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	January 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, 27 December 2022

_____

BEYTULLAH INCE

Contents

List of Figures

List of Tables

List of Abbreviations

A2C Advantage Actor Critic

A3C Asynchronous Advantage Actor-Critic

AI Artificial Intelligence

DDPG Deep Deterministic Policy Gradient

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

SAC Soft Actor-Critic

SB3 Stable Baselines3

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.

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 complex domains. Specifically, methods based on the Machine Learning paradigm Reinforcement Learning. In Reinforcement learning, an Artificial Intelligence system, typically called agent, is learning to do a task via trial-and-error. The agent is performing different actions in an environment and is getting a signal, which tells, how it performed. For example, considering the Atari videogame Ms. Pac-Man, a game where the player has to collect pellets on a grid, that has obstacles and the player has to avoid enemies, as they will eat the player. In a Reinforcement Learning setting, the grid, player, pellets, enemies and everything else inside the game is considered as the environment and the player the agent. Initially, the agent has no experience, on how to play the game. It will try out various actions, such as moving left, right, up or down. Based on the outcome if this action, the agent will be reward. The reward can either be positively, to encourage the agent to do that action in that situation again, or negatively, to prevent taking an action in the same scenario. Examples for that could be, a positive reward for collecting a pellet and a negative reward for not avoiding the enemies. The agent will gradually improve its gameplay, by remembering the actions it took, at certain scenarios, alongside the reward signal it received. It can learn from its past experiences and improve its gameplay (Sutton, 2018).

A decade ago, January 01, 2013, a Reinforcement Learning method, named Deep Q-Network, was published (Mnih et al., 2013). This method was able to outperform back-then state-of-the-art methods in 6 out of 7 Atari 2600 games and human experts in 3 of those games. Three years later, February 25, 2015, another version of Deep Q-Network was released. This method was able to outperform all other, back then state-of-the-art, methods and perform on a level comparable of a human professional in 49 games, without altering its configurations. These results were achieved by combining a variation of the Reinforcement Learning method, Q-Learning, with a special type of an Artificial Neural Network. The special type of Artificial Neural Network is called Convolutional Neural Network. The Convolutional Neural Network is processing the input data, which is the current state of the game represented as raw pixels and the Q-Learning variant will select an action based on this information and past experience (Mnih et al., 2015). In following years, variations of the DQN were published, that tackled short-comings,

such as overoptimistically estimating the current state of an environment, resulting in sub-optimal choices for the actions (Sewak, 2019).

Methods like Deep Q-Network are incorporated into Artificial Intelligence systems, alongside other concepts, such as encouraging different behaviours during training, setting domain-specific rules, or using additional human data for training, in order to perform on world-class level in various domains. One of these systems is *OpenAI Five*. It is a system developed in 2016 by OpenAI, an Artificial Intelligence research company. The system is playing the videogame *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, as the system was only trained playing those heroes. OpenAI Five was trained with various methods, such as playing against itself and other human players, such as amateur players from the development team, or online versus amateur players. OpenAI Five got better, by playing against amateur players, whose skill level were gradually increased. The level of those players was 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 in 2019(Berner et al., 2019).

Deepmind, an Artificial Intelligence subsidiary of Google, developed another Artificial Intelligence system that is performing at the highest levels in its respective game. The name of this system is *AlphaStar*, was release in 2019 and it is 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).

The achievements of these systems are not only limited to video games. They can also play class complex strategic board games, such as Chess. The Artificial Intelligence system *Deep Blue* was already in 1996 capable to play on world-class level. The development started in 1985 by Feng-hsiung Hsu and managed to win various computer chess tournaments in the upcoming years by researching for IBM. In 1996 it lost two out of six games against a world champion, but managed to win in a re-match in the following year, with three wins and one draw (Hsu, 2002).

Another strategic board game 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 developed a system named *AlphaGo* in 2016. 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).

The successor to AlphaGo is referenced as *AlphaZero* and was released in late December 2017. AlphaZero is able to play multiple classical board games, such as chess, shogi (Japanese version of chess) and Go. AlphaZero is a more generalized approach of an enhanced version of AlphaGo, which is named AlphaGo Zero. Hence, it is able to perform well in various games. It was able to win most matches against its predecessor AlphaGo Zero, resulting in a winning ratio of 61%. However, AlphaZero is mainly known for its performance in the game chess. Previously, a chess engine named *Stockfish* was known as the strongest chest playing software. But, *AlphaZero* was able to beat the engine on various occasions. In two of these occasions, a in 100-games match and 1000-games match were played. The results were one-sided. In the 100-game setting AlphaZero won 28 games and the remaining 72 ended in draws (Silver et al., 2018a). AlphaZero was able to win 155 games, lost 6 times and the remaining 886 games ended in draws, in the 1000-games setting (Silver et al., 2018b). In Shogi there is a computer similar to Stockfish, which is named *Elmo*. The matches against Elmo are once again one-sided and resulted in AlphaZero winning 91.2% of its matches against Elmo. Due to its general approach, AlphaZero does not require domain-specific human knowledge or data. It is using the same architecture of deep neural networks, general-purpose Reinforcement Learning algorithms and general-purpose tree search algorithms, to learn these games entirely by playing against itself (Silver et al., 2018b).

The most recent publication from Deepmind is yet another enhanced version of the previous releases and was released in 2020, four years after the initial release of AlphaGo. This system is named *MuZero*. Contrary to its initial predecessor, MuZero does not require human data, or domain knowledge. AlphaGo Zero and AlphaZero also did not require this information, however, they required knowing the rules of the games, that they were playing in. With MuZero, this is no longer required. The system learns the rules of the game itself, by playing it. It is able to plan strategies, that helps winning games in environments that are previously unknown. Additionally, to the games of the successor, Chess, Go and Shogi; MuZero is also able to play games of the Atari console. It is outperforming state-of-the-art methods in the Atari domain. It is able to do this task, due the feature, that allows MuZero to learn the most important dynamics of the environments and plan its moves based on that (Schrittwieser et al., 2020).

All of these recently published advanced systems have one thing in common: they are all based on Reinforcement Learning techniques. This thesis will conduct

experiments to test the eligibility of four state-of-the-art Reinforcement Learning methods from the Atari domain: Deep Q-Network, Double Deep Q-Network, REINFORCE and Proximal Policy Optimization. The tests will be conducted on two custom implementations of classic board games, with respect to single- and multiplayer games. The thesis will first cover the fundamentals of Reinforcement Learning. Important concepts and technical terms will be reviewed, as they are necessary in order to comprehend, how the Reinforcement Learning methods function. The objectives, rules and the gameplay of the games *Mastermind* and *Battleship* will be presented, followed by details regarding the implementation of the games and choices of libraries. Instructions for setting up the entire workspace will be provided. Finally, it is demonstrated how the tests were carried out and finished off with the results and a lookout.

2 Fundamentals

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 explanations of more advanced Reinforcement Learning concepts and terms. Afterwards a Reinforcement Learning method will be described and eventually state-of-the-art Deep Reinforcement Learning methods, such as Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Proximal Policy Optimization (PPO) and REINFORCE, will be explained.

2.1 Reinforcement Learning

2.1.1 Background

Reinforcement Learning is one of the three main paradigms in *Machine Learning*. This paradigm uses algorithms that learn from experience to enhance the performance in their respective tasks (Mohri, Rostamizadeh, & Talwalkar, 2018). The other 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 associate 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).

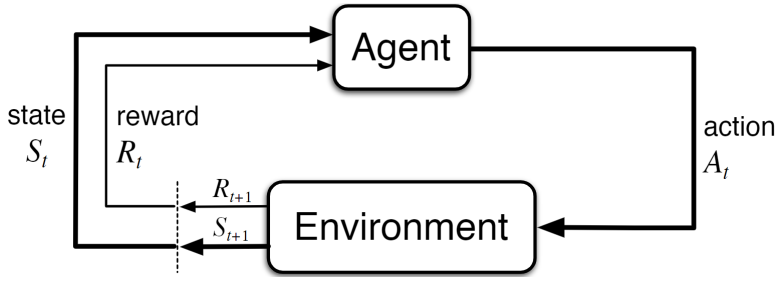


Figure 2.1: Agent-environment interaction. The agent performs an action $\boldsymbol{A_t}$ in state $\boldsymbol{S_t}$ of an environment. The environment transitions into the successor state $\boldsymbol{S_{t+1}}$ at the end of the episode and the agent received a reward $\boldsymbol{R_{t+1}}$ for his 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.

$$\mathcal{P}^{a}_{ss'} = \mathbb{P}[\boldsymbol{S}_{t+1} = s' | \boldsymbol{S}_t = s, \boldsymbol{A}_t = a] \tag{2.1}$$

$$\mathcal{R}^{a}_{s} = \mathbb{E}[\boldsymbol{R}_{t+1} | \boldsymbol{S}_t = s, \boldsymbol{A}_t = a] \tag{2.2}$$

The discounted reward is calculated for each timestep in an episode. From timestep t onwards, the rewards are added to the *return* G_t (Silver, 2020a).

The return is formalized by:

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

Partially Observable Markov Decision Process A Partially Observable MDP (POMDP) is, as the name suggests, an 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' (Silver, 2020a), formalized by 2.4.

Finally, a POMDP can be formalized as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma \rangle$.

$$\mathcal{Z}^a_{s'o} = \mathbb{P}[\boldsymbol{O}_{t+1} = o | \boldsymbol{S}_{t+1} = s', \boldsymbol{A}_t = a] \tag{2.4}$$

2.1.2 Taxonomy

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

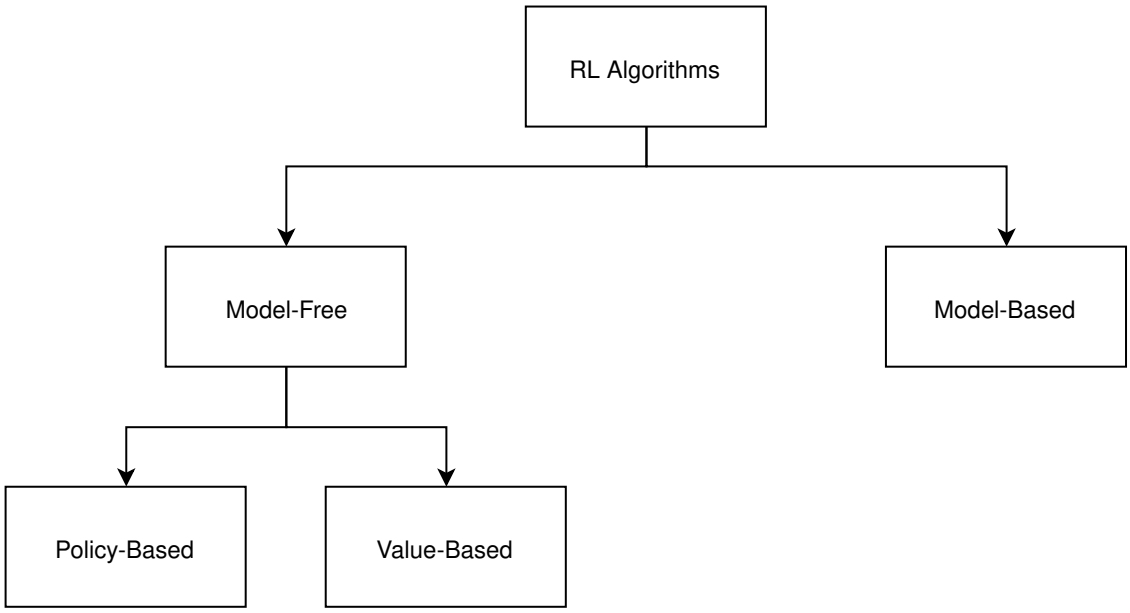


Figure 2.2: Taxonomy of Reinforcement Learning algorithms. Source: Adapted from (Akanksha et al., 2021)

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).

