Hochschule Aalen

Master´s thesis
Course Computer Science

Beytullah Ince

Reinforcement Learning for Strategy Games

First examiner:	Prof. Dr. Ulrich Klauck
Second examiner:	Prof. Dr. Martin Heckmann
Submission date	Januar 2023

Statutory declaration

I, Beytullah Ince, hereby declare that the present information in this thesis has been written by me independently. Furthermore, I assure that I have not used any sources or aids other than those indicated, and that all statements taken verbatim or in spirit from other writings have been identified. The same applies to attached sketches and illustrations. In addition, I assure that the work in the same or similar version has not yet been part of a study course or examination.

Bietigheim-Bissingen, 08 December 2022

____

BEYTULLAH INCE

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

DDQN Double Deep Q-Network

DQN Deep Q-Network

MDP Markov Decision Process

POMDP Partially Observable Markov Decision Process

PPO Proximal Policy Optimization

RL Reinforcement Learning

TD Temporal Difference

TRPO Trust region policy optimization

1. Introduction

In recent years, research in the field of Artificial Intelligence progressed to the point where Artificial Intelligence systems are able to perform on expert level in their respective fields in complex domains. Specifically, in the strategic games domain. Systems can perform at a world-class level and even defeat humans at that level.

One of these games is *Go*. Go is a turn-based board game in which the goal is to surround opposing pieces with your own on a (usually) 19×19 board. Deepmind, an Artificial Intelligence subsidiary of Google, developed a system named *AlphaGo*. AlphaGo was the first system to, play at a world-class level and defeat a world champion in Go alongside various amateur and professional players. Previously, systems were only able to play at amateur level, due to the game its complexity. They were using search trees to play the game. AlphaGo is using search trees as well, however, it is additionally using deep neural networks. Games versus amateur players were utilized to improve gaming by gaining a better grasp of the human gameplay. Also, AlphaGo was training by playing versus itself and learn its own mistakes (Silver et al., 2017).

say neural network Another example of a system that is playing on world-class level is *OpenAI Five*. It is a system developed by OpenAI, an Aritifical Intelligence research company. The system is playing the game *Dota 2*. Dota 2 is an online real-time strategy game played by 5 versus 5 players. Each player chooses a character from a pool of 123 heroes, each of which fulfills a specific role in the game. The aim is to destroy a specific opposing structure that is located at the core of the opposing team its base. To achieve this goal, players must collect resources to enhance the abilities and attributes of their heroes and destroy more opposing structures to clear the way to the enemy base. OpenAI Five uses five different agents, each of which controls one of 18 heroes. Similar to AlphaGo, OpenAI Five was trained with various concepts, playing against itself and other human players, such as amateur players from the development team, or online versus amateur players. The level of the amateur players gradually increased, as OpenAI Five got better. The level of those players were measured with the game its own rating system. Gradually, it played against better teams and eventually, OpenAI Five was able to defeat various professional teams, including the world champions of that time (Berner et al., 2019).

Deepmind developed another Aritifical Intelligence system that is performing at the highest levels in its respective game. The name of this system is *AlphaStar* and its playing the game *Starcraft II*. Starcraft II is another online real-time strategy game. The goal is the same: the destruction of a specific enemy structure in the center of the enemy base. The difference to Dota 2 is, that in Dota 2, the player is controlling one hero at any time, however, in Straftcraft II this is different. The player must control many different units with different purposes at the same time and plays (mostly) against a single player. He has to gather resources to build structures and develop all his units and structures. The structures are responsible for creating units, that will either collect resources or attack the enemy. The system is able to play the game without any restrictions by using deep neural networks. It was able to defeat various amateur players in online matches with varying skill levels based on the game its rating system. Eventually, the system was able to defeat top professional players in matches of five games, without losing once (Vinyals et al., 2019).

What all these systems have in common is, they are using Reinforcement Learning in their architectures. Reinforcement Learning is a Machine Learning paradigm where an Artificial Intelligence system is learning by interacting with an environment and is rewarded based on its actions, i.e. the system is playing a game either is positively rewarded for an action that has a positive outcome, or in the opposite case, negatively.

This thesis will cover the fundamentals of Reinforcement Learning. Important concepts and technical terms will be explained, as they are necessary in order to comprehend how the presented four methods are working. Custom implementations of the classic board games *Mastermind* and *Battleship* are programmed. The objectives, rules and gameplay of those games will be presented, followed by the implementation and interaction details regarding the games and methods. Finally, experiments will be conducted that will show how the methods perform in these games.

2. Fundamentals

FRAGE: Diagramme aus tensorboard, wie viel Glätten?

replace taxonomy with actual chapters: "functions to improve behaviour" - policy, value fnc, q-fnc; TD: qlearning, sarsa; exploration vs exploitation: all strategies

ML chapter with: supervised, unsupervised, RL libs vorstellen und dann sagen für openai entschieden: openai baselines, stable baselines, openai spinning up, RLlib hyperparams chapter if more pages needed: Neural Network basics: neurons, weights, bias, batch, normalization, regularization, activation function, layers, etc. + deep + recurrent, cnn, etc. feed forward, backpropagation, bellman equation, TD(Qlearning, SARSA) and Monte-Carlo (thorough) boltzmann exploration, (more policy strategies? can add actuall algorithms to fill up space more/similar DRL algorithms, e.g. explicit TRPO, DDPG, A2C, A3C, Rainbow, Genetic Algorithms? ONE-STEP, MULTI-STEP add equation for everything summary for each chapter? regression, classification svm, trees, auto-encoder? was first chapter, review this all

In this chapter, the fundamentals of Reinforcement Learning (RL) are introduced. First, the basics of RL is explained, such as the definition of key terms and concepts, and notations, if any. Followed by basic RL-algorithms and finally, depp algorithms.. rewrite when done This is necessary to understand the deep algorithms, that are used later on in the experiments in chapter 5, namely, Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Proximal Policy Optimization (PPO) and REINFORCE.

2.1. Reinforcement Learning

2.1.1. Background

Reinforcement Learning is one of the three main paradigms in *Machine Learning*. These paradigms use algorithms that learn from experience to enhance the performance in their respective tasks (Mohri, Rostamizadeh, & Talwalkar, 2018). The remaining paradigms are *Supervised Learning* and *Unsupervised Learning*. In the former paradigm, the *learner* learns from training data, that is *labeled*. This means that the training data has been augmented with additional information, such as output labels of the correct *class* in the case of a classification problem, i.e. in a training set consisting of various animal images, the output labels would be the name of the animal, for each image. The learner learns to association the training data with their respective labels and applies the knowledge to unseen data, to map them to the correct labels. The latter paradigm also uses predefined training data, however, it does not use labels. Thus, it is not clear what to look for and there is also no feedback, since there are no labels associated with the training data. However, Unsupervised Learning can be used to identify patterns in unstructured data, e.g. clustering (Alzubi, Nayyar, & Kumar, 2018).

In Reinforcement Learning the learner, or rather *agent* in that context, learns from a *reward*, that is acquired by interacting with an *environment*, by trial-and-error. The learner is observing a *state* in an environment, at a specific time, and is performing an *action*, based on the information it has. Finally, the agent obtains a reward for its action and the environment transitions into its next state. This interaction is known as the agent-environment interaction and is illustrated in Figure 2.1.

