration strategies on the MDPs. They all show diverse exploration behavior like in (a), (b) or (c). In (f) the best mean average reward for each agent and problem is given.

6.1.2 Recent Exploration Strategies

This section compares the more recent strategies with each other, namely Bootstrapped DDQN (B-DDQN), Shared B-DDQN (SB-DDQN), UCB-IG-DDQN and Pseudo-Count Exploration (PCE). In Figure 6.5 reward plots from different runs are presented. In the maximum average reward along with it's episode are given.

Note that PCE is capable of exploring the environment very fast, due to it's intrinsic count-based bonus, for (b) and (d) it converges in episode 375 and 66 respectively. Whereas for other tasks it converges to clearly suboptimal solutions, e.g. in (a) and (c) the agent is only allowed to do one wrong action during a rollout to find the reward. Note that a bonus is only given, when the state-action pair was already executed. This phenomena is extremely present in (c), where it only achieves an average reward of 0.54. However the other agents also fails to find the correct way. When running on (d) it converges to the highest reward in only 66 episodes. It can be simply explained, because if the density for a state positioned right from the start was increased, all states in between were also encountered once. Obviously this doesn't apply for stronger connected graphs.

The remaining three are either Bootstrapped or a method derived from it. In task (c) all of them converge to the best solution. Nevertheless a different number of episodes is needed for reaching an average reward of 1, whereas B was approximately twice as fast as UCB-IG. SB elapsed episodes were situated between both. Generally they all show comparable performance, which is caused by the same underlying principle. The differences consist in the way action selection for running and training is done, e.g. UCB-IG-DDQN takes all heads into account by determining a confidence interval. SB-DDQN determines the next action for the target value by remembering using the best policy, which is calculated in every fixed number of time steps. All of them fail to achieve a reward significantly higher than the average in Binary Flip. Especially this environment has a much stronger connected graph then the other problems, which makes it more unlikely that one head finds the correct policy. It is even strengthened by the fact that there are more paths, which return a reward around zero than in the upper or lower reward regions. As this run only took Bootstrapped strategies into account with

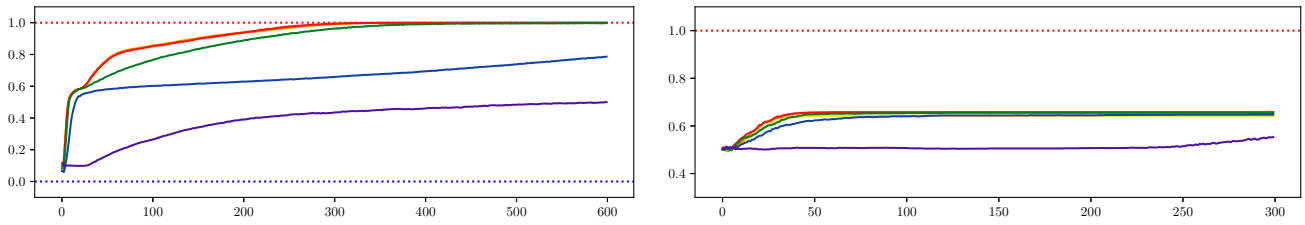
$K = H$, in subsubsection 6.1.2.4 is shown that for some tasks it is reasonable to set $H < K$.

Generally using a intrinsic reward like PCE is beneficial to more deeply explore actions, which were not that often performed. Additionally it can be integrated into other methods, e.g. the Bootstrapped like proposed in section 4.1. This combined strategy gets shortly evaluated in subsubsection 6.1.2.4 against each of it's components.

6.1.2.1 Pseudo-Count Exploration

The agent drives exploration by remembering which state-action pairs weren't visited that often. Whenever a state-action pair is observed, the reward gets shaped according to subsection 3.5.3, which integrates that knowledge into the $\mathcal{Q}$-value. Several values for $\beta \in [0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 1, 5, 10]$ were tested, however only the most expressing runs were plotted in Figure 6.6.

First of all smaller β - tend to find the reward faster. This phenomena can be explained easily, because to receive a reward bonus the agent needs to have already executed the state-action. When β is high the bonus will be relatively big as well. If in addition the rewards returned by the problem are very small, the agent explores the current region exhaustively instead of going to yet unseen actions. After some time the bonus gets smaller and erased from the bootstrapped $\mathcal{Q}$-function, which inclines the agent to further explore.



(a) Shared Chain (N=33) **(b)** Deep Sea Exploration (N=5)

Figure 6.6: Different β for Pseudo-Count-Exploration

The runs were executed for 7000 episodes and averaged over 1500 models. (a) Different values for β change the exploration, e.g. using $\beta = 10$ results in the worst run, whereas smaller values result in a fast convergence. (b) For this task the values except $\beta = 10$ are exploring very well at the beginning, but stop to explore near the end converging to a suboptimal solution. However for the chain-based environment all 1500 models converge to the optimal solution except for $\beta = 10$ on $\mathcal{M}_S$