Formalized by:

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

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 widely 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 referred to as the *actor* and the value function as the *critic*. During training, the actor is receiving feedback from the critic 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 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:

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

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 acquired information should be incorporated 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 $\boldsymbol{R}_{t+1}$, the estimate of the current state $\boldsymbol{Q}(s,a)$ and the optimal estimate of the successor state $\gamma \max_a \boldsymbol{Q}(s',a')$. 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

The DQN by Mnih et al. (2013) value-based off-policy method based on the Q-Learning 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 sizing 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.

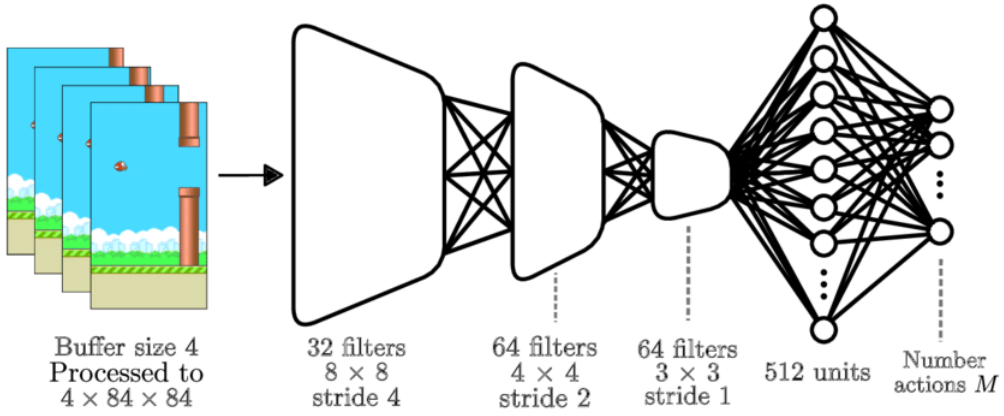


Figure 2.3: DQN Architecture. The input data consists of four captured observations from the environment, that have been pre-processed into 84 x 84 dimensions. The input is successively passed into three hidden convolution layers, to extract and learn features from the observations. Each of the convolution layer is using rectified linear unit (ReLU) for the activation. Finally, the convoluted data is passed to the fourth 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 $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$, where 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. 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 (Silver, 2020b).

The error function is formalized as:

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

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 (Silver, 2020b).

The full algorithm from Mnih et al. (2013) is presented in Algorithm 1.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $\mathcal{N}$
Initialize action-value function $\boldsymbol{Q}$ with random weights
for episode = 1, $\mathcal{M}$ **do**
  for t = 1, $\mathcal{T}$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a \boldsymbol{Q}^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \boldsymbol{Q}(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - \boldsymbol{Q}(\phi_j, a_j; \theta))^2$
  end for
end for

Double Deep Q-Network Double Deep Q-Network is an extension 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:

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

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:

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

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 $\nabla_\theta \boldsymbol{J}(\theta_t)$ 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:

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

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 $\boldsymbol{A}^{\mathrm{w}}(\boldsymbol{s}, \boldsymbol{a})$.

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).

TRPO 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).

The KL divergence in TRPO is formalized by:

$$\mathbb{E}_{s\sim p^{\pi_{\theta_{old}}}}[\boldsymbol{D_{KL}}(\pi_{\theta_{old}}(.|s)||\pi_\theta(.|s))] \leq \delta \tag{2.15}$$

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 no longer used. This is formalized by following equation:

$$\mathcal{L}^{CLIP}(\theta) = \hat{\mathbb{E}}_t\Big[\boldsymbol{min}(r_t(\theta)\hat{\boldsymbol{A}}_t, \boldsymbol{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{\boldsymbol{A}}_t)\Big] \tag{2.16}$$

Here, $r_t(\theta)$, is the probability ratio between the old and new policy and $\hat{\boldsymbol{A}}_t$ 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. The full algorithm provided by Achiam (2017) is presented in Algorithm 2.

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.

Algorithm 2 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for k = 0,1,2, ... **do**
Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
Compute policy update $\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \mathcal{L}_{\theta_k}^{\text{CLIP}}(\theta)$
by taking K steps of minibatch SGD (via Adam), where
$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \underset{\tau \sim \pi_k}{\mathrm{E}} \left[\sum_{t=0}^{T} \left[min(r_t(\theta)\hat{A}_t^{\pi_k}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k}) \right] \right]$
end for

3 Games

In this chapter, the board games, Mastermind and Battleship, used for the experiments in chapter 5 will be introduced. Rules and gameplay will be discussed. The objective when looking for suitable games was, that foremost, the games should be classic well-known board games. The implementation of the games should not be too complex and time consuming. Additionally, it was important that games were chosen based on the number of players required to play the game, so one single player and one multiplayer game. In this thesis, Mastermind is regarded as a single player game and Battleship as a multiplayer game, whereas in reality, each are played with two players.

3.1 Mastermind

Mastermind is a guessing type strategic board game. It is played by two players. The 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 colored pegs, where the order of the colors matter. Usually, the number of colors in the pool is 6 and the length of the code 4. The game is played in 10 rounds, where the codebreaker has to guess each round. The game board has a hidden row, which allows the codemaker to place the colored pegs, which form the code, inside. When the codemaker has decided on a code and place it inside the hidden row, the codebreaker starts to guess the code. Every turn, the codebreaker is placing a colour combination inside a row. The codemaker responds to the guess, by placing pegs next to the row with the guess. A black peg is placed, when a color is correct and placed and the right position and a white peg, for correct color, but wrong position. A feedback field is left empty, if a color was not in the code. The fields of the feedback do not correspond to a column in a row, hence, the codebreaker does not know, which color is placed correct, but rather, that *a* color is placed correct. Thus, the codebreaker has to make various guesses to include, or exclude, colors and simultaneously figuring out the correct spot (*How to play Mastermind | Official Rules | UltraBoardGames*, n.d.). A modern version of the game board can be seen in figure 3.1.

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

3.2 Battleship

Another classic 2-player board game is Battleship. Battleship is also a strategy game, where the players have to make guesses, to win a game. It can be either played on paper, or on a board, designed for its purpose. Both players have their own board, which is not visible for the opposing player. The players get battleships, which they can place on the board. The goal is to find all battleships of the enemy. Battleships can be placed vertically, or horizontally, but not diagonally. There are many variants of this game, therefore, the size of the ships and the number, of how many one player can place, varies. The number of ships a player can place is always equal to the opponent. Usually, the sizes of the ships range from two to five. These are popular options to place ships:

- Every player can place his ships once.
- Every player can place his ships once, except the 3-sized ship can be placed twice.
- Number of ships decreases with increasing length starting with 4, i.e. 4 x 2-sized, 3 x 3-sized, 2 x 4-sized and 1 x 5-sized battleships, which is used in figure 3.2

In this thesis, the placing rules from Hasbro (Wikipedia contributors, 2022a) will be adapted, where each of the following five ships are placed:

- Patrol Boat - size: 2
- Submarine - size: 3
- Destroyer - size: 3
- Battleship - size: 4
- Carrier - size: 5

The battleships are allowed to touch, but not to overlap. When each player placed his ships, they game will start and players will take alternately turns. Players will tell, which coordinate they want to attack and the opponent will say, whether the attack was a hit, or miss. Both players will mark their boards accordingly. By hitting the last slot of a battleship, the opponent must announce, that a ship, with its corresponding name, has been sunken. The game ends, if one of the players has no ships left. A board of an on-going game can be seen in figure 3.2.

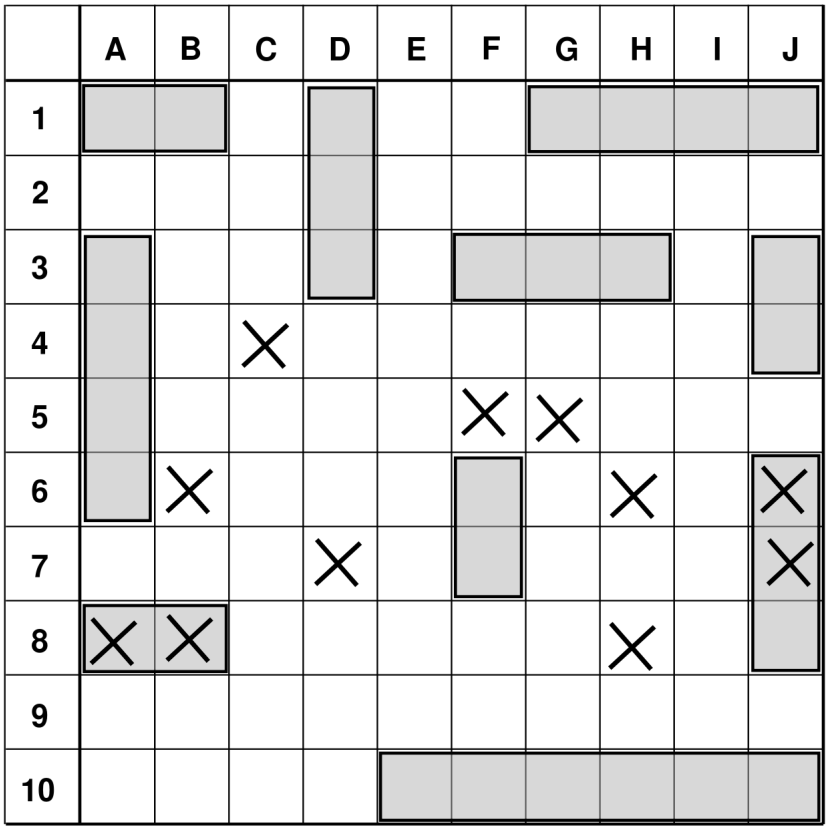


Figure 3.2: Ongoing battleship game on a 10 x 10 board. The grey boxes represent the battleships of varying sizes, the white boxes water and the cross marks indicate either a hit, or a miss. Source: Reprinted from (Wikipedia contributors, 2022b)

4 Implementation

In this chapter, various aspects regarding the implementation of the agents and game are discussed. It is explained, why several choices, such as choosing an RL library for handling agents, or implementation details for the environments, were made. The requirements and instructions on, how to install the complete setup, are provided. Design choices and implementation details for the games are explained, as well as, details regarding the agents from the RL library. Additionally, a concept used for action selection is presented.

4.1 Development

For the programming language python was chosen. Many frameworks and libraries regarding Machine Learning are provided in python, due to its ease of use. Hence, both, the implementation of the agents and the implementation of the environments, are written in python. Furthermore, for the implementations of the agents *jupyter notebook*, due to caching of variables. For the environment plain python was used, as it is only required for the agent to use. Linux has to be used for the operating system, because the library, which is used for the implementation of the Reinforcement Learning methods, is mainly using *Reverb* by Cassirer et al. (2021) to handle data collection in the experience replay. However, the setup can be run on a Windows machine as well, by using a *Docker* container. Instructions are provided in 4.1.2.