The aim of the agent is to maximize the expected cumulative reward from the environment. This is done, by continuously interacting with the environment in an endless loop, until a termination condition is reached. A single transition in the loop is usually called an *episode*. The agent does not know in advance, what rewards it can expect for its action in a given state. It has to learn those by trial-and-error, thus exploring the environment and finding out which actions lead to the highest rewards. Based on the environment, an action might not only influence the immediate reward, but distant rewards as well, due to state transitions after each episode. This process can be described as a finite Markov Decision Process (MDP) (Sutton, 2018).

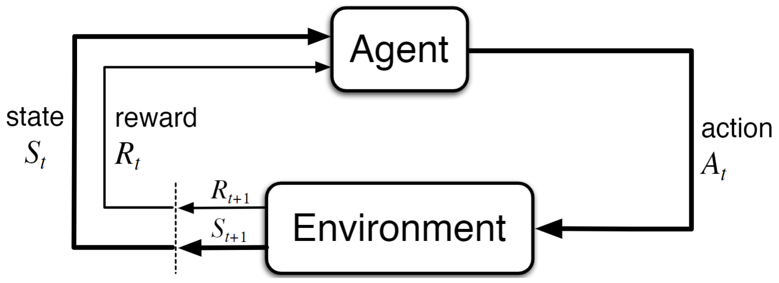


Figure 2.1.: Agent-environment interaction. The agent performs an action A_t in state S_t of an environment. The environment transitions into the successor state S_{t+1} at the end of the episode and the agent received a reward R_{t+1} for his action and the new state. This procedure is repeated until a termination condition is reached. Source: Reprinted from (Sutton, 2018, p.48)

Markov Decision Process A MDP can be formalized as a tuple $\langle\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\gamma\rangle$, where $\mathcal{S}$ and $\mathcal{A}$ are finite sets of states and actions, respectively. $\mathcal{P}$ is a state transition probability matrix, where the probabilities for all states s with action a to all successor state s' is stored. $\mathcal{R}$ is a function to calculate the expected reward for the agent its action. Both formalized in 2.1 and 2.2 respectively. The discount factor $\gamma \in [0,1]$ is used to weight an immediate reward against a distant reward in the reward calculation (Silver, 2020a).

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \tag{2.1}$$

$$\mathcal{R}_{s}^{a} = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.2}$$

The discounted reward is calculated for each timestep in an episode. From timestep t onwards, the rewards are added to the *return* G_t. The return is formalized by:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

Partially Obeservable Markov Decision Process A Partially Observable MDP (POMDP) is, as the name suggests, a MDP that is only partially observable. This results in the addition of another set and a function. The set $\mathcal{O}$ is representing a finite set of observations. These observations are fully visible, whilst the states $\mathcal{S}$ are hidden. The function $\mathcal{Z}$ calculates the probability of observing o, after taking action a and landing in state s', formalized by 2.4. Finally, a POMDP can be formalized as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma \rangle$.

$$\mathcal{Z}_{s'o}^{a} = \mathbb{P}[O_{t+1} = o | S_{t+1} = s', A_t = a] \tag{2.4}$$

2.1.2. Taxonomy

Figure2.2 illustrates the categorizations of RL methods. In this section various terminologies, regarding the illustrated categories, will be explained.

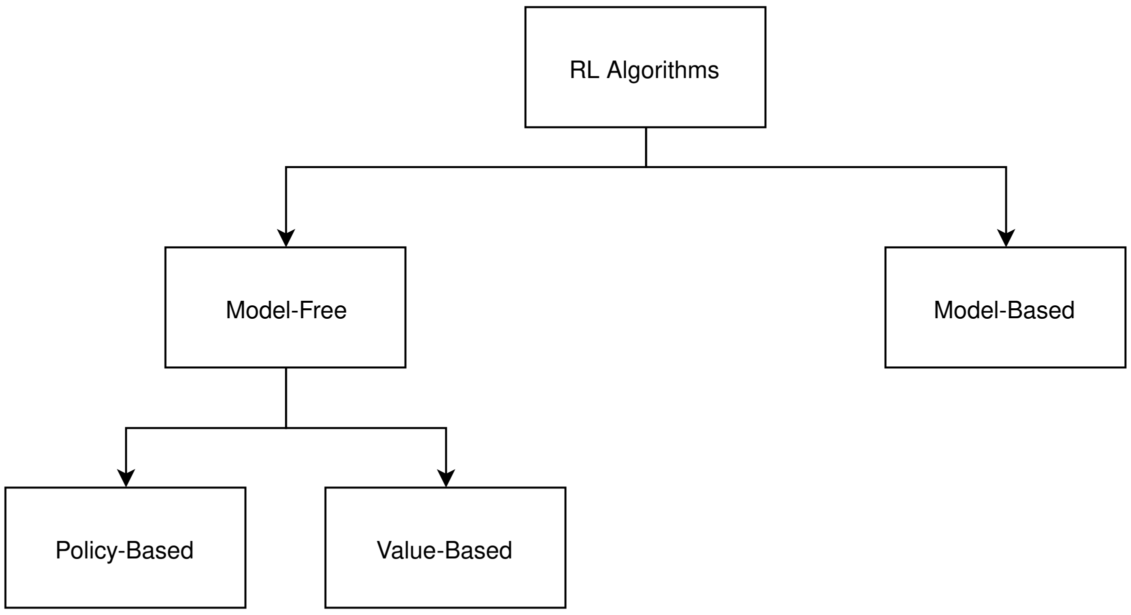


Figure 2.2.: Taxonomy of Reinforcement Learning algorithms. Source: Adapted from (Akanksha et al., 2021)

Model-based Model-based methods require a model of the environment, hence, the agent has information regarding, how the environment will behave in various states and transitions. This information can be used to do *planning*, i.e. predicting various situations in the future, without actually interacting with the environment. This is possible, due to the known state transition probability matrix, see MDP, (Sutton, 2018).

Model-free The agent in model-free methods does not have a model of the environment. Therefore, by trial-and-error, the agent has to do *learning*, i.e. learning which action maps to which state and which reward (Sutton, 2018).

Policy-based In policy-based methods, a policy calculates the most optimal action in a given state. Thus, a policy π can be seen as the distribution of a given state to all possible actions (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017):

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{2.5}$$

Value-based The value-based methods are using a value function to determine the benefit of being in a given state, i.e. the expected return for being in a given state is estimated. Starting in state s and following policy π afterwards, the estimation of the discounted return G_t (Arulkumaran et al., 2017). The state-value function can be formalized as following:

$$\mathrm{v}_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.6}$$

Similarly, the action-value function, or widelier known as the *Q-function*, can also be used to calculate the expected return. The Q-function is the expected return when starting in state s, taking action a and following policy π afterwards (Arulkumaran et al., 2017). Formalized as following:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.7}$$

Actor-critic Actor-critic methods are a combination of policy- and value-based methods. The policy is refered to as the *actor* and the value function as the *critic*. During training, the actor is receiving feedback from the cirtic on its performance, which it is using to improve its learning. In these methods, the critic is used as the baseline for the error calculation (Arulkumaran et al., 2017), see 2.1.3.

On-policy In on-policy methods, the policy itself, that is making the decision for the next action to take, is either evaluated, or improved (Sutton, 2018).

Off-policy Contrary to on-policy methods, off-policy methods do not evaluate, or improve, the policy, that is making the decision for the next action, but rather an additional policy, that is not used for action selection (Sutton, 2018).