	β=0.001	β=0.005	β=0.01	β=0.05	β=0.3	β=1	β=10
$\mathcal{M}_G$	0.900 ± 0.015	0.895 ± 0.015	0.894 ± 0.016	0.909 ± 0.015	**0.915 ± 0.014**	0.909 ± 0.015	0.310 ± 0.023
$\mathcal{M}_B$	0.948 ± 0.002	0.948 ± 0.002	0.949 ± 0.002	0.950 ± 0.002	0.949 ± 0.002	0.948 ± 0.002	**0.956 ± 0.002**
$\mathcal{M}_S$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	0.607 ± 0.002
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	0.643 ± 0.024	**0.659 ± 0.024**	0.645 ± 0.024	0.657 ± 0.024	0.657 ± 0.024	0.653 ± 0.024	0.647 ± 0.024

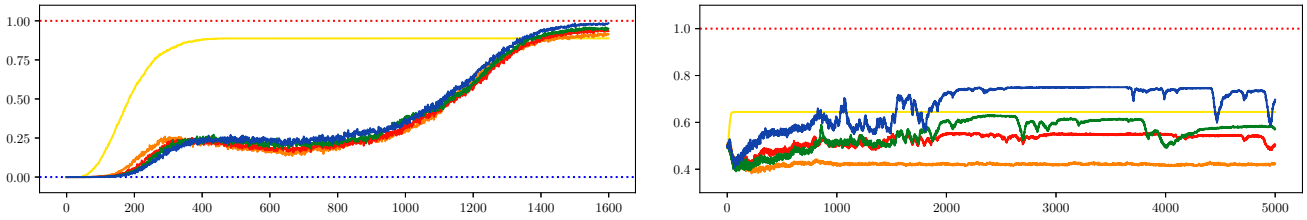
Table 6.4: Best Achieved Average Reward for Pseudo-Count-Exploration

6.1.2.2 Bootstrapped

	K=3	K=5	K=7	K=10
$\mathcal{M}_G$	0.947 ± 0.011	0.967 ± 0.009	0.981 ± 0.007	**0.994 ± 0.004**
$\mathcal{M}_B$	**0.501 ± 0.001**	0.500 ± 0.000	0.500 ± 0.001	0.500 ± 0.000
$\mathcal{M}_S$	**0.811 ± 0.000**	**0.811 ± 0.000**	**0.811 ± 0.000**	**0.811 ± 0.000**
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	0.515 ± 0.025	0.560 ± 0.025	0.631 ± 0.024	**0.751 ± 0.022**

Table 6.5: Best Achieved Average Reward for Bootstrapped

The general concept of Bootstrapped is very simplistic: K different $\mathcal{Q}$-functions are initialized and each training sample gets used to learn H heads. In the overview on recent strategies the agent with value $K = 1$ was left away, because this

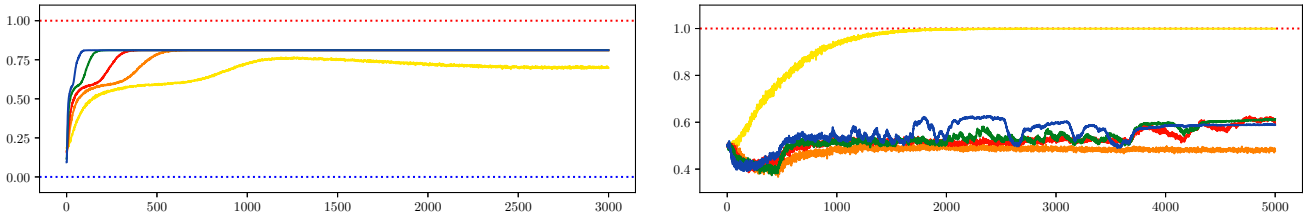


(a) Grid World (N=10)

(b) Deep Sea Exploration (N=5)

Figure 6.7: Different K for Bootstrapped

The runs were executed for 7000 episodes and averaged over 1500 models. (a) With $K = 1$ it starts to converge very fast, but is not able to reach the optimum in the end. Higher values get even better after 1500 episodes. (b) shows that the runs for Deep Sea Exploration exhibits similar behavior.



(a) Shared Chain (N=33)

(b) Deep Sea Exploration (N=5)

Figure 6.8: Different H for Bootstrapped with $K = 7$

The runs were executed for 7000 episodes and averaged over 1500 models. On the left side all runs from Grid World were plotted and on the right side the Deep Sea Exploration is shown. For the first smaller H give lower results at the beginning, and better ones in the end. As the absolute difference between them is neglectable, the best performing for (a) is $H = 7$. As opposed to (b) where it is better to choose $H = 1$.

effectively reduces to a Greedy-only one. In fact it performed better for the Binary Flip Environment than Bootstrapped, however as this strategy only exploits knowledge it can't be scaled up to more complex tasks.

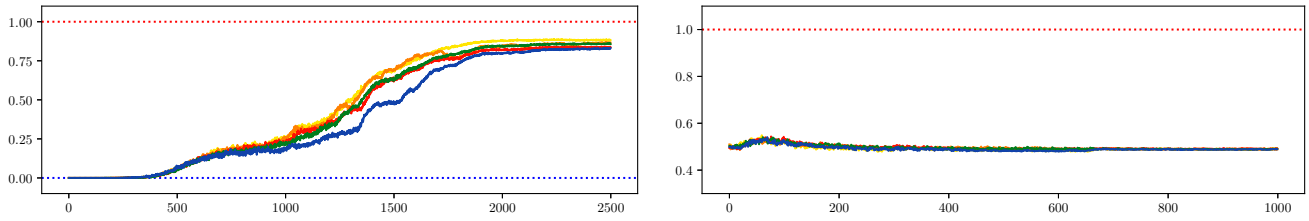
See Figure 6.7 for plots where values from $K \in [1, 3, 5, 7, 10]$ and $H = K$ were tested. If only one head is active the agent starts to explore earlier, but is not able to reach the optimum. For both problems higher K values yield a higher cumulative reward when the episode runs infinitely long. In Figure 6.7b for $K = 10$ it is very instable during the first episodes. This effect is caused on one hand by the policy-to-reward structure and on the other by the variance of all heads. Obviously the diversity will reduce over time, however it can be maintained by setting $H < K$.