4.1.1 Library

Several popular and established libraries were considered for the experiments. Among them were: OpenAI Gym, Stable Baselines3, RLlib and TF-agents.

OpenAI Gym Gym by OpenAI provides a website with an overview of their API, environments and tutorials. The API covers core functionality, available formats for the action and observation spaces, wrappers for the environment and couple utility functions. Wrappers contain the ability to alter the environment, without modifying its code, e.g. a wrapper for preprocessing Atari environments according to arguments, flatten the observation, normalizing rewards, and more. The environments covered are based on Atari, MuJoCo, Toy Text, Classic Control, and Box2D, and are supplemented by several third-party environments. The overview

of all environments includes information such as the actions available in that environment, whether they are discrete or continuous, same applies to the observation. Additionally, the goal in that environment is described and if alternative versions of specific an environment is available and what its containing. A total of 2600 Atari games, including variations in an environment, are covered. MuJoCo is short for Multi-Joint dynamics with Contact. These type of environments are using a physical engine to emulate movement in multiple joints, such as robotics. The Toy Text environments are based on native python libraries, e.g. StringIO and are developed with simplicity and debugging RL methods in mind. Classical Control environments involve balancing and rotating (moving) objects. The last environment type is using a 2D physics engine to move objects to a certain goal/position. The tutorials cover the implementation of a custom environment and using wrappers. The GitHub repository contains code for all of the environments, except the Atari ones (Brockman et al., 2016).

RLlib Ray provides various libraries regarding scaling AI and python workloads. One library is RLlib and is handling Reinforcement Learning tasks. It supports custom environments and learning by prerecorded data. Again, environments can be imported from OpenAI gym. Various state-of-the-art methods are supported, such as A2C, Asynchronous Advantage Actor-Critic (A3C), a single player version of AlphaZero, DDPG, DQN, Rainbow and APEX-DQN (variations of DQN), PPO, SAC and many more. The library supports Discrete and Continuous action-spaces. An addition to this library, is the support of multiple agents in common environment, i.e. multiple agents are playing the same game at the same time. The underlying framework it is using, can be chosen, either PyTorch, or TensorFlow (Liang et al., 2018).

Stable Baselines3 The Stable Baslines3 (SB3) itself does not offer any environments, however, it is possible to use the environments from OpenAI gym and custom environments. SB3 offers various implementations of state-of-the-art RL methods and is using PyTorch, an open-source machine learning framework, its base. Among others, the implementations include Advantage Actor Critic (A2C), Deep Deterministic Policy Gradient (DDPG), DQN, PPO, Soft Actor-Critic (SAC), TRPO and a maskable PPO. The library has various integrations to other libraries, such as a framework that provides pre-trained agents and tuned hyperparameters, OpenAI gym, a library that has implementations of latest RL methods and also TensorBoard, a framework to visualize training and evaluation for experimenting. Also, various types of action- and observation-spaces are supported, including, but not limited to, Discrete, Multi-discrete and Box. It is also possible to customize policy networks. The documentation is extensive (Raffin et al., 2021).

TF-agents TF-agents is a RL library that is using TensorFlow, another popular open-source machine learning framework, as its base. It supports multiple state-of-the-art methods such as DDPG, DQN, DDQN, PPO and variations of PPO (PPOClip and PPOKLPenalty), REINFORCE and SAC. The environments from OpenAI gym are supported, as well as custom environments. The agents are able to handle

Discrete and Continuous action-spaces. Various tutorials are provided, including how to create custom environments, creating custom policy networks, handling the collection of data and saving/loading previous policies. It supports the integration of TensorBoard to visualize training and evaluation data. Base concepts of Reinforcement Learning are explained, when following the guides on the website. The documentation is clear and extensive. It is easy to comprehend, what respective functions are doing. Therefore, making the process of customization easy (Guadarrama et al., 2018).

Summary OpenAI gym provides an extensive number of environments, however, no implementations of state-of-the-art RL methods are implemented. When using OpenAI gym alone, the user has to implement its own implementation of methods. The other libraries, all offer those implementations, on top of providing the environments from OpenAI gym. The libraries have good documentations, but some are clearer and easier to grasp. All three libraries provide solid solutions for experimenting with state-of-the-art RL methods and the choice might rely on a feature, that is included in one, but not the other libraries.

TF-agents was chosen as the library for the experiments. It provides implementations for the methods DQN, DDQN, REINFORCE and PPO, which are used later on. It has an API (Actor-Learner) that makes the usage of agents even easier and enables distribution of training on multiple Graphics processing units, however, for this thesis only the former part was relevant. As a result, the code for the agents was constructed in such a way, that the four RL methods and environments may be easily changed out, which makes it less complex and less error-prone.

4.1.2 Setup instruction

The setup requires following libraries to be installed on the system:

- 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

Additionally, to installing these libraries, the TF-agents library has to be adjusted. The reason for this is that currently it is not possible to use action-masking with PPO agents, due to a bug in the source code. The code for creating a value network is missing an argument to determine, whether to use a mask, or not. Therefore, this argument has to be added. The file is located under following path:

```
1 /usr/local/lib/python3.8/dist-packages/tf_agents/networks/value_network.py
```

An argument `mask` with the value `None` has to be added to the function `call`, in order for action-masking to work. An issue (#762) addressing this problem was created in the library its repository on GitHub.

Scripts for the whole setup process on Windows are provided in section 4.1.3 and 4.1.3. By using Docker, it is possible to virtualize a Linux operating system on Windows, which is needed for the Reverb library. Both scripts, 4.1.3 and 4.1.4, have to be saved in a common folder. The former one is a Windows batch file that is running two docker commands. The later one has to be named *Dockerfile*, as it is the configuration file for Docker and tells Docker what to do. The Windows batch will first execute a command, that will build a Docker container in regards to the configuration file. Docker will install a Linux operating system, as well as, the python packages tf-agents and dm-reverb. The other libraries, IPython and ipykernel, will be installed automatically alongside. Additionally, Docker will fix the previously mentioned bug, by editing the file and adding the argument. The second command of the Windows batch file will run the container and map ports between the host and virtual machines according to the configuration. A jupyter instance will be running on the docker container and is accessed by opening the provided URL from Docker, or attach an IDE, e.g. Visual Studio Code, to the running container. Additionally, the data for training and evaluation can be inspected via TensorBoard, which is hosted on localhost with corresponding port(s) (typically the first port from the configuration file, i.e. 6666).

4.1.3 start.bat

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

4.1.4 Dockerfile

```
1  FROM tensorflow/tensorflow:2.9.1-jupyter
2
3  RUN pip install tf-agents==0.13.0
4  RUN pip install dm-reverb==0.8.0
5
6  # https://github.com/tensorflow/agents/issues/762
7  RUN sed -i 's/training=False/training=False, mask=None/g'
       /usr/local/lib/python3.8/dist-packages/tf_agents/networks/value_network.py
8
9  WORKDIR "/tf"
10 RUN rm -rf tensorflow-tutorials
11 COPY ./ /tf/
```

4.2 Environment

The games have to be specifically designed in a way, so that the agents from the RL library are able to interact with the game, which is essentially called an

environment. The environment has to fulfill several requirements. It is essential that it provides several information regarding the structure of the game and implement functions, that will be used by the agents. Both games were designed with regarding the requirements and have to implement an interface provided by TF-agents. By implementing the class, it is ensuring, that the agents have all information they need, in order to train, or evaluate, policies. If a requirement is not implemented, or implemented incorrectly, the agent will not start training, or evaluation. Following requirements need to be implemented:

1. The environment has to be written as a class and implement an interface provided by the library.
2. Specifications that describe the shape, datatype, the minimum and maximum values and name of the actions.
3. Specifications that describe the shape, datatype, the minimum and maximum values and name of the observations.
4. A step function has to be implemented, which will be executed at each iteration. This will handle the interaction between the action chosen by the agent and the environment. A tuple consisting of the step type, such as first, mid, or last step in a sequence/episode, the reward for the agent its action, the next state of the environment and a discount for the reward is returned at the end of this function.
5. Steps to be executed during each step iteration, such as calculating the reward for the current action, calculating the next state based on this action.
6. A reset function, which will be executed at the beginning of each sequence/episode. Mainly used to reset all variables, which were affected by a previous run, i.e. resetting the state to its initial value, resetting step counter.
7. (Optionally) information on how the environment should be presented

4.2.1 Mastermind

The features for the Mastermind environment are as following:

1. It is possible to choose 4 out of the following 6 colors for the code: yellow, orange, red, blue, green and brown. Accordingly, the size of the action space covers 6^4-1 units, each of which covers a possible color combination. The actual action is formed by 4 numbers ranging from 0 to 5, each of which an index for a color. However, internally, these numbers are transformed into a single number. The reason for this is, that the DQN and DDQN agents were not able to process multi-dimensional actions, as the initial action-space was designed as a 1×4 array. Thus, the action-space had to be reduced to a single number ranging from 0 to $6^4 - 1$.
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 space consists of a 2×4 array. The current state of the environment consists of information regarding the guess of the codebreaker and the feedback of the codebreaker. This information is stored separately in two arrays. The feedback is calculated by checking how many colors were guessed at the right and wrong position, resulting in two values, whereas the first value always tells how many colors were guessed correctly at the right position and the second how many colors were guessed at the wrong position. The remaining two values of the feedback array are filled with zeroes, as it is required that arrays in an observation have the same size. The second array in the observation space is holding the mask for action-masking. The mask is an array of Boolean values, which indicate, whether an action (color combination) is valid or not. It is updated at each iteration, until eventually only the code itself is left. For generating the mask, three different approaches were followed, which will be discussed in chapter 5.

The boundaries of these values are according to the number of total colors.

3. During the iteration of a step, the action is transformed from a single digit, into the representation of the respective color combination. The current guess in added to a list, which contains all guessed so far. The feedback is calculated by checking how many colors are at the right and wrong position and is then used, alongside the action, to form the next state. Additionally, the mask is updated to exclude invalid actions from the possible actions. Finally, the reward is calculated and returned with the information, whether the episode finished, or not.

4. At the beginning of each episode, a new random code is generated. The mask is reset to its initial value, so that all actions are valid. The current state is also set to its initial empty value. The end of an episode is reached, when either the agent breaks the code, or 10 rounds have passed.

5. The environment is rendered in the terminal, by iterating the list containing all guesses so far and their respective feedbacks. Appropriate colored tiles are displayed to represent the colors of the code and guess, as well as the feedback.

4.2.2 Battleship

Battleship features have been implemented as following:

1. The action-space for Battleship is straight-forward, 10×10 actions, as the game is played on a board of the same size. Each action is reflecting the coordinates of the cell to attack. The value of the action is again a single digit integer, due to the constraints mentioned earlier. The values for actions are range from 0 to 99.

2. The specifications for the observation-space, similar to the domain of the Mastermind environment, consists of two values. The shape is the same as in the Mastermind environment a 2×4 array. The first value is covering the

current state of the environment, which is the observation of the agent, i.e. the game board with all hits, misses and unexplored cells. The second value is the mask for the actions. The observation is a flattened array of 100 values, where each of which, is a value ranging from zero to two. These values indicate the state of the corresponding cell:

- 0 - unknown, i.e. this cell has not been explored yet
- 1 - explored cell: miss, i.e. did not hit a battleship
- 2 - explored cell: hit, i.e. hit a part of a battleship

The mask consists of an array of 10×10 Boolean values, to indicate, whether if a cell has been already explored, or not. Therefore, repeating an action, that attacks an already explored cell, is regarded as invalid and eliminated from the possible actions.

