Saarland University

Master Thesis

Performance Comparison of Deep Reinforcement Learning Algorithms for the ProGen Benchmark

Submitted by:

Ilka Kopping

Submitted on:

September 28, 2022

Reviewers:

Univ.-Prof. Dr. Verena Wolf

Univ.-Prof. Dr. Martina Maggio

Abstract

Deep reinforcement learning has recently been a fast-developing field of research. However, it is hard to compare different reinforcement learning algorithms due to varying evaluation protocols. Additionally, many protocols do not consider an agent's performance in similar yet modified environments. In this thesis, we trained a PPO, a REINFORCE, and two DQN agents in two settings on the ProcGen benchmark and compared their training and generalization performance and the differences between the settings. Additionally, we investigated the impact of different game mechanisms on the agents' performances. The results show that PPO outperforms the other agents in nearly all environments. However, REINFORCE and one of the DQN agents generalize best. We noticed that a dense reward setting benefits all agents and that some specific tasks within the environment benefit the agents' performances. Different surroundings had different impacts on each agent.

Acknowledgments

I highly appreciate the opportunity provided by Prof. Wolf and Timo P. Gros to work on this exciting topic.
I thank Prof. Maggio for her willingness to review this master's thesis.
Thanks to my advisor, Timo P. Gros, for all the advice and constructive discussions.
Also, thanks to Joschka Groß for jumping in during Timo's honeymoon.
Additionally, I wish to thank all other thesis students at the MoSi chair I had constructive discussions with during working on my thesis.
Lastly, I wish to thank my family and close friends who supported me during this time.

Contents

1. Introduction

Since the breakthrough of deep Q-networks achieving human-level results [33] on the famous arcade learning environment (ALE) [3], many new reinforcement algorithms have been developed. Many of them show promising results, indicating significant progress in deep reinforcement learning (DRL).

However, the evaluations of newly developed algorithms usually do not follow a specific protocol. Given the broad range of RL benchmarks, algorithms are seldomly evaluated on the same benchmark. However, a good performance on one benchmark does not imply a good performance on another one. But even if two algorithms are evaluated on the same benchmark, different evaluation procedures are often followed [29]. As a result, it is often impossible to compare the different results without further experiments.
What aggravates the problem even further is that the evaluation is commonly solely based on an agent's training progress. Implicitly it is assumed that the performance in similar environments could be similar as well. However, studies repeatedly demonstrated that this assumption does not hold [6, 7, 18, 48]. Even minor environmental differences can lead an agent to perform much worse, i.e., the agent overfitted to the known training environment. This means that the agent did not generalize this knowledge such that it can be applied in similar scenarios. Therefore, suggestions to additionally consider the generalization performance [48] as well came up [29, 47, 48].

Motivated by this, Cobbe et al. developed the coinrun environment [7] and, finally, the ProcGen benchmark [6]. This benchmark consists of 16 distinct environments. For each environment, the levels are sampled instead of humanly handcrafted. By restricting the training set to a fixed number of levels and evaluating the agent on newly sampled and unseen levels, the generalization skills of an agent can be easily investigated [6].

In this thesis, we use ProcGen environments as a training and evaluation environment to compare three different algorithms. Motivated by Henderson et al., who demonstrated that different implementations of the same algorithm could lead to different results, even using identical hyperparameters [17], as well as prior claims that one of the algorithms does not perform well on ProcGen [6, 34], we try two different implementations of this algorithm. Therefore, we compare four agents in total.

Our analysis comprises four different aspects.
Firstly, we will investigate the training performance of each agent. The training performance will be measured as usual in reinforcement learning. In the analysis, we will consider each environment's maximal and trivial scores [6] to get an impression of each agent's performance.
Secondly, we will investigate the generalization performance of each agent.
Thirdly, we will compare each agent's performance in easy and hard settings. These

settings differ in distribution over the difficulty level, the number of training steps, and the number of training levels.
Finally, we will investigate the impact of different environmental aspects.

Overall, this thesis aims to understand and evaluate the performances of the four agents in the ProcGen benchmark.