H can be set to K for some problems. Note that the diversity automatically increases when using a neural network, as training with the same samples on neural network keeps the variance up. Atari games were played by [2] using $K = H$. Nevertheless as this examines the tabular case and training the heads doesn't influence another head, smaller values for H were tested. A plot is given in Figure 6.8 where different H were evaluated. Smaller values tend to explore not that fast, but might find the correct policy where others fail, e.g see Figure 6.8b. As shown in Table 6.6 that for all tasks except the Shared Chain the best solution was achieved by using $H = 1$

	H=1	H=2	H=3	H=5	H=7
$\mathcal{M}_G$	**0.989 ± 0.005**	0.610 ± 0.025	0.728 ± 0.023	0.858 ± 0.018	0.978 ± 0.007
$\mathcal{M}_B$	**0.501 ± 0.002**	0.491 ± 0.001	0.500 ± 0.000	0.500 ± 0.000	0.500 ± 0.000
$\mathcal{M}_S$	0.805 ± 0.007	**0.811 ± 0.000**	**0.811 ± 0.000**	**0.811 ± 0.000**	**0.811 ± 0.000**
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	**0.999 ± 0.001**	0.512 ± 0.025	0.626 ± 0.024	0.614 ± 0.025	0.627 ± 0.024

Table 6.6: Best Achieved Average Reward for Bootstrapped $K = 7$

6.1.2.3 Shared Bootstrapped



(a) Grid World (N=10) **(b)** Deep Sea Exploration (N=5)

Figure 6.9: Different S for Shared Bootstrapped with $K = 5$

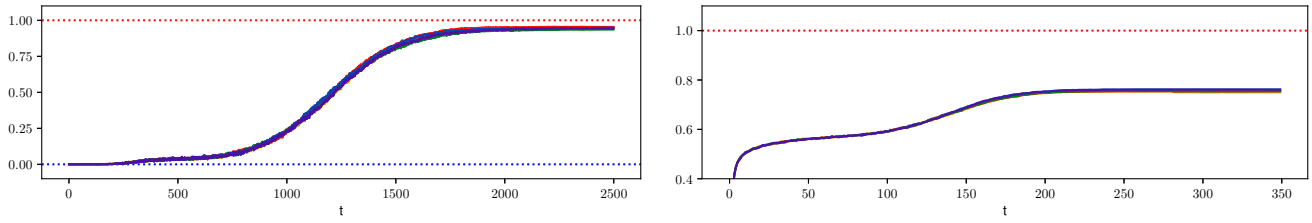
The runs were executed for 7000 episodes and averaged over 1500 models. S is basically the number of step until the new best agent gets determined. For the Deep Sea Exploration task it is not capable of learning better than the average, whereas Bootstrapped at least finds some better solutions. On the right it basically converges to a suboptimal solution

In general the evaluations from Figure 6.9 show that Shared Learning gives no improvement on Bootstrapped for the investigated tasks. One possible reason is that the selection which head should be selected is based on which agent has the highest value for $\mathcal{Q}(s_t, a_t)$. However as this is only a very rough approximation to select the best head it might actually hinder the learning process. the step parameter controls a little bit the speed of convergence, but for (b) it doesn't change the abstract behavior.

	S=10	S=30	S=50	S=70	S=100
$\mathcal{M}_G$	**0.891 ± 0.016**	0.871 ± 0.017	0.844 ± 0.018	0.866 ± 0.017	0.835 ± 0.019
$\mathcal{M}_B$	0.500 ± 0.000	**0.500 ± 0.000**	0.500 ± 0.000	0.500 ± 0.000	0.500 ± 0.000
$\mathcal{M}_S$	**0.782 ± 0.003**	0.778 ± 0.004	0.780 ± 0.004	0.780 ± 0.004	0.776 ± 0.004
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	**0.549 ± 0.025**	0.535 ± 0.025	0.544 ± 0.025	0.541 ± 0.025	0.541 ± 0.025

Table 6.7: Best Achieved Average Reward for Bootstrapped $K = 7$

6.1.2.4 UCB-InfoGain Bootstrapped



(a) Grid World (N=10) **(b)** Shared Chain (N=33)