3. While iterating through steps, the action is transformed into its respective x and y coordinates. Internally, a representation of the actual board is used to check, whether the agent hit a battleship, or not. This board is hidden from the agent. Upon hit, or miss, the value of the board, that the agent is seeing, is updated and a reward is provided. Finally, the current action is regarded as invalid for future actions and the reward is returned, alongside the information, whether the game is finished, or not.

4. At the start of each episode, various things are set to its initial value. The state hidden from the agent and the state visible for the agent are reset to empty values. Battleships are created and random coordinates are assigned for each of them. The ships are placed either horizontally, or vertically. They are allowed to touch, but not to overlap. Accordingly, the visible and hidden states are updated to include the ships. The number and size of the ships are as following:

   - Size: 2; Quantity: 1
   - Size: 3; Quantity: 2
   - Size: 4; Quantity: 1
   - Size: 5; Quantity: 1

5. The environment is once again rendered in the terminal. Both states are displayed next to each other, displaying the state of the cell with appropriate emojis, i.e. the visible state is using a ? for displaying unexplored cells, while for the hidden state a water droplet is used. The droplet is also used for missed shots in the visible state. A hit on a ship is represented by a fire emoji on the visible state and ignored on the hidden state. Finally, each coordinate of a ship contains a ship emoji on the hidden state. It is feasible to see what the agent is doing using this representation.

4.3 Policy

Policies have various tasks and depend on the underlying implemented method. The number of how many policies an agent has, also varies. Most of the agent its policies are configurable, without creating an entirely new agent (and policy). All four agents have each two policies, one for collecting data and one for evaluation. The DQN and DDQN agents use per default an ϵ-greedy policy for collecting data and a greedy policy for evaluating it. The REINFORCE with baseline agent uses a greedy policy for its evaluation and a stochastic policy for its data collection. Lastly, the PPO agent uses a stochastic policy for both, data collection and evaluation. It is possible to replace the default policies, or wrap them with a strategy, such as a greedy one.

4.4 Pre-Processing

In Mastermind, the agent is choosing an action from 0 to $6^4 - 1$. This is necessary, as the DQN and DDQN agents are not able to handle multi-dimensional actions. However, for the observation and rendering the game, an array containing the indices of each guessed color is necessary. To solve this issue, the action of the agent is transformed from a single digit, into an array of four numbers (when four numbers are required for the secret code). These four numbers, each correspond to a distinct color from the color pool and can be used for the next state and rendering the game. 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 this type of issues is to give the agent a penalty for invalid actions. A penalty includes, giving the agent a negative reward, when it is performing a pointless action. The penalty is supposed to discourage the agent to repeat actions, that will not result a desired state (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 ac-

cording to feedback and in Battleship, actions were masked based on hits and misses.

5 Experiments

This chapter presents the conducted experiments and the results. It will explain the process and reasoning behind the experiments.

5.1 Hyperparameters

For each of the agents, the default hyperparameters values of the TF agent implementation are used. The learning rate of $1e-3$ was adopted from the tutorials of TF-agents. There are two different set of hyperparameters: one for the neural network(s) and one for the method.

5.1.1 DQN & DDQN

Q-Network

- 2 fully-connected layers: 75 and 40 units
- Activation function: ReLU

Method

- ϵ-greedy: 0.1
- gamma: 1

5.1.2 REINFORCE

Actor Network

- 2 fully-connected layers: 200 and 100 units
- Activation function: ReLU

Method

- gamma: 1

5.1.3 PPO

Actor Network

- 2 fully-connected layers: 200 and 100 units
- Activation function: ReLU

Value Network

- 2 fully-connected layers: 75 and 40 units
- Activation function: ReLU

Method

- ratio clipping: 0
- gamma 0.99

Throughout the experiments, these hyperparameters will be used, unless otherwise noted.

5.2 Mastermind

In the Mastermind environment it is possible to choose 4 out of the following 6 colors for the code: yellow, orange, red, blue, green and brown. Accordingly, the size of the action space covers $6^4 - 1$ units, each of which covers a possible color combination. The observation space consists of 2×4 array.

First approach The list of possible actions is narrowed down, by generating a feedback between the guess and the actions from the list. Only actions, that yield the same feedback, as the original feedback with the secret code, are regarded as valid actions. The logic behind this strategy is, that the secret code is obviously included in this list and, necessarily, a feedback based on the guess had to be made. So, the code, has to be an action, which yields the same feedback. With this strategy, the list of possible actions is drastically reduced at each step.

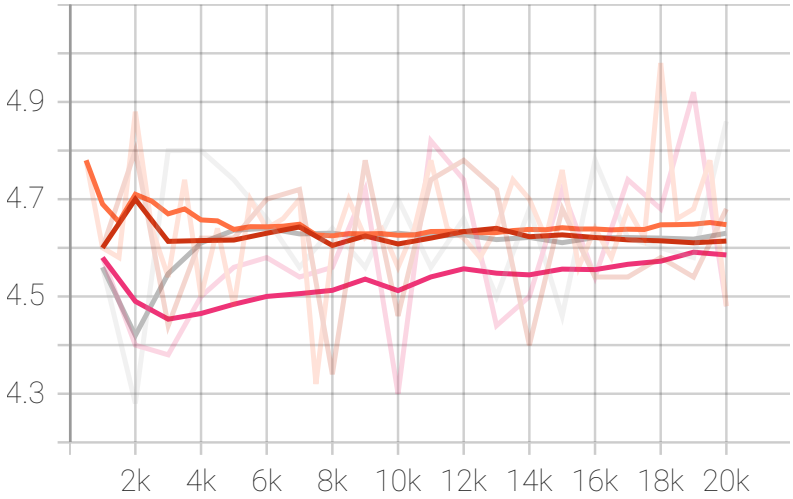


Figure 5.1: Average episode length in Mastermind with Approach 1. The x-axis displays the number of steps and the y-axis the episode length. Orange for DQN, red for DDQN, pink for PPO and gray for Reinforce.

Training was conducted for 20.000 steps. The graph in figure 5.1 shows the average length of an episode of all four agents. It is evident, that the average episode length fluctuates between 4 and 5. All agents start off with a low value for the average episode length and an improvement in relation to training duration cannot be concluded, hence, the agents do not require the used number of training steps, to successfully play the game. This is due to the applied strategy. After excluding actions, that are not deemed as valid, around the fourth round, only the code itself is left as a valid action, which is also the reason, why all agents require the same number of moves, to break the secret code.

Table 5.3 shows the result of evaluating the agents on 1000 iterations. All agents are able to solve the secret code in 4.6 moves, with the DDQN agent being the slowest (4.675 moves) and the Reinforce agent the fastest (4.625 moves). However, all agents are performing equally well, as they are able to solve the secret code within the same number of rounds and differ only marginally in the decimal point.

Table 5.1: Average number of moves required, to finish an episode, i.e. solving the secret code in 1000 iterations.

Algorithm	Number moves
DQN	4.629
DDQN	4.675
REINFORCE with baseline	4.625
PPO	4.654

Second approach This approach uses the strategy, where possible actions are reduced, as well. However, the deselection of the colors is based on different terms. An agent that selected an action, that yielded no feedback at all, i.e. none of the colors were on the right, nor wrong position, are all excluded from future actions. Additionally, an action that yielded in full feedback, i.e. any combination of colors placed at the right, or wrong, position, where the sum equals the total number of colors to guess, was also used for shrinking the size of legal actions. From all colors, only the colors from this action are regarded as valid, hence, the remaining colors are regarded as invalid and never chosen.

The observation space was changed for this approach, as initial results showed no learning. The colors were one-hot encoded, meaning, that each color is represented by a list of 0 and 1s in specific order, instead of the index of the color. The feedback is also one-hot encoded, leading to a list by the size of numbers to guess multiplied by two, for both colors. Feedback always starts with the black peg and if no feedback is given, both values are zero. The total numbers in a single state are $6 \times 4 + 2 \times 4$, where in the first part, the values for the colors is encoded (4 is the number of colors to guess and 6 is the number of total colors) and the second part is encoding the feedback. Both numbers are multiplied, by the number of rounds, an episode is lasting, to represent the entire game board. Resulting in an observation, which is eventually flattened. An example of all values, before stacking into a state, that will be added to the observation may look like following:

- $\underbrace{[0,0,0,0,0,1]}_{\text{color\#1}}\underbrace{[0,0,0,0,1,0]}_{\text{color\#2}}\underbrace{[0,1,0,0,0,0]}_{\text{color\#3}}\underbrace{[0,0,1,0,0,0]}_{\text{color\#4}}$
- $\underbrace{[1,0}_{\text{feedback\#1}}\ \underbrace{1,0}_{\text{feedback\#2}}\ \underbrace{0,1}_{\text{feedback\#3}}\ \underbrace{0,0]}_{\text{feedback\#4}}$

The reward function in this experiment was also altered. In this experiment, the agent will receive a positive reward for making guesses, that results in full feedback, i.e. a feedback where the sum of black and white pegs equals the length

of the code, or in none feedback, i.e. the agent was not able to get any correct colors. In this approach, the total number of rounds has been increased from the initial 10 to $6 * *4 - 1$, as the initial value was not high enough. In order to compensate the high number of rounds, the previous reward is scaled down with the number of rounds played and is reduced as the game progresses, to encourage finding colors early on, that are not in the code, as they are masked and the agent would be left with sorting the colors into the right positions. Alongside a negative reward for each round played is given. The agent receives a negative reward for the distance between the code and the guess. Finally, a high reward for finding the secret code is given. Figure 5.2 shows the result of this approach.

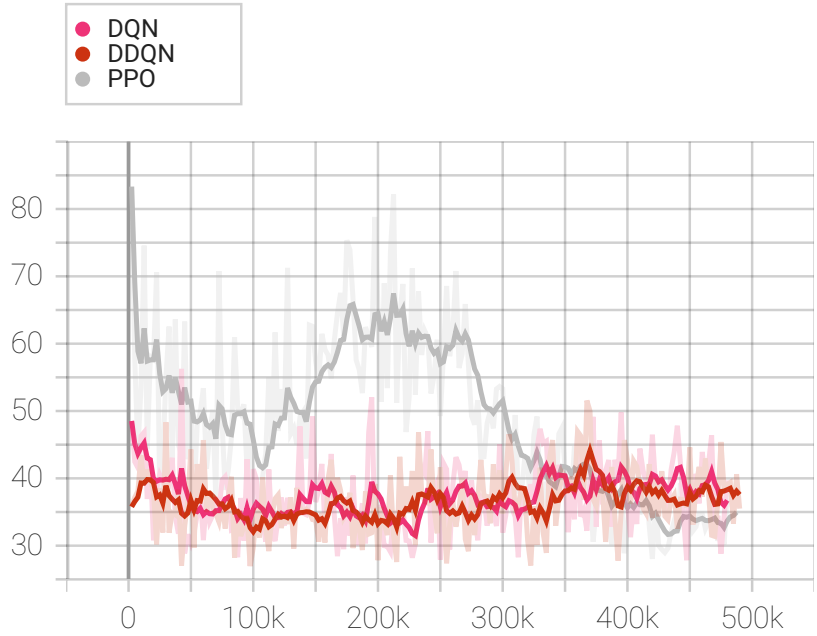


Figure 5.2: Evaluation metrics of the DQN, DDQN and PPO agents in respect to one-hot encoded observation-space. The Reinforce agent is not included in this experiment, due to training taking too much time. The x-axis displays the number of evaluation steps. The y-axis displays the average episode number.