Exploration vs Exploitation In RL tasks, it is not a trivial to determine when an agent should focus on exploring the environment to possibly find eventually better actions and thus gain better rewards, or exploit the best known actions to maximize the return. Policies implement various strategies to tackle this trade-off. For instance, an ϵ-greedy policy will select a random action with probability ϵ and will select the optimal action with probability $1-\epsilon$. Initially, the probability of exploring the environment with an ϵ-greedy policy is high, but it diminishes over time as more of the environment is explored. Another approach is a greedy policy, which will always select the action with the highest expected return (Sutton, 2018).

2.1.3. Q-Learning

Q-Learning is a value-based off-policy method that belongs to the family of Temporal Difference (TD) methods. These are basically model-free methods, that learn from raw experience and update their values based on estimates, rather than exact values. The Q-Learning method is defined by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[\underbrace{R_{t+1} + \gamma \max_{a'} Q(s',a')}_{\text{TD target}} - Q(s,a) \Big] \tag{2.8}$$

In this method, the Q-function directly approximates the optimal action-value function, without relying on the policy, that is being followed. Here, α, is the *learning rate*, which generally determines how much of the newly aquired information should be incooperated into the existing knowledgebase of a model. Essentially, this hyperparameter affects how fast an agent is learning. The equation within the brackets is a variation of the TD *error*, which calculates the sum of the reward for the state transition R_{t+1}, the estimate of the current state $Q(s, a)$ and the optimal estimate of the successor state $\gamma \max_a Q(s', a')$. Calculating the difference between the estimate of a current and successor state is called bootstrapping (Sutton, 2018).

2.2. Deep Reinforcement Learning

2.2.1. Deep Q-Network

Maybe better introduction The DQN by Mnih et al. (2015) is based on the Q-Learning method, hence, it is a value-based off-policy method. It combines Reinforcement Learning with artificial neural networks. Initially, it was developed to play arcade games from the Atari 2600 console on an emulator, where the environment is the game itself, the state is the current frame of the game and an action was simply an action from all available actions of the game. DQN agents performed on human-like level or above in 29 of the 46 games, that were used for experiments (Mnih et al., 2015).

The deep convolutional neural network is processing images by the size of 210 x 160 pixels with a 128-color palette. In an effort to enhance the capabilities and performance of the network, images are pre-processed, meaning the dimensions are downscaled to a single one (grey-scale) and cropped, so only the relevant area of the image is captured. Resulting in images with sizings of 84 x 84 pixels. In addition to those steps, four successive images are stacked, to obtain clearer information about the current state of the environment. A reason for this approach is that, when looking at a single frame, it is not clear, in which direction objects are moving to, or coming from. Hence, the decision to pre-process the data to 84 x 84 x 4 dimension input. The resulting input is then fed into the architecture depicted in figure 2.3.

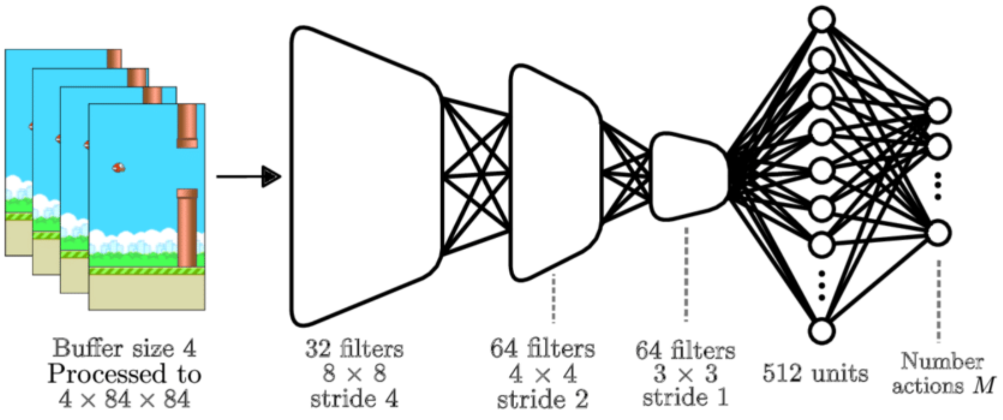


Figure 2.3.: DQN Architecture. The input data consists of four captured observations from the environment, that have been pre-processed into 84 x 84 dimensions. The input is successively passed into three hidden convolution layers, to extract and learn features from the observations. Each of the convolution layer is using rectified linear unit (ReLU) for the activation. Finally, the convoluted data is passed to the foruth hidden layer, which is fully-connected and consists of 512 rectifier units, followed by a fully-connected linear output layer, which has a mapping for each valid action in the environment. (Mnih et al., 2015). Source: Reprinted from (Spears et al., 2017)

DQN agents use techniques such as Experience replay, fixed Q-targets, to learn the best action to take. In Experience replay, the experience of the agent, at each time step, is stored in a memory replay. Mnih et al. (2015) stored tuples of $\langle \boldsymbol{S}_t, \boldsymbol{A}_t, \boldsymbol{R}_{t+1}, \boldsymbol{S}_{t+1} \rangle$, where $\boldsymbol{S}_t$ is a stack of four images, to the memory in the Atari experiments. These experiences are collected at each timestep. Q-value updates are performed on a subset of the collected experience. The subset for the mini-batch training is selected random- and uniformly. The advantages of these approaches are, on the one hand, to increase the stability of the Q-network and, on the other hand, to increase the efficiency of the training, because correlation of successive experiences are removed (Sutton, 2018).

The other technique, fixed Q-targets, uses an additional neural network with fixed parameters/weights (Silver, 2020b). This neural network is called the target network and is initially a copy of the Q-network. However, during training, the weights of the target network are not updated, only those of the Q-network. They are frozen for most of the time and only updated with the Q-network weights after a certain number of training steps. Instead of using the Q-network as the target for the error calculation, this target network is used, hence the name.

$$\mathcal{L}_i(w_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}_i}\left[\left(\boldsymbol{r} + \gamma \max_{a'} \boldsymbol{Q}(s', a'; \theta_i^-) - \boldsymbol{Q}(s, a; \theta_i)\right)^2\right] \tag{2.9}$$

The error, or loss, function for the DQN can be seen in equation 2.9. Here, $\mathcal{D}_i$, is the memory replay of the network, that consists of the previously mentioned experience tuples. Similar to the Q-learning method, the difference between the target and the current state is calculated. The difference to the Q-learning method, however, is that both Q-functions are parameterized policies, represented by the notations θ. This implies that neural networks are utilized for estimating the Q-values. The dash in θ_i^- implies that the parameters, or weights, are fixed, i.e. the weights of this neural network are frozen and will not be updated. Freezing the weights of the target network results in an even more stable training of the Q-network.

Double Deep Q-Network Double Deep Q-Network is an extention of the DQN. It addresses the issue that DQNs tend to select overestimated Q-values. This is due to the fact that the *max* operator on the target Q-value is used for both, action selection and evaluation. Therefore, the selection of an action is decoupled from its evaluation (Van Hasselt, Guez, & Silver, 2016). This results in the following formalization of the target network:

$$\boldsymbol{Y}_t^{DoubleQ} = \boldsymbol{R}_{t+1} + \gamma \boldsymbol{Q}(s', \operatorname*{argmax}_a \boldsymbol{Q}(s', a; \theta_t); \theta_t') \tag{2.10}$$

Action selection is still due to θ_t, however, the evaluation of the values is handled by the second neural network θ_t'.