Figure 6.10: Different λ, ρ for UCB-InfoGain with $K = 7, H = 7$

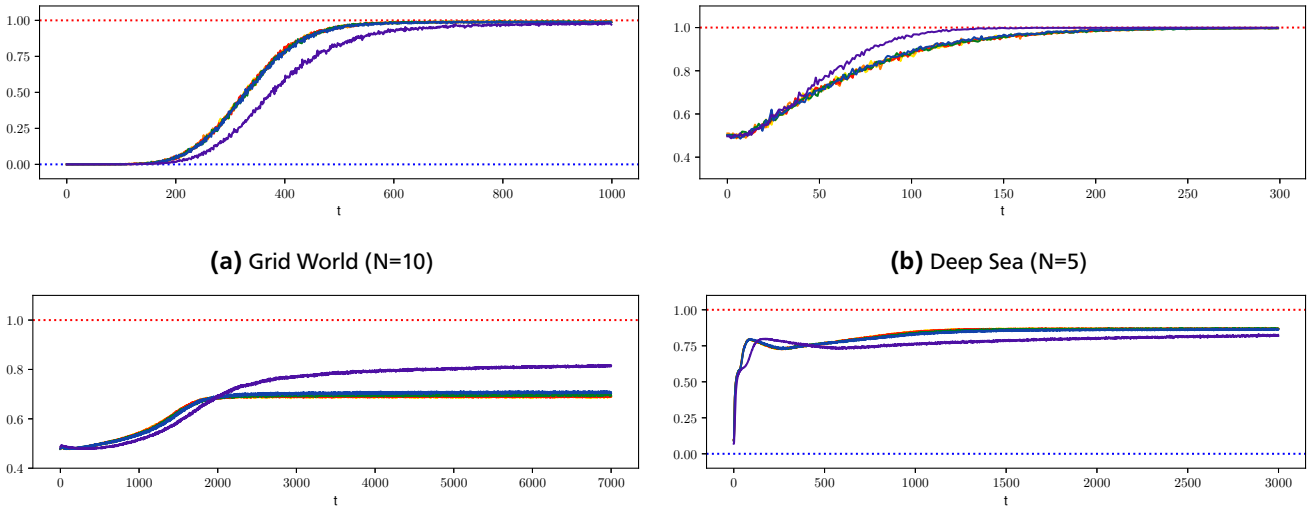
The runs were executed for 7000 episodes and averaged over 1500 models. S is basically the number of step until the new best agent gets determined. For the Deep Sea Exploration task it is not capable of learning better than the average, whereas Bootstrapped at least finds some better solutions. On the right it basically converges to a suboptimal solution

All combinations from $\lambda \in [0.005, 0.01, 0.05, 0.1]$ and $\rho \in [0.005, 0.01, 0.05, 0.1]$ were evaluated, whereas the corresponding plots are given in Figure 6.10. Note that changing the values doesn't really alter performance. However this might occur due to the different heads converging relatively fast to each other when $H = K$. As a consequence the variance amongst them goes to 0 and hence the average KL-divergence from the average softmax of each $\mathcal{Q}$-function also approaches 0. So they converge to act like on greedy agent in the end. However if a neural network is used the variance is inherently kept up, see also subsubsection 6.2.1.2.

	λ=0.005 ρ=0.005	λ=0.005 ρ=1	λ=0.01 ρ=0.1	λ=0.05 ρ=0.005	λ=0.1 ρ=0.005	λ=0.1 ρ=1
$\mathcal{M}_G$	0.941 ± 0.012	0.936 ± 0.012	**0.957 ± 0.010**	0.951 ± 0.011	0.952 ± 0.011	0.939 ± 0.012
$\mathcal{M}_B$	0.500 ± 0.000	**0.500 ± 0.000**	0.500 ± 0.000	0.500 ± 0.000	0.500 ± 0.000	0.500 ± 0.000
$\mathcal{M}_S$	0.756 ± 0.005	0.756 ± 0.005	0.750 ± 0.005	0.789 ± 0.004	**0.805 ± 0.003**	0.804 ± 0.003
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	0.516 ± 0.025	0.516 ± 0.025	**0.518 ± 0.025**	0.513 ± 0.025	0.509 ± 0.025	0.507 ± 0.025

Table 6.8: Best Achieved Average Reward for Bootstrapped $K = 7$

6.1.2.5 Bootstrapped Pseudo-Count



(a) Grid World (N=10)

(b) Deep Sea (N=5)

(c) Binary Flip (N=6)

(d) Shared Chain (N=33)

Figure 6.11: Different β for Pseudo-Count Bootstrapped with $K = 7$

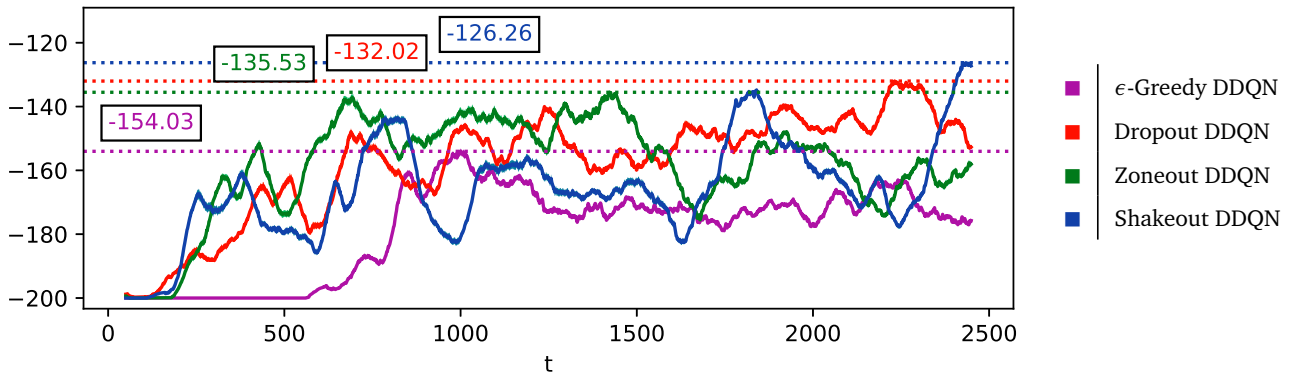
A simple evaluation on Grid World and Deep Sea. The agent converges rapidly for both problems to a good solution. The exact value can depend on the problem as seen in Table 6.9. Additionally the learning curve is relatively smooth despite the variance.