1.1. Thesis Outline

This thesis consists of seven chapters in total. After this introduction, chapter 2 provides a formal introduction to reinforcement learning. Chapter 3 discusses the topic of generalization in reinforcement learning. The ProcGen benchmark is introduced and discussed in chapter 4. The experimental setting is discussed and results are shown in chapter 5. An analysis of the results is performed in chapter 6. The analysis of each game mechanisms' impact to the agents is discussed in chapter 7. Finally, the thesis is concluded and provided with suggestions for future work in chapter 8.

2. Reinforcement Learning

Reinforcement Learning (RL) is a subarea of machine learning where an *agent* learns to achieve a goal within an *environment* via learning by trial and error. This learning process happens in a continuous interaction loop: the agent selects an action, and the environment immediately updates the agent with the new state of the environment and a possible reward. In the next step, the agent again performs an action and gets updated about the new state and possible reward. This process is repeated until the learning process converges. During the learning process, the agent aims to maximize the overall reward.

RL problems are usually represented as Markov decision processes (MDPs) [41]. We define an MDP as a tuple $(S, A, T, R, s_0, \gamma)$, where S is the state space of the problem, A is the action space of the problem, $T(s, a, s')$ with $s, s' \in S, a \in A$ is the state transition function which determines the next state s' for a state s after action a is applied. This transition function can either be deterministic, i.e., s' is unique, or stochastic, i.e., a probability distribution over multiple possible next states $s'_1, s'_2, ...$ Furthermore, $R(s, a)$ with $s \in S, a \in A$ is the reward function, indicating the reward signal the agent will receive for applying action a in state s. We denote the environments initial state by s_0. Additionally, the MPD has a *discount factor* $\gamma \in$ (0,1]. It is used to prioritize rewards over time. With prioritized rewards, short-term rewards are valued more during the agents' learning progress than long-term rewards. The agent tries to maximize the *discounted return*

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= \sum_{k=0}^{T} \gamma^k R_{k+t+1}, \end{aligned}$$

where $t = 0, 1, ..$ denotes the time steps and T is the last, i.e., the terminal, time step of the current episode. In non-episodic cases, T is ∞.

In order to maximize the return, an agent can follow different approaches for RL algorithms. This thesis discusses policy-based approaches and value-based approaches. Policy-based approaches focus on the agents' policy. A policy π indicates the probabilities for selecting an action a in state s. In small state spaces, the agent computes an optimal policy π^*. Optimal policies lead to maximal returns. In large state spaces, the agent needs to estimate π^*. This kind of RL algorithms is discussed in section 2.2. Value-based approaches focus on computing or estimating optimal state-value functions $v^*(s)$. A state-value function determines the expected return of a state s if following a given policy. The approach of value-based algorithms is discussed in section 2.3. Sections 2.2 and 2.3 discuss the algorithms relevant to this thesis. As all discussed algorithms fall into the deep reinforcement learning (DLR) category, we will first discuss this topic in section 2.1.

2.1. Deep Reinforcement Learning

In small environments, an agent can store its experiences in simple data structures and optimize the expected reward according to these prior experiences. This is not possible in environments with large state spaces, in which it is infeasible to observe even nearly all states. Therefore, an agent must be capable of generalizing experiences to similar states. To this end, instead of calculating the optimal policy or state-value function during the learning process, an agent needs to learn to *approximate* them. As a function approximator, an agent can use an artificial neural network (ANN) [41]. ANNs consist of several *units* grouped into layers. An input vector X with p variables $X = (X_1, X_2, ..., X_p)$ serves as *input layer*. An arbitrary number of *hidden layers* can follow it. Units in these hidden layers compute functions that the ANN will learn based on input data. An *output layer* completes the neural network. This output layer finally predicts the neural network's outcome $f(X) = Y$ [19]. The final output of a neural network is then the estimation made by the agent. In the following, θ refers to the parameters of a neural network.

A special kind of neural network is a *convolutional* neural network (CNN). CNNs are specialized for processing input data with grid-like structures. Hence, CNNs are often used for working with image input [13].

2.2. Policy-based Algorithms

Policy-based algorithms focus on the agents' *policy* to reach the goal within an environment. A policy π is a mapping over the state space S to a probability distribution over the action space A. Therefore, a policy π determines which action a the agent takes if being in state s with what probability. Therefore, $\pi(a, s)$ is the probability of the agent taking a in s. Policy-based algorithms use the interaction learning loop of the agent and the environment to adjust the agents' policy in order to maximize the expected reward. The agent chooses an action by sampling over the policy. Sampling has the side effect that, given enough training time, the agent will perform and observe the consequence of all possible actions.