The Reinforce agent was not included into this test, as the training duration was too high. It needed 8 hours and 27 minutes for just 50.000 training steps and training it for 500.000 training steps would take 84.5 hours. The remaining three agents were trained for around 500.000 training steps. The DQN started with an average episode number of 48.52 and the DDQN with 35.78. The DQN managed to reduce its number to 39.7 and the DDQN agent to just 35.5. It appears, that the DDQN agent was not able to improve its policy, while the DQN agent was and managed to reduce the number of guesses it needs to break the code by around 9 rounds. During evaluation, both agents were able to get the same minima, 28 for the DQN and 27 for the DDQN, various times. This, and the course of the graph, indicate that both agents were not able to improve their policies early on and remained the same. On the contrary the PPO Agent, was able to improve its policy during most of the training. The PPO agent had a much

higher starting value. At the start of its training it needed 83.32 moves to finish an episode. However, it was able to reduce this number by around half, just after a couple training iterations. It appears to be unstable, during the first 300.000 training steps, as the difference between the maxima and minima amounts to around 40%. Afterwards, the differences got smaller and the PPO is able to guess the code in 35.04 rounds on average at the 500.000 mark. The lowest episode number achieved is around 28. The last 100.000 training steps indicate, that further training would not result in a better policy for the PPO agent.

Table 5.2: Average number of moves comparison. Note: The episode length is capped at 10 for Approach 1 and at $6^4 - 1$ for Approach 2.

Algorithm	Approach 1	Approach 2
DQN	4.629	39.7
DDQN	4.675	35.5
REINFORCE with baseline	4.625	-
PPO	4.654	35.04
Random policy	9.82	790.04

Table 5.2 shows the average guess number that each agent needed to break the code using their respective approaches. The values for each agent, in their respective approaches, are fairly similar. The only outlier is the DQN agent in Approach 2, which needs around 4 moves more, than the other 2 agents. All agents are able to perform better, than a random policy, in their respective approaches. Notably, the game is usually played over 10 rounds, but in Approach 2 none of the agents are able to crack the code in that time. They all need at least three times as much.

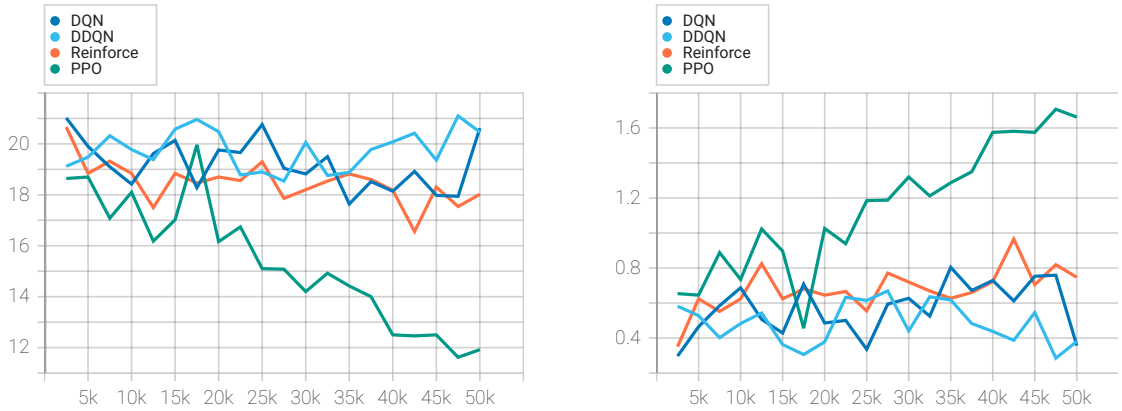
5.3 Battleship

The board game Battleship is usually played on board of 10×10. A player is able to place 6 battleships, each its size ranging from 2 to 5. The battleship of size 3 is placed twice. Several experiments will be conducted, each with various sizes of the board and numbers of ships placed. The ratio between empty and occupied cells is maintained, when possible. In the initial setting, this ratio is 17% of the cells are occupied.

5.3.1 Observation-space

The experiments regarding the observation-space were conducted on a grid size of 5×5 and one ship of size 4. The reason for this choice is, that it requires less training time, due to its less complex action- and observation-space. For the experiments, 50.000 training steps were chosen.

Single observation The first design choice was to use a single observation for monitoring the hits and misses of the agent. Hence, the values for the state range from 0 to 2, indicating whether a cell is unexplored, has a missed shot or has a hit, respectively. Figure 5.3 shows the average length of an episode and the average return with this approach.



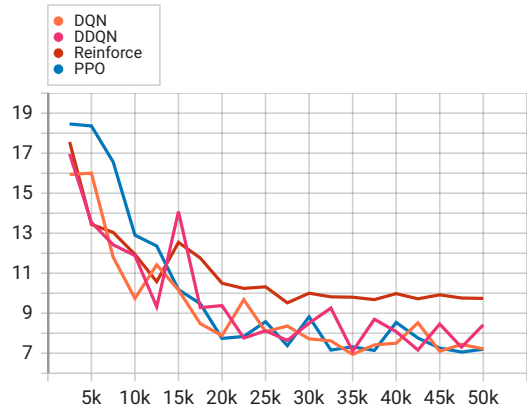
(a) Battleship single observation average episode length

(b) Battleship single observation average return

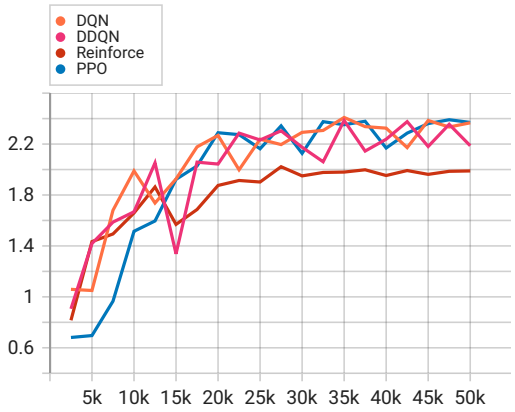
Figure 5.3: Evaluation metrics of all four agents in respect to single observation approach. In both diagrams the x-axis displays the number of evaluation steps. The y-axis displays the average episode number in (a) and the average return in (b).

With the single observation approach, the agents DQN, DDQN and Reinforce are not able to learn a good policy in 50.000 training steps. During the entire training process, the DQN und DDQN agents are averaging 19-20 moves per episode, which is equal to their starting value, to find the ship. The Reinforce agent is performing slightly better than the DQN and DDQN. It starts with 20 moves at the beginning and ends with 18 moves, which indicates, that this agent is learning, but at a slow pace. The best performing agent in this approach is PPO. At the end of the training it is able to find the ship in average of 12 moves. The curve is still declining at the end of the training, indicating, that it is able to improve its policy even more and finish an episode with fewer moves. The rewards for the DQN, DDQN and Reinforce agents are mainly constant, only the Reinforce agent has a slight incline. The PPO agent was able to get the highest average reward with a value of 1.66. The DQN and DDQN agent are getting a value of around 0.4 and the Reinforce agent a value of 0.75 at the end of the training.

Double observation The double observation, is using separate representations for the hits and misses. Both use 0 for the negative value and 1 for the positive value. Resulting in an observation with double the size of the single observation approach. The observations are initially processed in 10×10 lists, but returned flattened to reduce complexity. Results are shown in figure 5.4.



(a) Battleship double observation average episode length



(b) Battleship double observation average return

Figure 5.4: Evaluation metrics of all four agents in respect to double observation approach. 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).

In this approach, all agents are learning a good policy. They are able to finish an episode with single digits: Reinforce 9.74 moves, DDQN 8.42 moves, DQN and PPO in .7.2 moves. Initially, for the first 10.000 steps, the PPO agent need 1 more move than the other agents, to finish an episode, but is able to catch-up at 15.000 steps and perform equally well, if not better. However, the Reinforce agent is learning the game equally well, at the beginning, but at the 15.000 steps mark it is declining in performance (might be an outlier) and not able to recover and learn as good as the other agents, anymore. The graph indicates, that all agents were able to learn the game between 25.000 and 30.000 training steps.

Conclusion It is evident, that separate coding of hits and misses leads to better results. All agents developed a better policy and are able to solve the game in faster times. Previously, the DQN, DDQN and Reinforce agent were not able to play the at all, however, with this approach, they show performance similar, or equal, to PPO, who previously was able to play the game. Table 5.3 shows the percentage increase in performance, which is measured by the average episode length.

Table 5.3: The increase in performance of all agents with respect to both approaches regarding the observation space.

Algorithm	Performance increase in %
DQN	64.99
DDQN	58.85
REINFORCE with baseline	45.95
PPO	39.6

The DQN agent was able to increase its performance the most with a value of 64.99%. It previously required 19 moves to finish an episode, which essentially are random moves, however, with the second approach, the DQN agent is able to find the hidden ship on average within 7 rounds. PPO had the least increase in performance. It is only performing 39.6% better than before. However, the PPO agent was already performing with the previous approach well. It needed on average 12 moves and was able to reduce it to 7. The duration of the training time changed as well with this approach. The results are mixed in presented in table 5.4. Onwards, the second approach will be used for the experiments.

Table 5.4: The training duration of all agents in both experiments, as well, as the percentage change. Notably, DQN, DDQN and PPO need roughly the same amount of time, in contrast, the REINFORCE agent needs up to 8 times the duration of other agents. This will lead to much longer training times, with increasing grid size.

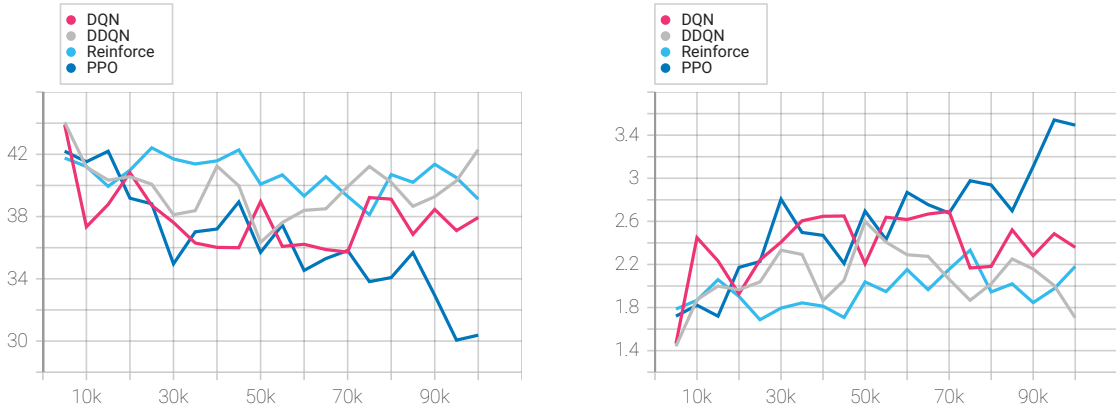
Algorithm	Duration single observation	Duration double observation	Change in %
DQN	9 min	14 min	+ 55.55
DDQN	9.5 min	10.5 min	+ 10.52
REINFORCE with baseline	1h 20 min	55.5 min	- 30.625
PPO	14.5 min	11.25 min	- 22.41

5.3.2 Varying grid-size

7×7 **grid** In this part of the experiments, the board has been enlarged to 7×7. The number and sizes of the ships placed are as following:

- size: 3 - quantity: 1
- size: 4 - quantity: 1

Training was conducted for 100.000 steps. Figure 5.5 shows the results of the agents for this run.