2.2.2. REINFORCE

Although Williams (1992) initially introduced REINFORCE as a class of methods, it now refers to a Monte-Carlo Policy Gradient method. It is, as the name implies, a policy-based method, that learns a parameterized policy. Monte Carlo methods, in the context of Reinforcement Learning, are methods that estimate the expected return by averaging the return from an entire episode. Consequently, parameterized policy-based methods are formalized by:

$$\pi_\theta(s, a) = \mathbb{P}[a|s, \theta] \tag{2.11}$$

Policy gradient methods are methods that use the gradient of the objective function, $\mathrm{J}(\theta)$ (similar to the loss function in DQN), to find the optimal policy. In order to do this, a local maximum is explored during the gradient ascend of the policy. Formalized by:

$$\Delta\theta = \alpha \nabla_\theta \mathrm{J}(\theta) \tag{2.12}$$

Here, α, is the variable for the step-size and $\nabla_\theta J(\theta_t)$ is the policy gradient (∇ is symbolizing a gradient). $\Delta\theta$ represents the change of θ, i.e. the updated weights

(Sutton, 2018).

Finally, the objective function in a policy gradient method can be formalized as following:

$$\nabla_\theta \mathrm{J}(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\; Q^{\pi_\theta}(s,a)] \tag{2.13}$$

Baseline Without altering the expectation, the variance can be decreased by subtracting a *baseline* from the policy gradient. Subsequently, the state-value function is used as a baseline and subtracted from the Q-function, resulting in the advantage function $A^{\mathrm{w}}(s,a)$. A policy gradient with baseline is formalized by:

$$\nabla_\theta \mathrm{J}(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, A^{\pi_\theta}(s,a)] \tag{2.14}$$

2.2.3. Proximal Policy Optimization

PPO by Schulman, Wolski, Dhariwal, Radford, and Klimov (2017) is another policy-based method. It is similar to the Trust Region Policy Optimization (TRPO) method by Schulman, Levine, Abbeel, Jordan, and Moritz (2015). In TRPO, training stability is enhanced by constraining the extent to which a policy can be subjected to change at one step. The so-called *trust region* prevents performance collapse, which may happen during policy updates with large step size updates. It is determined by calculating the Kullback-Leibler (KL) divergence between old and new policies at each iteration of a policy update. The KL divergence is a metric that shows how two probability distributions differ from one another . In TRPO the objective function is usually referred to as *surrogate objective* (Schulman et al., 2015).

PPO has the same aim as TRPO, maximizing the scope of the policy update in a single step without compromising the performance. Unlike TRPO, PPO uses less complex methods to achieve that goal, which makes it simpler to implement and easier to tune. There are two variations: PPO-Clip and PPO-Penalty.

PPO-Clip In the PPO-Clip variant, the objective function is clipped and the KL divergence is not longer used. This is formalized by following equation:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\Big[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big] \tag{2.15}$$

Here, $r_t(\theta)$, is the probability ratio between the old and new policy and $\hat{A}_t$ is an estimator at timestep t of the advantage function. The hyperparameter ϵ controls how far the new policy can deviate from the old one, e.g. 0.2. By clipping the probability ratio and additionally applying the *min*-operator to the original and clipped ratio, updates to the policy with large steps are omitted.

PPO-Penalty The PPO-Penalty variant uses a penalty on the KL divergence. Each policy update iteration, this penalty coefficient is adapted to meet a target value. However, during the experiments by Schulman et al. (2017), this variant performed worse than the PPO-Clip variant and is only mentioned for completion.

3. Games

can extend both by including historical info In this chapter, the board games, Mastermind and Battleship, used for the experiments in chapter 5 will be introduced. The objective, rules and gameplay will be discussed. maybe explain why these two games were chosen? and/or why strategic board games?

3.1. Mastermind

Mastermind is a 2-player board game. Both players have different tasks and roles, namely codebreaker and codemaker. Like the names imply, the task of one player is to make a code and the task of the other player is to break it. In this game, the codemaker generates the code from a pool of various coloured pins. The order of the colours matter. The game is usually played on a 10 x 4 board and 6 colours. The codemaker has an additional hidden row to hide his code. In each column, there is a hole for the codebreaker to put in a pin for his guess attempt. Additionally, in each row there are four smaller holes, which the codemaker uses to give feedback. The codemaker will use black/red to indicate colour and position are correct, white for correct colour, but wrong position, or leave it empty, if the colour is not available in the code (*How to play Mastermind | Official Rules | UltraBoardGames*, n.d.). A modern variation of the game board can be seen in figure 3.1.

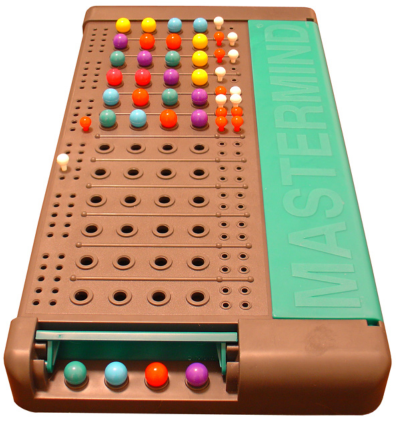


Figure 3.1.: Mastermind game board with 12 rows and 6 colours. Additionally, a hidden row for the code, at the bottom. Source: Reprinted from (Wikipedia contributors, 2022b)

For the experiments, a game board of size 10 x 4 and 6 colours were used. verify

3.2. Battleship

Battleship is another classic 2-player board game. The objective is to sink all enemy battleships. At the start of a game, each player has a pool of battleships with mixed lengths and has to place them, either horizontally, or vertically, on their own 10 x 10 board. The boards are kept hidden from each other. Typically, the length of the battleships ranges from two to five. These numbers were also used in the experiments verify. gehört das überhaupt hier rein? kann man doppelt machen, aber vllt hat das hier nichts zu suchen. Ships are allowed to touch, but not to overlap. Alternately, players call out which coordinate they want to attack. The opponent responds with either *hit*, or *miss*, when the attacker hit a battleship, or not, respectively.

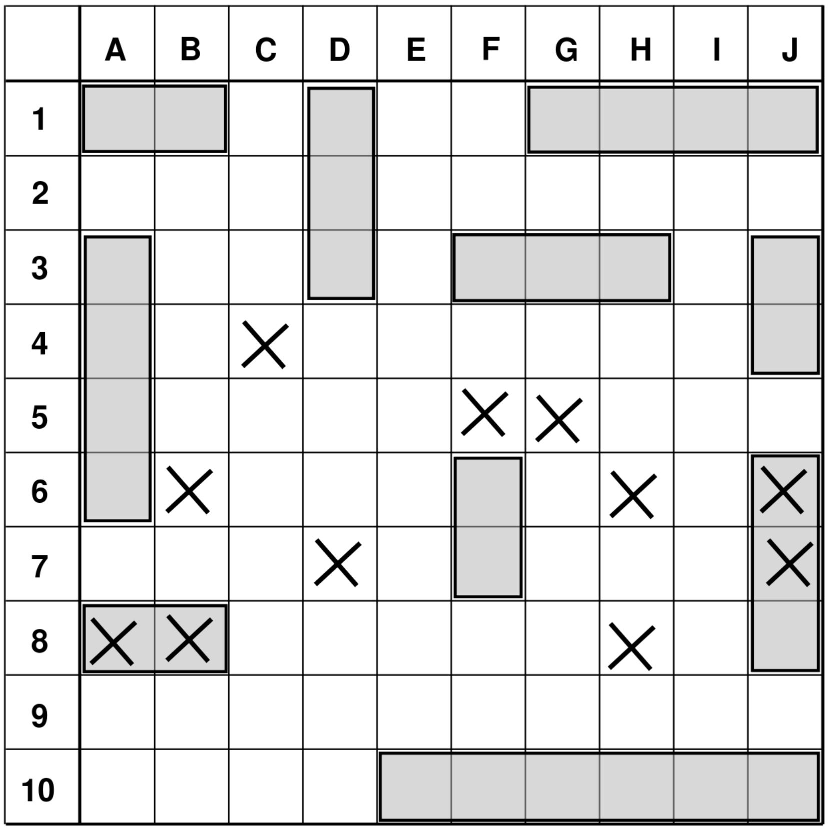


Figure 3.2.: Ongoing battleship game on a 10 x 10 board. The grey boxes represent the battleships of varrying sizes, the white boxes water and the cross marks indicate either a hit, or a miss. Source: Reprinted from (Wikipedia contributors, 2022a)

4. Implementation

say code was written in general fashion with actor learner API, so environments can be plugged in and out

can explain classes/files in appendix?

pictures of the NN of each algorithm

This chapter will discuss, how the games and the agents were implemented. Additionally, required steps, to reproduce the setup will be provided.