2.2.1. REINFORCE

REINFORCE [45] is a policy-based gradient ascend method.

Given a policy parameterization $\pi(a|s,\theta)$ and a step size α, REINFORCE works by applying algorithm 1.

Algorithm 1 REINFORCE Pseudocode

```
Initialize a policy network $\theta$
while not converged do
    Perform a complete episode $e = S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    for $t = 0, 1, ..., T-1$ in $e$ do
        $\theta \leftarrow \theta_t + \alpha\ G_t \frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} = \theta_t + \alpha\ G_t\ \nabla ln\ \pi(A_t|S_t,\theta_t)$
    end for
end while
```

2.2.2. Proximal Policy Optimization

Another policy-based method is proximal policy optimization (PPO) [40]. It was introduced in two versions: a version using a clipped surrogate objective and a version using a kullback leibler divergence [28] penalty coefficient. Relevant to this thesis is the first version with the clipped surrogate objective. The key idea behind PPO is that for each update step the agent takes, it is ensured that the new policy does not deviate much from the old one. In order to achieve this, it uses two policy networks: the current policy $\pi_\theta(a_t|s_t)$ and the old policy, which was used for sampling, $\pi_{\theta_{old}}(a_t|s_t)$. Using this, let $r_t(\theta)$ be the probability ratio defined as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}.$$

Additionally, we introduce $\hat{A}_t$ as the estimator of an advantage function in t. An advantage function $A_\pi(s,a)$ indicates the value of an action [2]. In small state spaces, $A_\pi(s,a)$, the agent can compute the advantage of action a in state s as

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s).$$

The agent needs to estimate it in large state spaces, similar to the Q-function. $\hat{A}_t$ is used in this algorithm instead of the expected reward. Finally, we introduce the clipped objective for PPO as

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

where ϵ is a hyperparameter which controls the clip range. The main objective $\hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t]$, introduced by Schulman et al. for the trust region policy optimization (TRPO) [39], is clipped by the second term. Therefore, if the probability ratio is not within the interval $[1-\epsilon, 1+\epsilon]$, it is clipped. The clipping prevents changes in the policy that are too large. Through this clipping process, an agent's learning process becomes more stable, which leads to better results [40]. Finally, we can discuss the full PPO algorithm based on a clipped surrogate objective. Given policy parameters θ_0 and a clipping parameter ϵ, algorithm 2 demonstrates the PPO procedure.

Algorithm 2 PPO Pseudocode

```
Initialize $k = 0$
while not converged do
  Collect trajectories based on $\pi(\theta_k)$
  Estimate $\hat{A}_t^{\pi_k}$
  $\theta \leftarrow$ arg max $L_{\theta_k}^{CLIP}(\theta_k)$
  $k \leftarrow k + 1$
end while
```

2.3. Value-based Algorithms

Value-based algorithms focus on learning the state values for an environment. Given a state s, the state-value function estimates the expected return of s when following a given policy π. The value of taking an action a in state s under π is denoted by $q_\pi(s, a)$. The action-value function over all possible states for π is called q_π. Value-based reinforcement learning algorithms focus on learning the state values for an environment and acting by choosing the best action in a specific state. Note that in this case, active exploration is necessary. Otherwise, an agent possibly does not discover the entire state space.

2.3.1. Deep Q-Networks

Deep Q-Networks (DQN) is a value-based reinforcement learning method introduced by Mnih et al. [31, 33]. It is a gradient descent method based on the concept of Q-learning. Q-learning is a method for small state spaces based on a *Q-table*. During the learning process, an agent builds up a table of state-action values indicating the value of an action a taken in step s. This value is then called *Q-value*. The agent will base its further decisions on Q-values to maximize the expected reward. Computing a table of Q-values is only feasible for small state spaces. In large state spaces, an agent needs to approximate the Q-values by an approximator. As a function approximator for Q-values, the DQN algorithm uses neural networks [31]. A second neural network is used to increase the stability of this approximator. This *target network* is updated periodically [33]. Additionally, as with this method the best action is always chosen, a DQN agent would not explore the entire action space. Therefore, active exploration is necessary, which is achieved by using ϵ-greedy exploration. Given a probability epsilon, a random action is selected instead of the most promising action [31]. DQN uses a memory buffer to replay experienced observations [31]. The general DQN algorithm works as shown in algorithm 3.