This section evaluates the method proposed in 2. Each run lasted about 2500 episodes, whereas only relevant parts are shown. For the basic Grid World the agent rapidly converges to the optimum. In the case of the Shared Chain it is not capable of receiving the full rewards for all models. Even for the Binary Flip task it is exploring the environment and performs better than all strategies except the Pseudo-Count version. It is not only capable of receiving some reward for Deep Sea Exploration, but even converges to the optimal solution, such that all models – randomly initialized – found the way.

	β=0.001	β=0.005	β=0.01	β=0.05	β=0.1	β=1
$\mathcal{M}_G$	0.997 ± 0.003	0.997 ± 0.003	**0.999 ± 0.002**	0.997 ± 0.003	0.997 ± 0.003	0.998 ± 0.002
$\mathcal{M}_B$	0.698 ± 0.004	0.700 ± 0.004	0.700 ± 0.004	0.704 ± 0.004	0.713 ± 0.004	**0.819 ± 0.004**
$\mathcal{M}_S$	0.876 ± 0.005	0.873 ± 0.004	0.874 ± 0.005	**0.878 ± 0.005**	0.873 ± 0.005	0.867 ± 0.007
$\mathcal{M}_E$	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
$\mathcal{M}_D$	0.999 ± 0.001	0.999 ± 0.001	0.998 ± 0.002	0.998 ± 0.002	0.999 ± 0.002	**1.000 ± 0.000**

Table 6.9: Best Achieved Average Reward for Pseudo-Count Bootstrapped with $K = 7$ and $H = K$

In this experiment one density model is used for all heads to achieve this result. All heads are drawn to state-action pairs which weren't explored that exhaustively. One of the next steps is to evaluate for $H < K$, this should actually engage each head to explore a slightly different region of the graph, which increases the variance in between the heads increasing the exploration. However due to time constraints it was not further examined in this thesis.



(a) MountainCar-v0 **(b)** Legend

Figure 6.12: Evaluation of Regularized Exploration

In (a) three runs with different regularization techniques are compared to the ϵ-Greedy run. The table (b) contains the color mappings. Each run lasts about 2500 episodes, whereas for each one model achieved -83 one time. The average maximum reward reached by each of them was also plotted.

6.2 Open-AI-Tasks

6.2.1 MountainCar-v0

In this section a self-made DDQN base-implementation with structure [2, 256, 256, 3] was utilized. All runs were executed five times over 2500 episodes. The replay memory is always 50000 and the sampled batch size per training step is set to 128. After 500 steps the weights are copied to the target network. These parameters stay the same for all upcoming evaluations. To make the plots more clearly, each independent run is convoluted using a right-aligned exponential decayed kernel with width 50 and $\lambda_d = 0.99$. After the runs have been filtered their value and variance is averaged over all five models. The boundary cases are simply skipped and hence a run starts from episode 50 and stops at 2450. All plots shows the maximum of each curve as a horizontal line in the same color.

To check if the code works, an initial evaluation of DDQN with an ϵ-greedy strategy was performed. Generally the strategy suffers from getting worse once it surpassed it's peak episode. After the correctness was verified, three regularization techniques were examined more deeply, to determine the usefulness of these techniques. The implementation was extended subsequently to algorithm 3, whereas the specific evaluations were discussed in the text.

6.2.1.1 Regularized DDQN (R-DDQN)

Regularized DDQN (R-DDQN) uses a regularization technique to simulate the heads for Bootstrapped DDQN (B-DDQN). At the beginning of every episode one mask is sampled from a Bernoulli distribution with probability φ. With a fixed network structure the mask's shape depends solely on the type of regularization, e.g Zoneout can't bypass the first layer since it contains a non-square weight matrix. Applying a mask produces a slightly different subnetwork. All induced subnetworks together group the used ensemble. The main conceptual difference from BDDQN's is that for training the sampled mask stays unchanged for one complete episode. It should be pointed out that training for multiple steps on the same network might result in strongly shifted $\mathcal{Q}$-functions for all other networks. However the extent strongly depends on how many steps the episode lasted, the used learning rate as well and the average number of shared weights – which in turn is determined by φ. To mitigate for the problem of extreme changes in later episodes, φ gets linearly increased to reach one when a previously specified number of time steps is trespassed. When this occurs the ensemble shrinks to a single unified model. All other algorithmic parts of DDQN remain unchanged. For Dropout, Zoneout and Shakeout one run was selected and presented in Figure 6.12.