(a) Battleship 7x7 average episode length

(b) Battleship 7x7 average return

Figure 5.5: Evaluation metrics of all four agents in 7×7 board size. 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 results of this experiment show resemblance with the Single observation experiment. The DQN, DDQN and REINFORCE agents are not developing a good policy, whilst the PPO is. The DDQN and Reinforce agents are performing nearly as good as prior to the start of the training. They started with 44.06 and 41.76, and ended up with 42.3 and 39.12 moves to find both ships, respectively. The DQN agent did slightly better, it started with 43.9 moves and ended the training with 37.94. On the other hand, the PPO agent was able to develop a policy, which is learning. Its declining curve is indicating, that there is still room for improvement and with further training, the number of moves required to find both ships, may decrease even further from its start value of 42.2 and its end value of 30.38.

The results of the agents with stagnant performance led to adjustments, which will be described in the next paragraph.

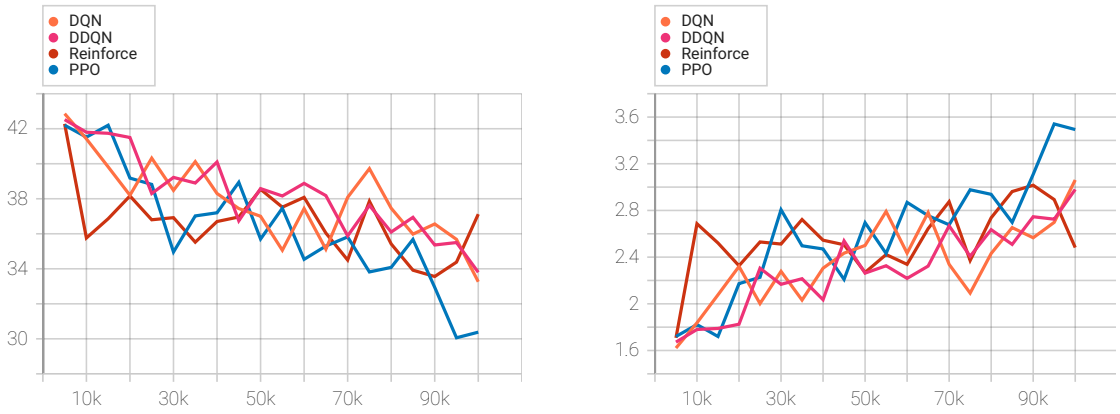
Trial ID	Show Metrics	ep_greedy	learning_rate	gamma	Average Return	Average Episode Length
b5562a6c1333a...	☐	0.000010000	0.10000	0.99000	0.76500	17.900
4f9aa8af73566...	☐	0.0010000	0.0010000	0.99000	0.79500	17.700
d5b063138cee1...	☐	0.10000	0.10000	0.90000	0.79500	17.700
e2eb5825fa835...	☐	0.0010000	0.0010000	0.90000	0.79500	17.700
31e22eb33ecd7...	☐	0.000010000	0.10000	0.90000	0.88500	17.100
f1e1078f73a57...	☐	0.000010000	0.000010000	0.90000	0.94500	16.700
3007c3f3f8665...	☐	0.000010000	0.000010000	0.99000	1.1250	15.500
0303140291d4e...	☐	0.10000	0.10000	0.95000	1.1400	15.400
f87255772c5b0...	☐	0.10000	0.0010000	0.90000	1.1700	15.200
b1606a2bcd867...	☐	0.0010000	0.000010000	0.90000	1.1850	15.100
77963d6b1c97d...	☐	0.0010000	0.000010000	0.99000	1.2000	15.000
55d3b6fe9084c...	☐	0.10000	0.000010000	0.99000	1.2450	14.700
6af7ef7b6ba3fe...	☐	0.000010000	0.000010000	0.95000	1.3500	14.000
cdef2fb50b456...	☐	0.0010000	0.000010000	0.95000	1.4700	13.200
82c157a5340c6...	☐	0.10000	0.000010000	0.95000	1.5000	13.000
2d35a21f43f65...	☐	0.000010000	0.0010000	0.99000	1.6200	12.200
529fc0ad13ced...	☑	0.10000	0.000010000	0.90000	1.7550	11.300

Figure 5.6: Detail from the HParams dashboard for hyperparameter optimization. The image detail is showing multiple values for the hyperparameters: ϵ-greedy, learning rate and gamma. Beforehand, metrics to measure the performance of an agent are selected and shown alongside (Average Return and Average Episode Length).

7×7 **grid optimized** In this experiment, the hyperparameter values, of the previously stagnant performing agents, were attempted to optimize. *HParams* dashboard, see in figure 5.6, will be used for this task. It is an addition to TensorBoard and allows to identify the best set of tested parameters. The changed parameters can be seen in Table 5.5.

Table 5.5: Adjusted hyperparameters of the agents.

Algorithm	First layer units	Second layer units	γ	Learning rate
DQN	old: 75	old: 40	old: 1	old: 1×10^{-3}
	new: 20	new: 100	new: 0.9	new: 1×10^{-5}
DDQN	old: 75	old: 40	old: 1	old: 1×10^{-3}
	new: 20	new: 100	new: 0.9	new: 1×10^{-5}
REINFORCE	old: 200	old: 100	old: 1	old: 1×10^{-3}
with baseline	new: 150	new: 0	new: 0.9	new: 1×10^{-5}



(a) Battleship 7x7 optimized average episode length

(b) Battleship 7x7 optimized average return

Figure 5.7: Updated evaluation metrics of all four agents in 7×7 board size in regards to hyperparameter optimizing. PPO hyperparameters were not changed and is only included as a reference. Training duration for DQN, DDQN, PPO and Reinforce were: 23 minutes, 22.5 minutes, 24 minutes and 5 hours 43 minutes, respectively. 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).

Figure 5.7 shows the results of agents trained for 100.000 steps in the same environment as in 7×7 grid. The adjustment of the hyperparameters had a mixed outcome on the agents, but overall, the effect is positive. The REINFORCE agent was able to end an episode with 37.12 moves and previously needed 39.12 moves. That's a difference of around 2 moves. The outcome for the DQN and DDQN agent is better. Previously, the DQN and DDQN agent needed 37.94 and 42.3 moves, now, they need 33.26 and 33.8, respectively. That is a difference of around 4 moves for the DQN and 8.5 moves for the DDQN. The DDQN agent benefitted the most from this optimization. Table 5.6 shows the percentage increase in performance.

Table 5.6: Change in average episode length, after optimizing hyperparameters for DQN, DDQN and REINFORCE.

Algorithm	Old Average Episode length	New Average Episode length	Change in %
DQN	37.94	33.26	- 12.34
DDQN	42.3	33.8	- 20.09
REINFORCE with baseline	39.12	37.12	- 5.11

The improvement of the DQN and REINFORCE agent may not seem significant, due to their changes being relatively low. However, the main take-away is, that

the curves of all agents are declining. Previously, this was not the case for DDQN and REINFORCE, and for DQN it was hardly notable. This indicates, that with more training, the agents will learn better policies and will be able to finish an episode earlier, than with just 100.000 steps, which was not the case in the previous experiment (except PPO).

Due to all agents achieving better results with different hyperparameters, the hyperparameters of the PPO agent were also adjusted using HParams dashboard. The new parameters can be seen in table 5.7.

Table 5.7: Adjusted hyperparameters of the PPO agent

Algorithm	entropy coefficient	ratio clipping	γ	Learning rate
PPO	old: 0	old: 0	old: 0.99	old: 1×10^{-3}
	new: 1×10^{-3}	new: 0.2	new: 0.7	new: 3×10^{-4}

10×10 **grid** The last experiment will be based on the original gameplay. It will be played on a 10×10 board. The number and sizes of the ships are as following:

- size: 2 - quantity: 1
- size: 3 - quantity: 2
- size: 4 - quantity: 1
- size: 5 - quantity: 1

The Reinforce agent will be considered as not eligible for this experiment, as the training duration is significantly higher, than the other agents. The second experiment on the 7×7 grid showed a training duration of 5 hours and 43 minutes, with 100.000 training steps. The other agents were around 23min each. Scaling the duration upwards to 1.000.000 training steps on the 7×7 grid, yields a training duration of approximately 57 hours, which may increase even more with increasing observation-space.

Figure 5.8 shows the result of the training for the DQN, DDQN and PPO agents. Training was conducted this time for 2.000.000 steps, as the increased observation-space required much more training to see progress.

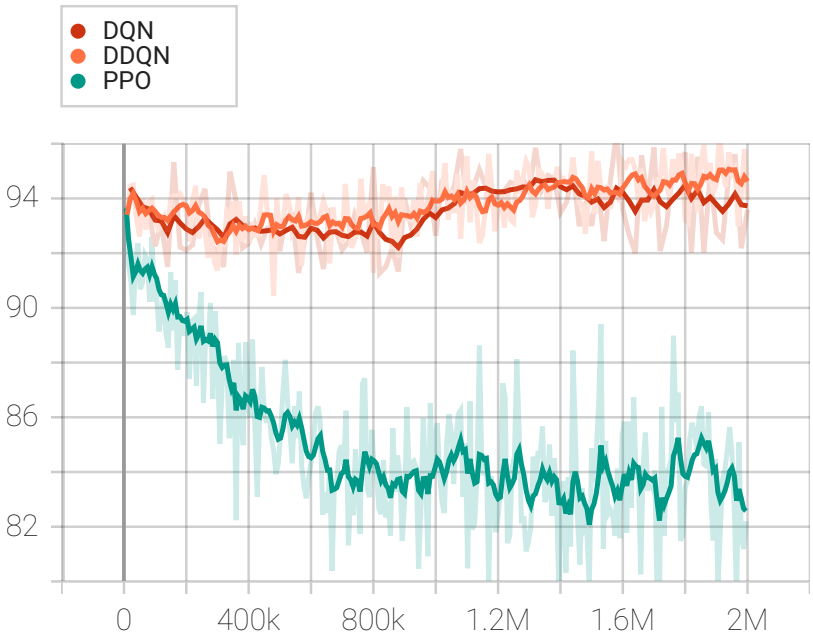


Figure 5.8: Evaluation metrics of the three agents DQN, DDQN and PPO on a 10×10 grid in Battleship. The x-axis displays the number of evaluation steps. The y-axis displays the average episode number

The figure shows, that the DQN and DDQN agents were not able to learn a good policy. After training for 2.000.000 steps, they pretty much learnt nothing, in regards to reducing the average number of episodes. The DQN and DDQN agents started with an average value of 94.39 and 93.4 to end an episode and ended the training with 93.59 and 93.94, respectively. There is no indication, that the agents would learn a better policy, by increasing the number of training steps. The PPO agent is able to improve its policy over time. Initially, the agent finished an episode in 93.64 moves and was able to reduce this number to 82.2. However, after 800.000 steps the PPO agent appears to stagnate and not able to further improve its policy.