Algorithm 3 DQN Pseudocode

```
Initialize the replay memory buffer $D$, the action-value network $Q$ with random weights $\theta$ and the target network $\hat{Q}$ with weights $\theta^{-} = \theta$
while not converged do
  Initialize an episode, set $t = 1$
  for $t = 1, ...$ do
    if prob $< \epsilon$ then
      Choose random action $a_t$
    else
      Choose best action $a_t$ according to $Q$
    end if
    Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
    Save the observation in $D$
    Sample a minibatch from $D$
    $\theta \leftarrow \theta_t + \alpha(r + \gamma\ max_{a'}\ Q(s', a'; \theta_k) - Q(s, a; \theta_k))\nabla_{\theta_k} Q(s, a; \theta_k)$
    After C steps, reset the target network $\hat{Q} = Q$
    $t \leftarrow t + 1$
  end for
end while
```

With the introduction of DQN, reinforcement learning algorithms achieved nearly human-level results in the arcade learning environment [3].

3. Generalization

Commonly, trained RL agents are applied in different environments than they were trained in [48]. Especially in real-life situations, details of the environment or their challenges can differ from those situations the agent experienced during training [25]. It is infeasible to train an agent in all possible scenarios. Hence, agents must transfer their knowledge and insights into similar conditions without the need to be retrained. It is crucial for the agent to understand the underlying task and not get lost in details that are not relevant to the task itself. At the same time, the agent needs to be able to transfer these insights when confronted with new or modified situations. This ability is known as *generalization* [41].

Yet, in reinforcement learning, an agent is often assessed by its training performance [29, 48], i.e., metrics such as game scores or goal reachability are measured during training. The better the score becomes over time, the more successful the agent is. At first sight, this seems a good metric: an agent that gets high rewards and therefore performs well has learned to act successfully in the environment. Yet, several studies show that the training score only indicates the agents' performance in the specific training environment [6, 7, 18, 48]. Even slightly different environments can be challenging for an agent that scored perfect in the training environment. Details such as a changed background color in a game environment can impact an agent's performance [42]. This scenario is known as *overfitting* [41]: the agent learned to solve challenges in its training environment, but the knowledge the agent gained is not general but specific to the training environment. It cannot transfer its skills, i.e., generalize, to similar scenarios. This issue is well known in machine learning, and other machine learning subareas developed evaluation strategies that consider the possibility of overfitting. This chapter investigates existing mechanisms to detect overfitting in reinforcement learning. In section 3.1, we discuss methods used to stochastically modify training environments. Section 3.2 focuses on the importance of splitting training and test environment. Finally, in section 3.3 , we discuss procedurally generated environments.

3.1. Stochastically Modified Environments

To be able to detect overfitted agents, several approaches were introduced to modify the environment during testing time. These approaches have in common that they are still based on the training environment but modify parts so that the environment behaves stochastically. Known methods to achieve this include *sticky actions* [30] and *random starts* [35]. The idea behind sticky actions is to repeat the action chosen by an agent in step $t-1$ instead of performing the action the agent chose in step t by a

specified stickiness factor ζ. Therefore, stochasticity affects the agents' behavior: the agent cannot be sure that its selected action is performed. Random starts put the agent into arbitrary initial states during evaluation time. Therefore, the stochasticity affects the environment's initial setting, but neither the agents' behavior nor aspects later during the episode. Machado et al. propose to use sticky actions during agent evaluation to detect overfitting [30]. And indeed, algorithms that exploit determinism within the environment, e.g., *the Brute* [4] failed in this setting [30]. These algorithms do not manage to adapt to these minor modifications. Yet, Zhang et al. demonstrated that neither sticky actions nor random starts are sufficient to detect or prevent overfitting for more complex reinforcement learning algorithms [48].
Therefore, while being an approach to detect agents that exploit determinism, stochasticity added to the training environment is not the ideal tool to detect overfitted agents in general.