All RDDQN agents reached the goal approximately twice as fast as ϵ-Greedy DDQN. Once an agent found a path to the flag, this information was stored in his memory and played back during training. The knowledge gets step-wise incorporated in a subnetwork. Although the learned policy can be used to reach the top of the right hill, he fails in exploring and finding a better control policy. Their abstract behavior is very similar, while for one it peaks earlier, but

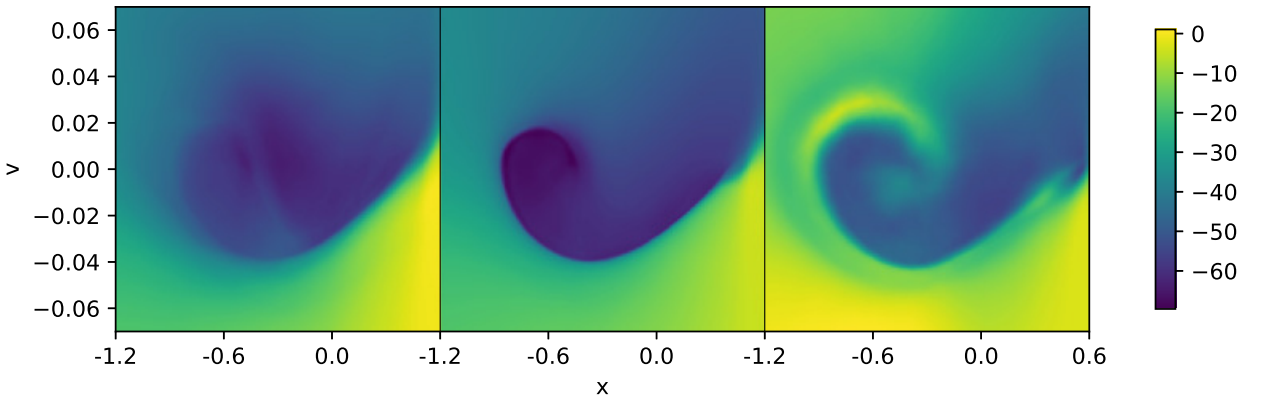
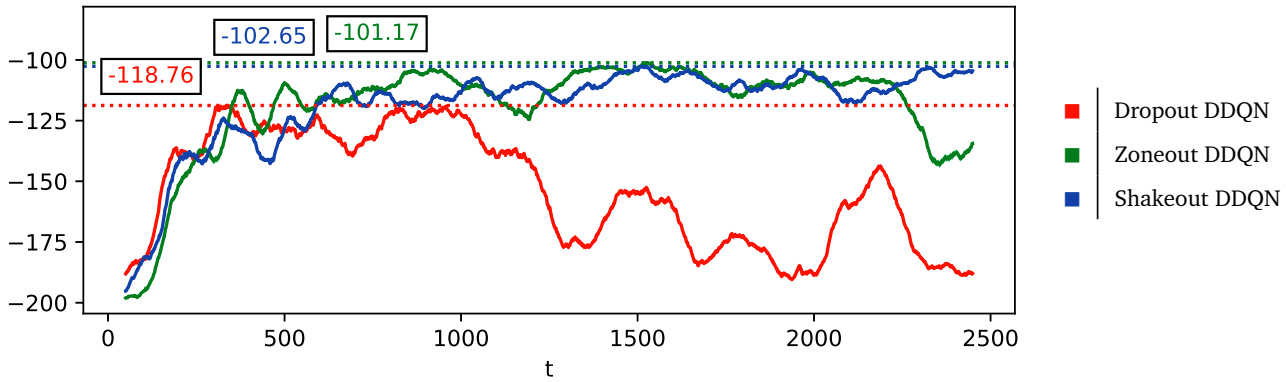


Figure 6.13: Learned $\mathcal{V}$-functions for Regularized Exploration

All $\mathcal{Q}$-functions were plotted after 2500 episodes. Their values are normalized such that one color always corresponds to the same number. The techniques used from left to right are Dropout, Zoneout and Shakeout. They differ slightly in their approximation. When compared to Figure 5.5a Zoneout ranks the best, because it's values are closer to the optimal than of the others.

therefore performs worse when another technique peaks. To provide a better insight into the quality of all models a plot of each approximated $\mathcal{V}$-function after 2500 episodes is given in Figure 6.13. Zoneout's relative result compared to Figure 5.5a is obviously the best, followed by Dropout and Shakeout. Shakeout has problems to represent the optimal policy, induced by the regularization scheme.

6.2.1.2 Regularized-UCB DDQN (R-UCB-DDQN)



(a) MountainCar-v0 **(b)** Legend

Figure 6.14: Regularized-UCB DDQN with $\rho = 0$

All strategies start good, but however the Dropout approach is not able to maintain the knowledge for later reuse. After around 500 episodes it actually starts to decrease it's performance. Zoneout and Shakeout are able to remember their knowledge and use it in the next steps as well.

The implementation was extended to use an ensemble of masks, which are sampled at the beginning of each episode. In addition an ensemble of masks is sampled before each training step, which then gets used for optimization. This results in the algorithm algorithm 3. For more details on the used configuration see subsubsection 6.2.1.1. A plot of one averaged run with $\rho = 0$ can be seen in Figure 6.14.