4.1. Development

Python was used as the programming language for the implementation of the agents, as well as environments. Furthermore, for the implementations of the agents *jupyter* was used. A Linux operating system is required, due to using *Reverb*, by Cassirer et al. (2021), for the experience replay. The agents were implemented with the deep RL library *TF-Agents* by Guadarrama et al. (2018). The library offers, among other things, implementations of various RL algorithms. It is easy to use and provides many code examples, alongside the extensive documentation. Full setup instructions are listed in appendix A.

4.2. Environment

rewrite first sentences One of the requirements is creating an environment. As previously discussed, in RL the agent is interacting with an environment, to aquire a reward for its action and transition to the next state of the environment. In TF-Agents, all environments must implement certain features to run. This includes:

1. Specifications regarding the format and size of possible actions.
2. Specifications regarding the format and size of states.
3. Steps to be executed after an episode is over.
4. Steps to be executed during each step iteration.
5. (Optionally) information on how the environment should be presented. is optional?

The details regarding the actions and states are used for the calculation of neural network layers and neurons check if true. The steps at the end of an episode, usually include resetting counters and variables to their initial value, such as, training step counter, current state of the environment and environment specific things. For instance, in Mastermind, a new code will be generated and previous guesses deleted. In Battleships, the ships will be replaced into different starting positions and the counters for hit and miss are reset. Steps after each action may include action pre-processing and reward calculation.

4.2.1. Mastermind

The features for the Mastermind environment are as following:

1. The action-space consists of 6^4 actions. This number reflects all variants of the code and is composed of the total number of colours and the number of colours to be guessed, respectively. Using a single digit instead of an array of integers reduces complexity, accordingly, training is faster.
2. In the observation specifications, two values are captured: the current state of the environment and all valid actions for action-masking, see section 4.5. The current state is reflected by a two dimensional array with four elements (number of colours to be guessed). The boundaries of the values are according to the number of total colours. The other value, all valid actions for the current state of the environment, consists of an boolean array with 6^4 elements, each of which corresponding to the validity of the action.
3. At the end of each episode, a new code is generated, valid actions and all states, this includes the current and accumulated states over an episode, are resetting. The end of an episode is reached, when either the agent breaks the code, or 10 rounds have passed.
4. During each step iteration, the action is transformed, see section 4.4, feedback, reward and valid actions are calculated.
5. The environment is shown in a terminal, along with all guesses and feedback.

4.2.2. Battleship

update, based on what the final version is Battleship features have been implemented as following:

1. The action-space has 10×10 actions, which reflects each cell of the board.
2. The observation-space covers the 10×10 gaming board, which is encoded in an integer array of the same size. Each cell has a value for its current status: unknown, hit, miss.

3. New ships are generated and placed after the end of each episode. Additionally, counters for hits and misses are reset, as well as the accumulated states.
4. At each step iteration, the action is transformed, see section 4.4, the status of the targeted cell is updated. Moreover, reward are calculated and accumulated states are updated.
5. Again, the environment is shown in a terminal. Each cell its status is shown (water, hit, miss).

4.3. Policy

what is stochastic (actor policy)? Policies can have different tasks, which depend on the algorithm, and the number may also vary. For most of the agents, they are configurable. DQN uses per default an epsilon greedy policy for collecting data and a greedy policy for evaluating it. The REINFORCE agent, using the baseline rather than the monte-carlo method, uses a greedy policy for its evaluation and a stochastic actor policy for its data collection. Lastly, the PPO agent uses.

4.4. Pre-Processing

In addition to the pre-processing steps taken by the library, further pre-processing steps were performed. Regarding Mastermind, this involves transforming the actions, due to the specifications of the actions, as well as observations. The action its value is transformed into an array of single digit integers, each of which corresponds to a distinct color of the code, rather than an index for a colour combination. In Battleship, the action is also transformed. Again, an integer, that acts as an index, is used for the input. The input is decomposed into two single digits; the x and y coordinates of the cell to be targeted.

4.5. Action-masking

It is not uncommon, that actions available in one state of the environment, cannot be performed in another one. In a grid-based game, for instance, there may be walls or obstacles that the player cannot pass, as a result, an action, that attempts to walk through that obstacle is pointless. However, the same action may work fine in another state. One common approach of tackling these type of issues is to give the agent a penalty for invalid actions. A penalty includes, giving the agent a negative reward for that action in that particular case (Huang & Ontañón, 2020).

Another approach includes masking invalid actions. Action-masking is performed during action selection for the current state of the environment. Before calculating the probability of each action, the logits of invalid actions are set to a high negative value, e.g. -1×10^8. As a result, the likelihood of taking such action under those circumstance will be virtually 0% (Huang & Ontañón, 2020). This approach was used for both games. In Mastermind, actions were masked according to feedback, i.e. subsequent actions must have at least same number of correctly guessed colours and in Battleship, actions were masked based on hits and misses, i.e. fields, that have been previously explored, are not explorable for the remaining episode.

5. Experiments

This chapter presents how the experiments were conducted. parameters? check what is mentioned in earlier chapters and include those 50k steps

Losses in 4 Bilder neben und untereinander. Die Eval ineinnander und Training metrics weglassen.

5.1. Mastermind

For the Mastermind environment, the size of the action space covers 6^4 units, each of which covers a possible colour combination, and the observation space is covered by 2×4 observations, one row for the guess by the codebreaker and the other for feedback by the codemaker. Additionally, the observation includes another list, which covers all valid actions for the current state of the environment. Based on the feedback, this list will be adjusted at each iteration and will only retain actions that score equally to, or better, than the current action. This ensures, that the agent is not making *worse* guesses and takes the feedback into consideration for its next move. The reward is simple, if the agent was able to break the code, it receives a reward of *1*, if not, the agent receives a reward of *-1*. An episode ends after 10 steps.

5.1.1. Results

Figure 5.1 shows the TD error for 50.000 training steps. It is visible that the DQN agent is already minimizing its loss in the very early stages of its training and the loss is fluctuating around 0.55, with 0.08 being the lowest value and 0.88 the highest. However, due to its extreme lowness, it might be an outlier and 0.23 could be considered the lowest value. All three extremes appear before passing 10.000 training steps, so it might be sufficient to run only those number of training steps, or less, to achieve similar results.