These findings led to a final change in the observation-space. The status of each ship will be added to the observation-space, i.e. whether they are alive, or not. This resulted in the following observation:

- $\underbrace{[1, 0, 0 ... 0, 1, 0]}_{10 \times 10 \text{ misses}}$
- $\underbrace{[0, 1, 1 ... 1, 0, 1]}_{10 \times 10 \text{ hits}}$
- $[0, 0, 1, 0, 1,]$, where 0 indicates for not destroyed and 1 for destroyed; sorted ascending by the size of the battleship

The changed observation-space yielded (partially) new results. The comparison for the DQN agent is illustrated in figure 5.11.

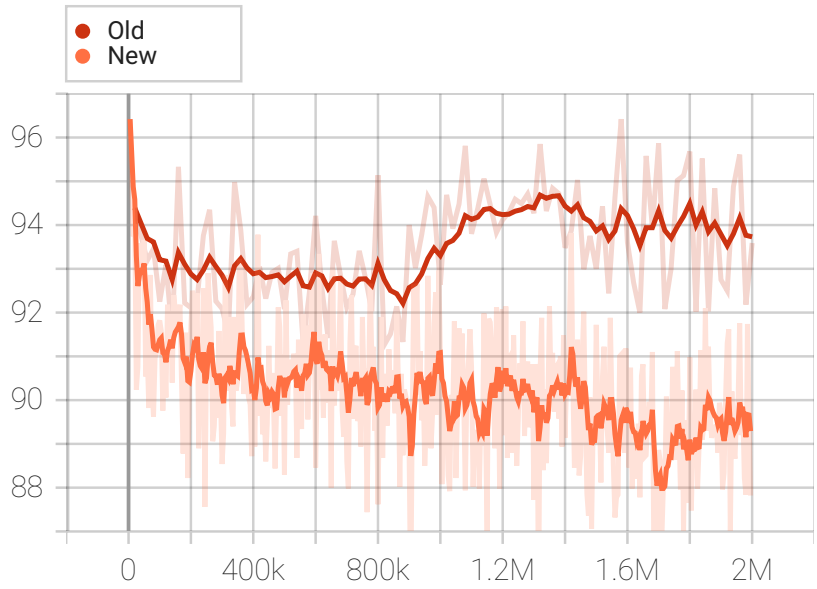


Figure 5.9: Evaluation metric for the DQN agent with (new) and without (old) including the status of the ships in 2.000.000 steps. The x-axis displays the number of evaluation steps. The y-axis displays the average episode number.

It is evident, that including the number of ships alive and destroyed, improved the policy of the DQN. Previously, the DQN agent was able to end an episode after 93.59 rounds and with the new observation-space, it is able to end an episode in 89.21 moves, on average. The DQN agent was to reduce the average episode length by 4.68% as shown in table 5.8. The lowest value it achieved was 85.06, after roughly 1.300.000 steps. Additionally, the graph is indicating, that the agent may learn a better policy, as it is slowly declining.

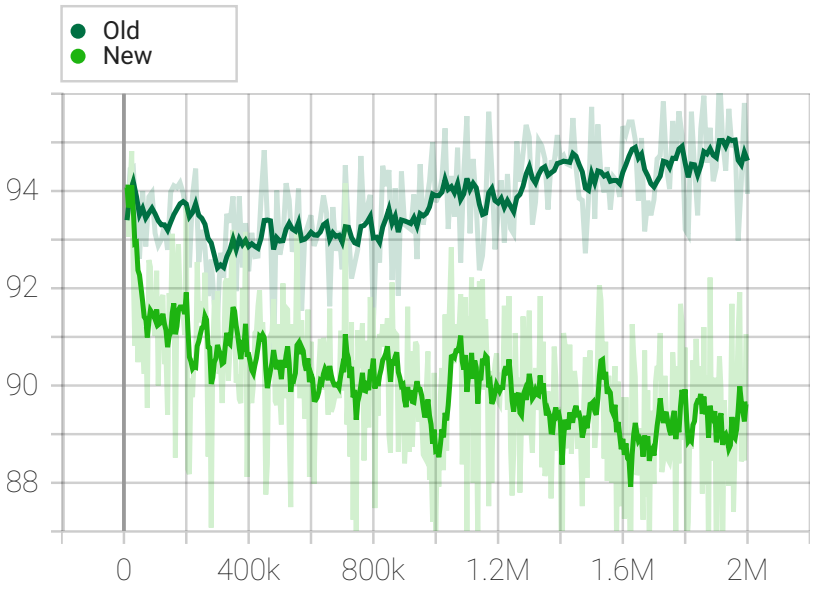


Figure 5.10: Evaluation metric for the DDQN agent with (new) and without (old) including the status of the ships in 2.000.000 steps. The x-axis displays the number of evaluation steps. The y-axis displays the average episode number.

Similarly, the DDQN agent was able to improve its policy. The DDQN agent was finishing an episode on average in 93.94 moves, with the previous approach. By including the number of ships alive, and destroyed, the agent is now able to finish an episode on average in 89.64 moves. This a decrease of 4.58% in terms of average episodes needed, to finish an episode. At around 1.400.000 steps, it got its lowest number of 84.92 in the average episode length metric. Again, the graph is indicating, that the agent may be able to learn a better policy, due to the graph gradually declining.

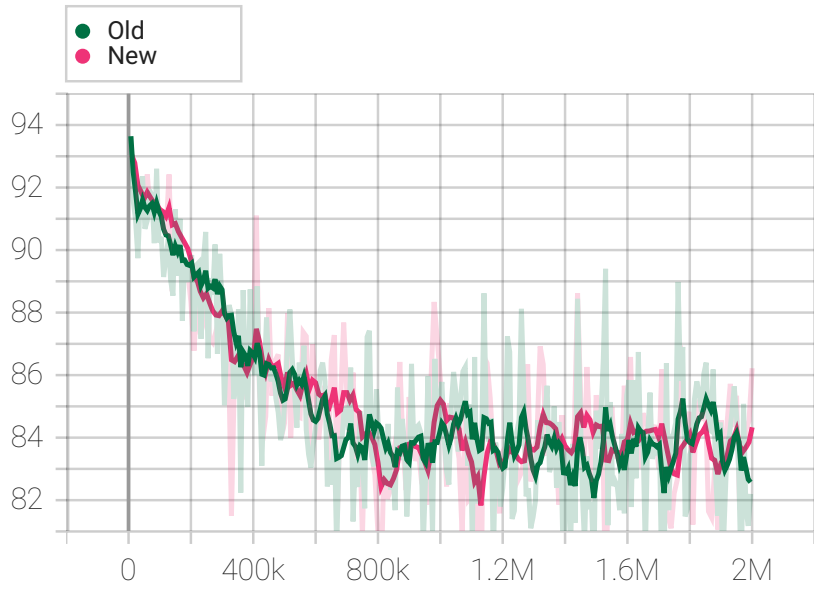


Figure 5.11: Evaluation metric for the PPO agent with (new) and without (old) including the status of the ships in 2.000.000 steps. The x-axis displays the number of evaluation steps. The y-axis displays the average episode number.

The results for the PPO agent differ from those for the DQN and DDQN agent. The PPO agent was able to learn a policy, that outperforms its initial performance. In fact, there is little difference in the training progress of the PPO agent as far as the different observation spaces are concerned. The starting value of the average episode length is similar for both settings, 93.64 without the status of the ships and 93.04 with. After 2.000.000 training steps, the agent is able to end an episode on average in 82.2 moves when the ship status is not included in the observation-space and 86.22, when it is included. However, the agent, with the ship status in its observation-space, is able to achieve a lower value for the average episode counter, just 5.000 steps before the training ends. It was able to end an episode on average in 81.7 turns. Its lowest value is 78.14 at around the 1.000.000 mark. The lowest value for the agent without the ship status is 79.3. It achieves 79 multiple times during its training. These results indicate, that there was no merit, in including the status of the ships for the PPO agent. Previously, it performed slightly better. It appears, that the PPO agent, with ship status, will not be able to learn a better policy. The average number of completed episodes does not appear to be getting smaller and its lowest values it has achieved, were around the 1.200.000 $\pm$ 400.000 step mark. This is not the case with the PPO agent without ship status. It has achieved lower values with increasing training time. However, the differences between the values are in the decimal range, which may appear insignificant.

Table 5.8: The table shows the difference between the approach with, and without, including the ship status in the observation-space of a battleship game on a 10×10 grid, after 2.000.000 steps.

Algorithm	10×10 avg. ep length	10×10 incl. ship status avg. ep length	Change in %
DQN	93.59	89.21	- 4.68
DDQN	93.94	89.64	- 4.58
REINFORCE with baseline	-	-	-
PPO	82.2	86.22	+ 4.89

The changes of training duration among the agents DQN, DDQN and PPO, in regards to adding the ship to the observation, are shown in table 5.9. As a matter of fact, an attempt to train the Reinforce agent, under the same settings, was conducted. The agent needed 33 hours and 2 minutes to train for 50.000 steps.

Table 5.9: Comparison of training duration between the first and the second approach, where the status of the ships is included in the observation-space. The value-based methods, DQN and DDQN, appear to be more prone in terms of training duration with respect to the increasing observation-space. The PPO agent was affected by less than, in half in comparison to the other agents, in regards to training duration. However, the PPO agent also has initially higher training time, than the other two.

Algorithm	10×10	10×10 incl. ship status	Change in %
DQN	4 hours 4 minutes	5 hours 31 minutes	+ 26.28
DDQN	4 hours 11 minutes	4 hours 57 minutes	+ 19.19
REINFORCE with baseline	-	-	-
PPO	5 hours 25 minutes	5 hours 57 minutes	+ 8.96

5.4 Summary

In this section, the Reinforcement Learning methods Deep Q-Network, Double Deep Q-Network, REINFORCE with baseline and Proximal Policy Optimization were experimented on two custom implemented environments. The environments for this task are the games Mastermind and Battleship. The implementation of the experiments has been oriented to the Actor Learner API from TF-agents. This enabled switching out agents and environment, without having to implement the

repetitive configurations for each agent individually. Initially, the hyperparameters for each agent were adapted from guides and tutorials located on the TF-agents website and repository.

5.4.1 Mastermind

The first game to experiment on was Mastermind. Two experiments were conducted, each following the same strategy, but with a different approach. The strategy is to reduce the number of possible actions. In the first approach, this was done by initialized a list, that contains all possible color combinations for the secret code. After each guess from the agent, a feedback is received, which tells, how many colors were guessed at the right wrong position. This guess is used to reduce the number of possible actions. A feedback for all possible actions is calculated, using this guess as the base. Only actions, that result in the same feedback are kept in the list of possible actions. The second approach used a different strategy, to reduce the list of possible actions. At the end of each turn, first, the current action is removed from the list of possible actions, so the agent is not repeating itself. Furthermore, the feedback is used again, to reduce the number of possible actions. The conditions for removing an action from this list are as following:

- if the feedback results in (0,0), i.e. none of the colors are in the code, then all combinations, containing any of these colors, are removed from the list
- if the feedback leads to a result, where the sum is the number of the colors to guess, e.g. (2,2), (1,3), when 4 colors have to be guessed, then all combinations, that do not contain any of these colors, are removed from the list

The results for the second approach showed, that the agents clearly needed more time to break the code, than the first approach. The agents in the first approach only needed around 4.6 rounds to find the code, while agents with the second approach needed 35-39 moves. This means, that they are also not able to find the code in the required number of rounds. However, all agents are performing better, than a random policy, especially the agents with the second approach, as the random policy needed 790 moves, in contrast to the 35-39 moves from the agents.