Dropout explored the environment very thoroughly in the beginning and performed even better than the others. However as soon as Dropout reached it's second peak, the approach is not capable to improve further. The moving average reward clearly followed a negative trend. Nevertheless Zoneout and Shakeout actually performed best when compared to Dropout or Figure 6.14. Like Dropout they create a lot of variance in their exploration, but their strength relies in

memorizing and generalization of the information. This phenomena can be verified in the plots as they can hold the moving average reward in the interval $[-125, -100]$ for approximately 1500 episodes. To assess the approximation quality, the $\mathcal{V}$-function after episode 2500 for each technique is visualized in Figure 6.15. As in Figure 6.13 Zoneout learned the best policy. Dropout resulted in a reasonable $\mathcal{V}$-function, but has relative high values around the area where the cart starts. This time Shakeout produced a better approximation than in Figure 6.13, it looks like a slightly morphed version of the optimal $\mathcal{V}$-function. If one $\mathcal{V}$-function is more accurate than the other, this does not necessarily induce that one technique is better than the other, e.g. it might be beneficial for exploration, if approximation errors are included. Additionally the $\mathcal{Q}$-functions for the different actions don't differ much hence it can occur through the neural network itself that this round one of them is highest, whereas after training one step, the other one has the highest value for action selection.

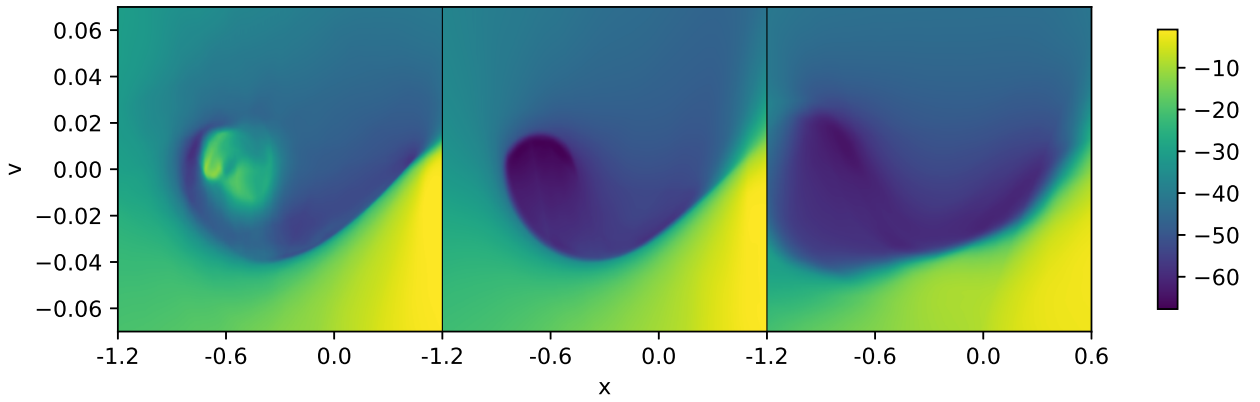
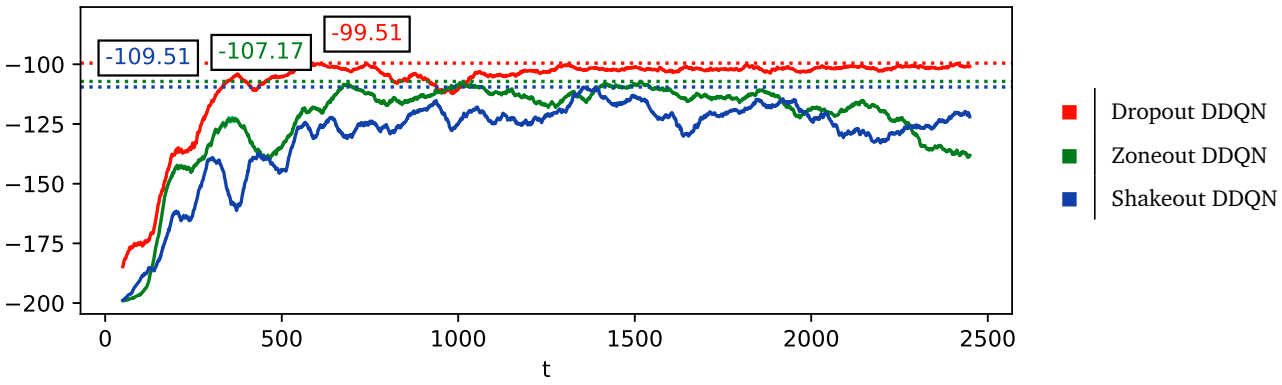


Figure 6.15: Learned $\mathcal{V}$-functions of R-UCB-DDQN for each regularization technique

All $\mathcal{Q}$-functions were plotted after 2500 episodes. Their values are normalized such that one color always corresponds to the same number. The techniques used from left to right are Dropout, Zoneout and Shakeout. They differ slightly in their approximation. When compared to Figure 5.5a Zoneout again ranks the best, because it's values are closer to the optimal than of the others.

Finally another run was performed with $\rho = 0.005$. This run created interesting results and can be viewed in Figure 6.16. Compared to the previous ones, it performed quite well. All other graphs imply that the agent is somehow forgetting his learned knowledge. However when looking at the run of Dropout it seems like that it converges to an equilibrium point, with a relative good result. Note that this task is solved when the agent achieves -110 for 100 steps.



(a) MountainCar-v0 **(b)** Legend

Figure 6.16: Regularized-UCB DDQN with $\rho = 0.005$

All strategies start good, but this time Dropout approaches really fast -99.51. It is not only able to reach that level easily, but it is also able to maintain it with minimal variance. Although the parameters weren't changed compared to Figure 6.14, they even start to diverge around $t = 1000$. Generally these methods have to be tuned to work in a real setting. The small modification $\rho = 0.005$ improves one technique extremely, whereas it hurts other ones.