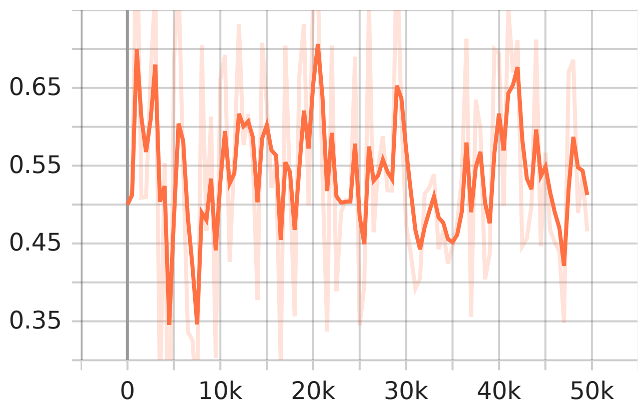
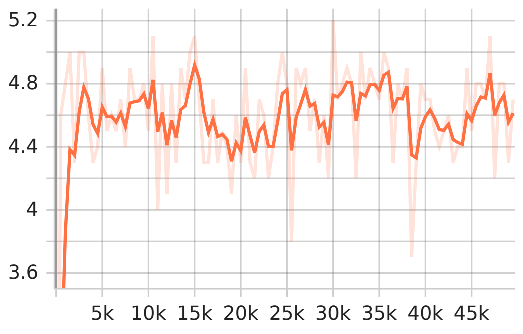
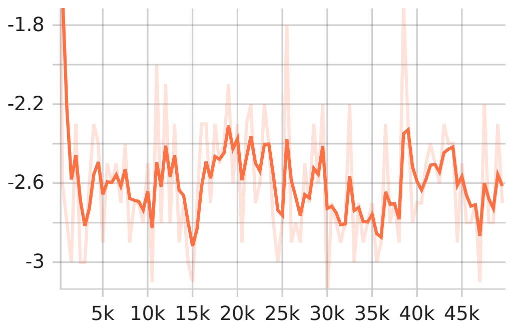


Figure 5.1.: Loss of the DQN agent in Mastermind. The x-axis displays the number of training steps and the y-axis the loss value.

The training metrics of the agent suggests the previous observation as well. The values for the average return and length of an episode are each marginally close, the return between around -2 and -3, and the episode number between around 4 and 5, as seen in figure 5.2. The average episode length, as well as average return, are fluctuating between those values; with no significant increase, or decrease, of performance regarding training duration.



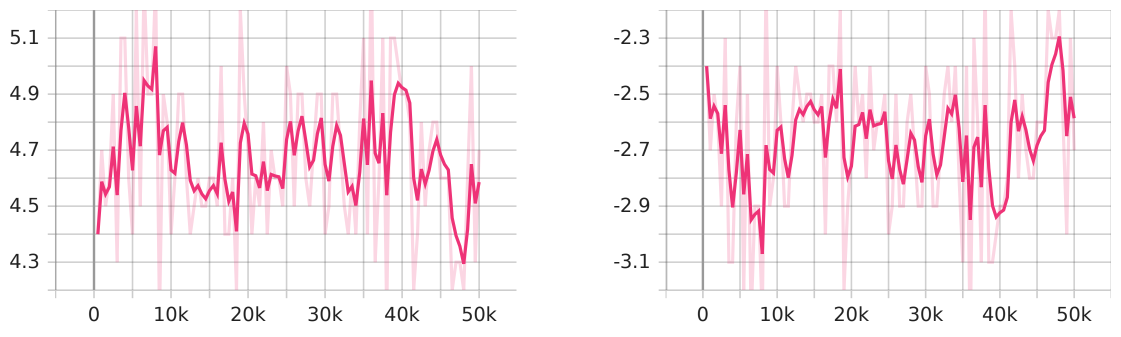
(a) Average episode length



(b) Average return

Figure 5.2.: Training metrics DQN. In both diagrams the x-axis displays the number of evaluation steps. The y-axis displays the average episode number in (a) and the average return in (b).

The evaluation of the agent in figure 5.3 confirms the observation that 10.000 training steps would be sufficient. The metrics show similar value to the training. Again, they fluctuating within the same range but generally, over time, no significant increase, or decrease, in performance.



(a) Average episode length (b) Average return

Figure 5.3.: Evaluation metrics DQN. The labels of the axes are the same as in figure 5.2.

insert combined training and eval metrics Figure X shows the training and evaluation metrics of all four agents combined. All agents show quite similar results, as seen in the figure. The reason for this result is that all agents mask invalid actions with the same method, i.e. actions that do not result in at least the same number of correctly guessed color combinations are not executed. Thus, resulting in breaking the code with each agent on average with **4.6** guesses. Complementary, the individual results of the remaining agents can be found in appendix B.

5.2. Battleship

The environment of the Battleship contains an action space of 10^2 actions and an observation space by the size of 10×10, resulting in 100 cells on the board. The values for the cells in the observation are bounded between minus one and one:

- -1 for a miss
- 0 for unkown, i.e. unexplored cell
- 1 for a hit

This is what the agent is receiving. There is another list that includes the complete placements of the ships, however, this is only used for rendering purposes and validation purposes. Reward?, Action masking?

6. Conclusion

References

Akanksha, E., Sehgal, J., Sharma, N., & Gulati, K. (2021, 04). Review on reinforcement learning, research evolution and scope of application. In (p. 1416-1423). doi: 10.1109/ICCMC51019.2021.9418283

Alzubi, J., Nayyar, A., & Kumar, A. (2018). Machine learning from theory to algorithms: an overview. In *Journal of physics: conference series* (Vol. 1142, p. 012012).

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017, nov). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26–38. Retrieved from `https://doi.org/10.1109%2Fmsp.2017.2743240` doi: 10.1109/msp.2017.2743240

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., . . . others (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Cassirer, A., Barth-Maron, G., Brevdo, E., Ramos, S., Boyd, T., Sottiaux, T., & Kroiss, M. (2021). *Reverb: A framework for experience replay.*

Guadarrama, S., Korattikara, A., Ramirez, O., Castro, P., Holly, E., Fishman, S., . . . Brevdo, E. (2018). *TF-Agents: A library for reinforcement learning in tensorflow.* `https://github.com/tensorflow/agents`. Retrieved from `https://github.com/tensorflow/agents` ((Online; accessed 25-June-2019))

How to play mastermind | official rules | ultraboardgames. (n.d.). `https://www.ultraboardgames.com/mastermind/game-rules.php`. ((Accessed on 11/21/2022))

Huang, S., & Ontañón, S. (2020). A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., . . . others (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of machine learning.*

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning* (pp. 1889–1897).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D. (2020a, March). *Lecture 2: Markov Decision Processes.* Retrieved from `https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf` ((Accessed on 11/14/2022))

Silver, D. (2020b, March). *Lecture 6: Value Function Approximation.* Retrieved from `https://www.davidsilver.uk/wp-content/uploads/2020/03/FA.pdf` ((Accessed on 11/14/2022))

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., . . . Hassabis, D. (2017, October). Mastering the game of go without human knowledge. *Nature*, *550*, 354–. Retrieved from `http://dx.doi.org/10.1038/nature24270`

Spears, T., Jacques, B., Howard, M., & Sederberg, P. (2017, 12). Scale-invariant temporal history (sith): optimal slicing of the past in an uncertain world.

Sutton, R. S. (2018). *Reinforcement learning: An introduction (adaptive computation and machine learning series).* A Bradford Book. Retrieved from `https://www.xarg.org/ref/a/0262039249/`

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 30).

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... others (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, *575*(7782), 350–354.