5.4.2 Battleship

In Battleship, more experiments were needed. The reason is, that the original observation-space is big. The experiments were set up in a way, where the observation-space starts small and increases with each experiment. In the first experiment, the grid-size was 5×5 and only a 4-sized ship was on the grid. The focus of this experiment was on how to shape the observation-space and two approaches were compared. The first approach was to include a single grid into the observation,

consisting of values ranging from 0 to 2, where each value represented one of the following states:

- 0 - unexplored cell
- 1 - explored cell; missed shot
- 2 - explored cell; hit shot

In the second approach, the hits and misses were separated from each other, resulting in the use of two grid for the observation-space. Both grids only used 0 and 1 for its value, where former is indicated, that the cell is unexplored and latter, explored. The results of this experiment show, that 3 of the 4 agents (DQN, DDQN & Reinforce) were not, or barely, able to learn a good policy with the first approach. They were stagnating with their progress. The fourth agent, PPO, on the other hand, was able to learn a policy, that find a 3-sized ship on a $5\times$ grid on average in 12 moves. In the second approach, all agents were able to learn a good policy, in the same amount of training time, and learned to finish an episode in 7-10 turns. Thus, the second approach was used for the design of the observation-space. The observation-space also included a mask for possible actions, where at each turn, the current action is removed from, to prevent repeated actions.

For the second set of experiments, the grid-size was extended to 7×7 and a 3-sized ships was added. The configurations from the previous experiment were used for all agents, however, the result with this setup did not result in good policies, except PPO. The result looked similar to the first approach of experiment#1. For the second approach of this experiment, the hyperparameters for the agents, that were not performing good, were optimized by using HParams dashboard. The results show, that the agents have learnt better policies than before and are able to finish an episode with lesser moves. Additionally, the results indicate, that the agents may learn better policies with more training time, as the curve for each agent is declining, except the REINFORCE agent.

For the last conducted set of experiments, the observation-space has been adjusted, according to the original game i.e. grid by size of 10×10 and ships of size and quantity placed as following:

- Size: 2; Quantity: 1
- Size: 3; Quantity: 2
- Size: 4; Quantity: 1
- Size: 5; Quantity: 1

The difference between the first and second approach in these experiments is, that one has additional information in the observation-space, regarding the status of the ships, i.e. whether a ship is alive or destroyed. The number of training steps has been increased, due to having a higher observation-space and the agents needing more time to learn. The REINFORCE agent is not included in this experiment, due to the training time being substantially higher, than the other agents, as seen in table 5.4 and with the experiments on a 7×7 grid. The Reinforce

agent needed 5.72 hours for 100.000 training steps, which would scale up to 57 hours, when training for 1.000.000 steps. In the last experiments, the agents were trained for 2.000.000 steps, which would double the amount of time. Additionally, the observation-space has been increased, which may increase the training time as well. The results from this set show, that the DQN and DDQN agent were able to improve their policies, when adding the information regarding the status of the ships. Previously, both agents were not able to learn, how to reduce the average number of episodes, which basically tells, that they were not able find the hidden ships. However, with the additional information, both agents were able to learn a better policy and indicated, that with more training steps, they may acquire better performance, i.e. finding the ships faster. This was not the case with the PPO agent. The performance with, and without ship status, is pretty similar. There was no significant change in performance. The agent performed equally in both approaches and it appears, that the agent will not learn a better policy with increased training time, regardless of the approach for the observation-space. However, it is hard to tell, because the training is still in the early phase and it might change.

The results for these experiments show, that the agents, regardless of the method, were able to learn a policy, that will find a hidden ship in the game its simplest form. However, with increasing complexity, i.e. increasing the grid-size and number of ships, this becomes a more difficult task. It requires more fine-tuning of the configurations, i.e. hyperparameter optimization, properly encoding and decoding the domain spaces and designing a good reward function. Else, the agents will not be able to learn a good policy and perform poorly.

6 Conclusion

6.1 Summary

The goal of this thesis was, to find out whether state-of-the-art Reinforcement Learning algorithm are eligible to play strategic games. The games in question are Mastermind and Battleship. These games were chosen based on factors, such as popularity, complexity of implementation, but foremost, the ability to play the games as single-, or multi-player. A Reinforcement Learning library was needed to play the games in that setting. By researching state-of-the-art Reinforcement Learning libraries and considering factors such as, activity, extend of documentation, ease of use, implementation of state-of-the-art Reinforcement Learning methods, the TF-agents library from TensorFlow was chosen for this project.

TF-agents provides implementations, among others, of Deep Q-Network (DQN), Double Deep Q-Network(DDQN), REINFORCE and Proximal Policy optimization (PPO). These algorithms were chosen for the experiments, because they are conceptionally different. The former two are value-based and the latter policy-based methods. Value-based approaches use a value function to calculate an estimation on how beneficial it is to be in a given state of an environment and/or perform a specific action in that state. In policy-based approaches, a policy directly calculates the most optimal action for a given state. The algorithms accomplish tasks differently because to differences in their approaches. The DQN and DDQN algorithm are fairly similar, regardless of their approach. The DDQN algorithm is a successor of DQN and has improvements in some areas. REINFORCE and PPO are conceptionally related.

The games had to be implemented in regards to the TF-agents library. Reason for this, that in order for the agents to communicate with the game, interfaces have to be implemented. This will result in the agent-environment loop, where an agent can play the game and will be notified, on how it is doing.

During experimentations, the environments and agents have been adjusted multiple times, in order to improve the learnt policy, because initially, they were performing poorly. One very impactful change was to include action-masking. Action-masking allowed to exclude illegal actions from a set of possible actions. This reduced training time, because in early phases of the training, agents were constantly repeating the same actions. Additionally, action-masking was used to implement strategies and rule-sets in the case of Mastermind, which otherwise would've been done, by designing reward function, that encourages certain behaviors and that would increase training time. The agents were performing better, than random policies, using those features. The experiments with Battleship showed, that agents were able to play the game in its simplest form, with default

configuration. However, with increasing complexity, i.e. an agent has to account more things, such as, more cells and/or token on a game board, details regarding the configuration of the agents and/or regarding the representation of the game, had to be adjusted. Adjustments allowed the agents to play games, with increasing complexity. Nevertheless, with increasing complexity and more things to consider, the training time also increased, due to having bigger observation-spaces and more time needing to learn a good policy. One method in particular stood out: REINFORCE with baseline. The agent based on this method required substantially higher training times with the same setup as the agents based on other methods, which was also the reason to exclude it from the final set of experiments and regard it as not eligible for high domain spaces. The training time, which increases with the increase of the complexity, turned out to be a problem, at the later parts of the experiments. Due to having high training times, the full potential of the agents in the Battleship environment could not have been fully evaluated. It was only possible to estimate those, based on the early phases of the trainings. However, the experiments showed, that with trial-and-error, it was possible to gradually increase the performance of the agents. It is not required to have advanced and expensive Artificial Intelligence system, as introduced in chapter 1. State-of-the-art Deep Reinforcement Learning methods were able to play two classic strategic board games, but the success depends on the game its complexity and the design choices made to train an agent.

6.2 Lookout

Throughout this project, the implementation of the games and agents have been changes various times. Initially, as the agents were only repeating their actions in Mastermind, it was attempted to build a LSTM-based neural network for one of the agents. The idea was, that the agent would memorize its past actions and feedbacks (when the observation-space consisted of only a single row), and learn to associate it with its actions. However, upon finding an article related to action-masking, this approach was discarded. Thus, Thus, several changes were made to the bases, of some of which led to good results and some of which led to poor results. Using a home computer may have slowed down the training progress and a cloud service with fast GPU calculations could, which may have resulted in more test and better other results. Especially necessary were the changes concerning the environments action and observation space, as well as the reward function. The domain spaces were limited, due to TF-agents implementations of the Deep Reinforcement Learning methods, to only support specific types, which works fine in one of the other presented libraries, e.g. multi-discrete action, instead of single encoded action. The training duration of the REINFORCE agent was unusually high, in comparison to the other methods. This negatively affected the project. This was especially noticeable in the later stages of the project when the Battleship environment observation-space became larger. The already long training hours were additionally extended and waiting for the results took half a day, or a whole, just to test a few different settings. The experiments could have been expanded, by testing more settings and ideas. The optimization of the hyperparameters haven taken a long time, thus, only few values and variables were

optimized. The rules of the ship placement could have been altered, e.g. the mutual touching of the ships is no longer allowed. During viewing the agents playing, it appeared, that the agent was sometimes confused, as to, if the ship has been already destroyed, or not, thus, sometimes it did not "finish off" an opposing ship. However, this was not always the case, so certainty is not given and by including the ship status into the observation-space, this "issue" may have resolved. However, further experimentation was not possible, as it would have exceeded the given time frame for this thesis. Furthermore, the project can be expanded by several features: it is possible to add Multi-agent system. In this system, multiple agents are able to interact with each other. Both games provide the ability to play it with 2 players. In Mastermind, one agent could take the role of the codebreaker and the other agent the role as the codemaker. One agent would learn to break codes, the other would learn to create them. They could learn from each other, while they play. In Battleship, this system could reach to two variations. In variant one, similar to Mastermind, one agent would place the ships and the other will search for them. The first agent is learning the optimal placement for x amount of ships on a $n \times n$ board. The other, as the current implementation, is trying to find them. In the other variant, agents would play against each other. Either an agent places its ships themselves, or they play 2v2, where in one team, one agent is placing the ships and the other agent is attacking. A Multi-agent system can lead to a lot of variations, depending on the environment.

The development in the Reinforcement Learning area is very active, as shown by the publication of Deepmind and others. Their latest release is a generalized application, which is able to play multiple games, from various domains and different levels of complexity, without knowing any rules, or relying on human domain knowledge. This paves the way for general artificial intelligence and problem solving without specifications.

References

Achiam, J. (2017, October). *Advanced Policy Gradient Methods.* Retrieved from http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_13_advanced_pg.pdf ((Accessed on 12/12/2022))

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*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *Openai gym.*

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))

Hsu, F.-H. (2002). *Behind deep blue: Building the computer that defeated the world chess champion*. Princeton University Press.

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

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., ... Stoica, I. (2018). Rllib: Abstractions for distributed reinforcement learning. In *International conference on machine learning* (pp. 3053–3062).

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, December). Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, arXiv:1312.5602.

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*.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, *22*(268), 1-8. Retrieved from http://jmlr

.org/papers/v22/20-1364.html
Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., ... others (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609.
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*.
Sewak, M. (2019, 06). Deep q network (dqn), double dqn, and dueling dqn: A step towards general artificial intelligence. In (p. 95-108). doi: 10.1007/978-981-13-8285-7_8
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., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... others (2018a). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144. Retrieved from http://science.sciencemag.org/content/362/6419/1140/tab-pdf
Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... Hassabis, D. (2018b, 12). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*, 1140-1144. doi: 10.1126/science.aar6404
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=1128617717 ((Online; accessed 23-December-2022))
Wikipedia contributors. (2022b). *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. (2022c). *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.