7 Conclusion & Outlook

This thesis collected existing exploration strategies, explained the basic functionality of them and examined their exploration behavior on small scale MDP's. Old approaches can be implemented quite easily, as seen in the evaluation they work very well on small scale problems, but however there is the need for other methods. One of these is Pseudo-Count-Exploration (PCE). It can be helpful to equip an agent with density model. Naturally humans perform this kind of exploration by trying out novel things. This strategy showed a good performance on the tasks as compared to others recent ones. Additionally it is scalable and was used by the authors [9] to play Atari games.

Another interesting type of exploration called Bootstrapped DQN (BDQN) was investigated. The inner workings are rather simplistic, but can be used for managing Deep Exploration, which is a term introduced by [2]. During the analysis it was found, that PCE and BDQN won't exclude each other. Evaluations on a combined tabular version of both with $\mathcal{Q}$-learning shows that this is indeed a reasonable combination, being the only strategy to completely solve the Deep Sea Exploration task.

It would be interesting to see more research on this, e.g. there is the need for an appropriate scalable density model, which can be used to track novelty for the single heads. A possible approach would be to condition it on the head, but this would restrict the number of different heads. Maybe there are more sophisticated approaches using an ensemble of shared networks, where one doesn't have to keep track of the single networks and thus give rise to many different combinations. Although not applicable to neural density models this thesis proposed a way of using regularization to guide exploration instead of managing multiple copies of the network independently.

In the examples was shown that either Dropout, Zoneout or Shakeout, which depend on a mask are able to create a diversity by just randomly sampling a mask. One of the initial thought was that at the beginning there will be much exploration as the submodels are all initialized at random, but over time they should more or less agree. It was found helpful for convergence to decrease the regularization over the time steps. Shared Learning approach was tried to be integrated to the R-UCB-DDQN, but experiments showed that it combined with the regularization approach wasn't competitive to the version without.

Future research should more deeply connect intrinsic motivation and neural networks. I think that these are awesome techniques to subtle guide the agent, but that the agent himself needs to explore around the guidance. I would also like to see some work on altering the structure of the network to guide exploration, e.g. formulated itself as a reinforcement learning problem, where the objective is to alter the structure such that the agent maximizes rewards.

Bibliography

[1] R. Y. Chen, S. Sidor, P. Abbeel, and J. Schulman, "UCB and infogain exploration via $\boldsymbol{Q}$-ensembles," *CoRR*, vol. abs/1706.01502, 2017.

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

[3] R. R. Menon and B. Ravindran, "Shared learning: Enhancing reinforcement in *q*-ensembles," *arXiv preprint arXiv:1709.04909*, 2017.

[4] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[5] D. Krueger, T. Maharaj, J. Kramár, M. Pezeshki, N. Ballas, N. R. Ke, A. Goyal, Y. Bengio, H. Larochelle, A. Courville, *et al.*, "Zoneout: Regularizing rnns by randomly preserving hidden activations," *arXiv preprint arXiv:1606.01305*, 2016.

[6] G. Kang, J. Li, and D. Tao, "Shakeout: A new regularized deep neural network training scheme.," 2016.

[7] I. Osband, D. Russo, Z. Wen, and B. Van Roy, "Deep exploration via randomized value functions," *arXiv preprint arXiv:1703.07608*, 2017.

[8] 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, pp. 529–533, 2015.

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

[10] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, pp. 1050–1059, 2016.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1.

[12] K. P. Murphy, "Machine learning: a probabilistic perspective," 2012.

[13] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[14] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," 2016.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

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

[17] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," *arXiv preprint arXiv:1707.01495*, 2017.

[18] E. Even-Dar and Y. Mansour, "Convergence of optimistic and incremental q-learning," in *Advances in neural information processing systems*, pp. 1499–1506, 2002.

[19] M. C. Machado, S. Srinivasan, and M. H. Bowling, "Domain-independent optimistic initialization for reinforcement learning.," 2015.

[20] N. Chentanez, A. G. Barto, and S. P. Singh, "Intrinsically motivated reinforcement learning," in *Advances in neural information processing systems*, pp. 1281–1288, 2005.

[21] A. van den Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu, "Conditional image generation with pixelcnn decoders," *CoRR*, vol. abs/1606.05328, 2016.

[22] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," *arXiv preprint arXiv:1601.06759*, 2016.

[23] H. Larochelle and I. Murray, "The neural autoregressive distribution estimator," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 29–37, 2011.

[24] B. Uria, I. Murray, and H. Larochelle, "A deep and tractable density estimator," in *International Conference on Machine Learning*, pp. 467–475, 2014.

[25] M. Germain, K. Gregor, I. Murray, and H. Larochelle, "Made: masked autoencoder for distribution estimation," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 881–889, 2015.

[26] B. Uria, I. Murray, and H. Larochelle, "Rnade: The real-valued neural autoregressive density-estimator," in *Advances in Neural Information Processing Systems*, pp. 2175–2183, 2013.

[27] D. Russo, B. V. Roy, A. Kazerouni, and I. Osband, "A tutorial on thompson sampling," *CoRR*, vol. abs/1707.02038, 2017.

[28] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.