Wikipedia contributors. (2022a). *Battleship (game) — Wikipedia, the free encyclopedia.* Retrieved from `https://en.wikipedia.org/w/index.php?title=Battleship_(game)&oldid=1120935103` ([Online; accessed 21-November-2022])

Wikipedia contributors. (2022b). *Mastermind (board game) — Wikipedia, the free encyclopedia.* Retrieved from `https://en.wikipedia.org/w/index.php?title=Mastermind_(board_game)&oldid=1122862799` ([Online; accessed 21-November-2022])

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3), 229–256.

A. Development

The setup uses the libraries presented in section A.1. The tf-agents library has to be adjusted, to be able to use action-masking with PPO agents. The code for the value network is missing an argument to determine, whether to use a mask, or not. Therefore, a new argument has to be added. The file is located under `/usr/local/lib/python3.8/dist-packages/tf_agents/networks/value_network.py`. The argument `mask=None` has to be added to the function `call` for action-masking to work. An issue (#762) addressing this problem was created in the tf-agent repository on Github, but as of 29.11.2022 no reply.

The whole setup process can be done automatically with *Docker* on a Windows operation system. The scripts, A.2 and A.3, have to be saved in the same folder with their respective names. In addition to making the previously mentioned code modifications, all necessary libraries will be installed in a Linux container, by executing *start.bat*. Finally, the container will be started and will listen to multiple ports, so statistics can be presented via *Tensorboard* hosted on localhost.

A.1. Libraries

Following libraries and versions were used for the setup:

- Docker desktop: 4.11.1
- Python: 3.8.10
  - IPython: 8.3.0
  - ipykernel: 5.1.1
  - tf-agents: 0.13.0
  - dm-reverb: 0.8.0

A.2. start.bat

```
docker build -t rl/tf_agents_with_reverb:2.9.1 .
docker run --name tfagents-reverb -p 8888:8888 -p 6006-6009:6006-6009 rl/tf_
```

A.3. Dockerfile

```
FROM tensorflow/tensorflow:2.9.1-jupyter

RUN pip install tf-agents==0.13.0
RUN pip install dm-reverb==0.8.0

# https://github.com/tensorflow/agents/issues/762
RUN sed -i ’s/training=False/training=False, mask=None/g’ /usr/local/lib/python3

WORKDIR "/tf"
RUN rm -rf tensorflow-tutorials
COPY ./ /tf/
```

B. Experiments

B.1. DDQN

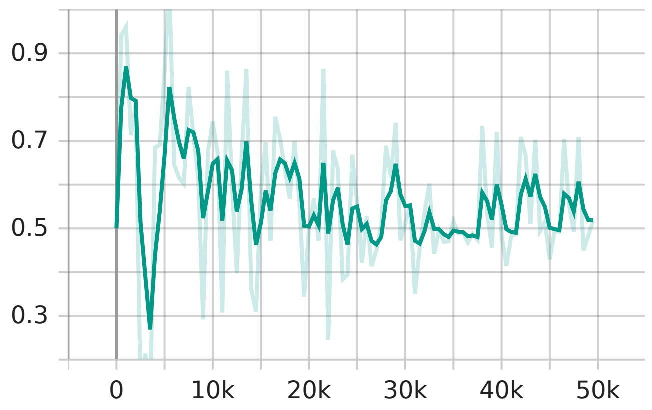
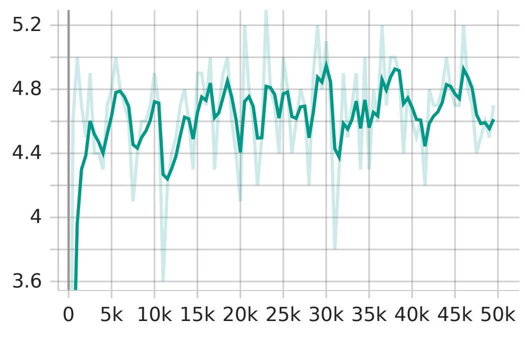
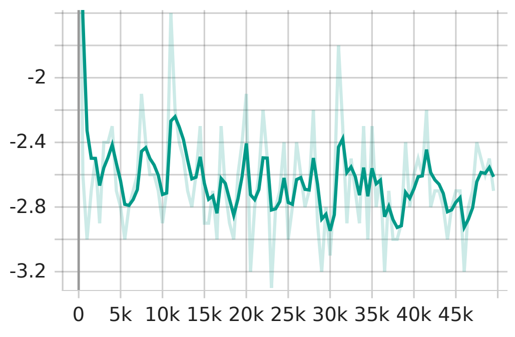


Figure B.1.: Loss of the DDQN agent in Mastermind. The x-axis displays the number of training steps and the y-axis the loss value.

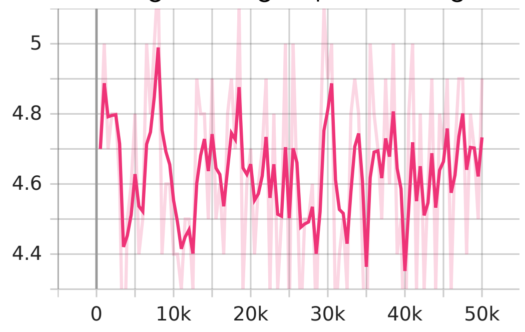
Initially, in contrast to the remaining values, the DDQN agent is having high extremes, 0.87 as its highest and 0.27 as its lowest value for the loss. However, after training for 10.000 steps it is starting to stabilize and not having that high spikes again, as seen in figure B.1.



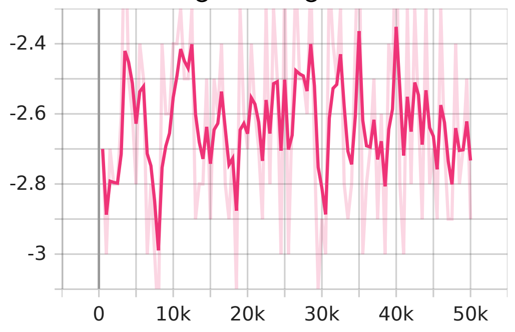
(a) Training: Average episode length



(b) Training: Average return



(c) Evaluation: Average episode length



(d) Evaluation: Average return

Figure B.2.: Metrics for the DDQN agent. In all diagrams the x-axis displays the number of training/evaluation steps. The y-axis displays the average episode number in (a) and (c), and the average return in (b) and (d).

Similarily, to the DQN (and other agents), the training and evaluation metrics, seen in figure B.2, show similar values in all aspects, due to action-masking.

B.2. REINFORCE

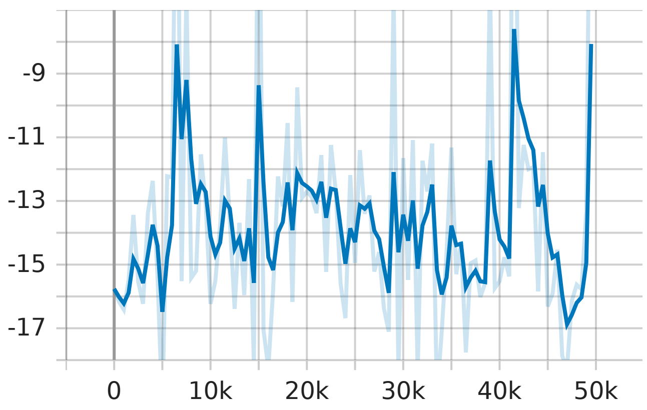
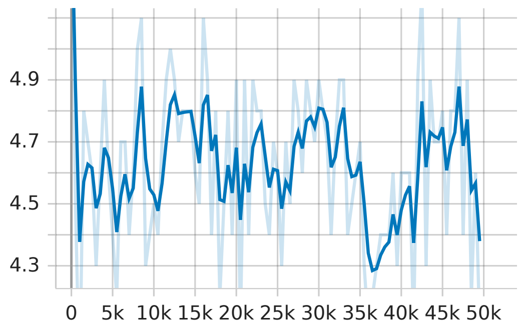
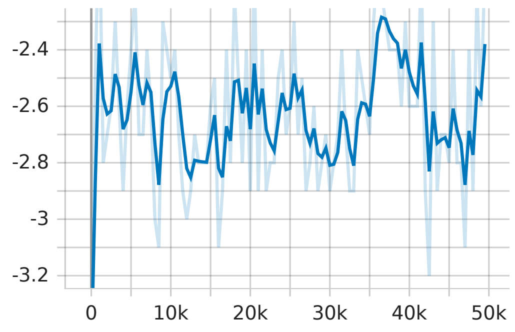


Figure B.3.: Policy gradient loss of the REINFORCE agent in Mastermind. The x-axis displays the number of training steps and the y-axis the loss value.

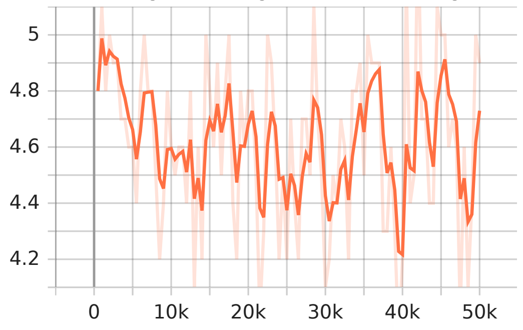
In contrast to the DQN and DDQN agent, the REINFORCE agent is calculating the loss of the policy gradient, hence the difference in numbers along the x-axis. Figure B.3 shows relatively high difference between the extremes, with -19.59 at 5.000 training steps being the lowest and 3.221 at 41.500 training steps the highest. However, most of the values are remaining at the same range between -11 and -16, regardless of training duration. Suggesting, that increasing training duration will not significantly improve performance.



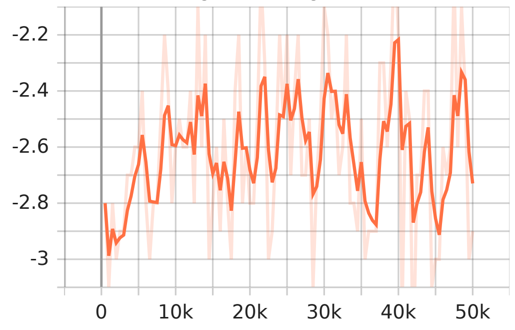
(a) Training: Average episode length



(b) Training: Average return



(c) Evaluation: Average episode length

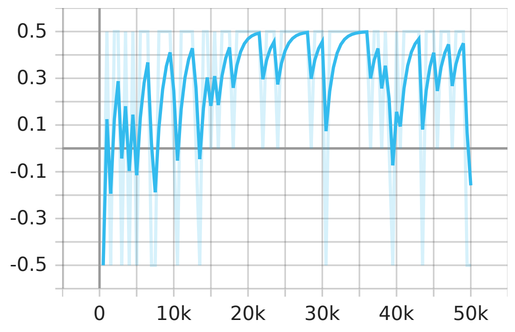


(d) Evaluation: Average return

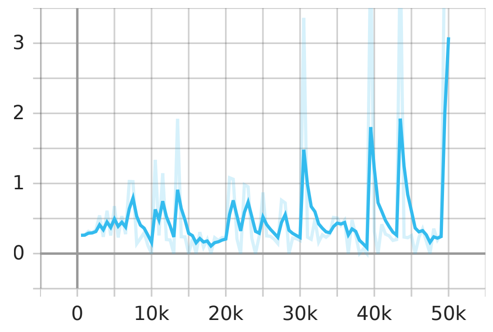
Figure B.4.: Metrics for the REINFORCE agent. In all diagrams the x-axis displays the number of training/evaluation steps. The y-axis displays the average episode number in (a) and (c), and the average return in (b) and (d).

The values of the training and evaluation metrics in figure B.4 indicate the same observation and are averaging the same metrics, as other agents.

B.3. PPO



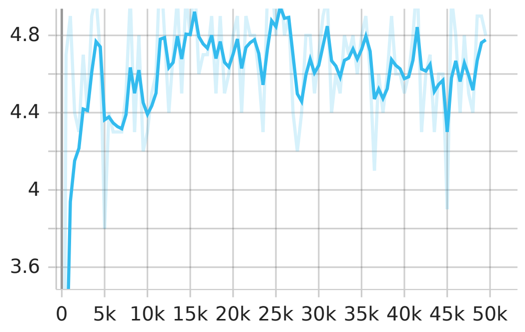
(a) Policy gradient loss



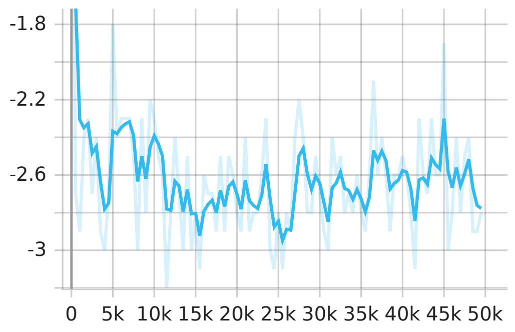
(b) Value estimation loss

Figure B.5.: Policy gradient loss (a) and value estimation loss (b) of the PPO agent in Mastermind. The x-axis displays the number of training steps and the y-axis the loss value.

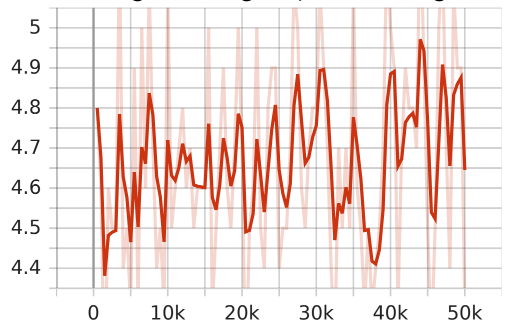
The PPO agent has an additional loss, the value estimation loss. The policy gradient loss is constantly fluctuating between -0.5 and 0.5, as seen in figure B.5. This is due to the *clip* variant of the PPO method. It is by nature restricted, how far the new policy can deviate from the old one. The other loss, is from the critic of the network. Initially, the loss is relatively low, but is having increased high spikes.



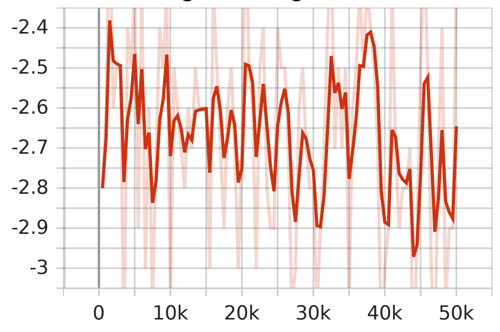
(a) Training: Average episode length



(b) Training: Average return



(c) Evaluation: Average episode length



(d) Evaluation: Average return

Figure B.6.: Metrics for the PPO agent. In all diagrams the x-axis displays the number of training/evaluation steps. The y-axis displays the average episode number in (a) and (c), and the average return in (b) and (d).

Again, the metrics for training and evaluation show similar values as the other agents, as demonstrated in figure B.6.