3.2. Splitted Training and Test Environment

As discussed, stochastically modifying the environment does not yield the desired solution. It neither prevents overfitting reliably nor does it guarantee to detect that an agent is overfitted to the environment [48]. Hence, as another approach to detect overfitted agents, the idea to split the training and testing environment arose: an agent is trained in a training environment, and its generalization capability is evaluated in a test environment [29, 47, 48]. With this training and test environment split, we get the *training performance* the agent achieves during training time. Additionally, the agent is tested in a similar but unknown environment, which provides us a *evaluation performance* or *test performance*. Having the additional test performance, we can either look at the test performance directly or compute the *generalization performance* [48]. The generalization performance is the difference between the training performance and test performance. This thesis will consider the generalization performance as suggested by Zhang et al. [47], and Zhang et al. [48]. With this procedure, the agents' training performance, as well as the generalization performance, can be investigated altogether. The training and generalization performance combination gives a good impression of the agents' overall performance. A good training performance with a bad generalization performance means the agent will most likely not be able to solve scenarios it has not explicitly seen during training. Deciding to apply an agent solely based on its training progress can therefore lead to poor decisions [49]. On the other hand, a good generalization performance of an agent with a low training score means the agent will most likely not be able to solve problems in general. Thus, an ideal agent has a high training performance and, at the same time, a good generalization performance. A good training score and a good generalization performance in combination mean that the agent can solve challenges and transfer the knowledge gained during training to new scenarios.

3.3. Call for Procedurally Generated Environments

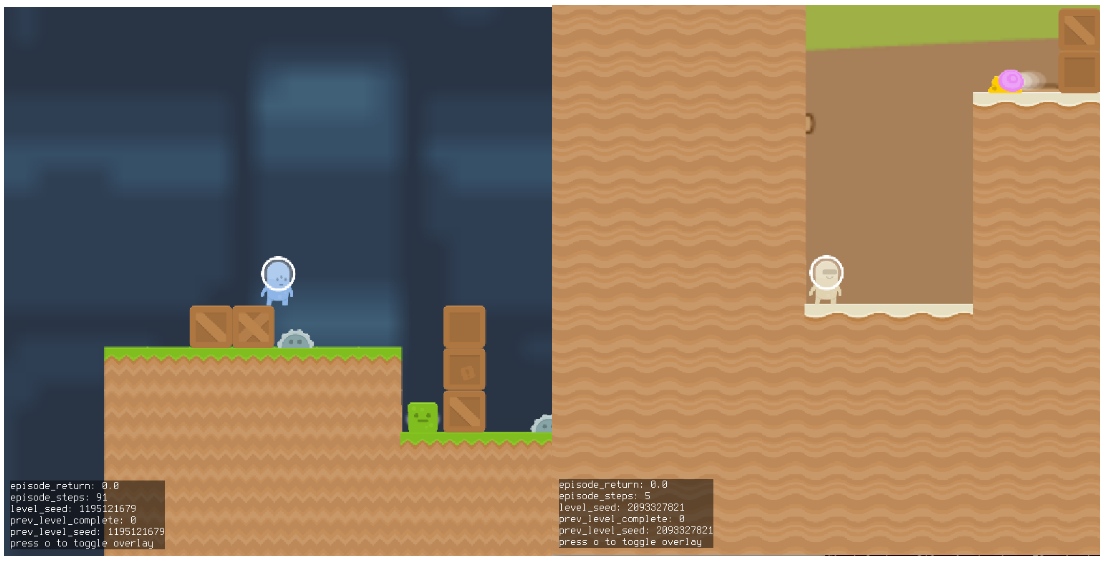
In well-known reinforcement learning environments, e.g., the arcade learning environment (ALE) [3], which is based on the *atari 2600 games*, the levels of an environment are humanly designed. Another option is to generate these instances or levels procedurally. This generation process can affect several aspects of an environment. In ProcGen, it influences parts of the environment like the level layout, game assets, the location of enemies, obstacles, and other relevant objects [6]. A study demonstrated that procedurally generated environments benefit the agents' performance [22]. Besides that, it has been shown that the generalization capabilities of an agent increase with the number of levels it can access during training [6, 7, 47, 48]. Procedural generation offers a way to create a large number of different levels [20] that encourage an agent to generalize insights [6]. This advantage makes benchmarks with handcrafted level design, e.g., ALE, which served as a gold standard for reinforcement learning agents [6], less valuable [6, 22]. It has been demonstrated that agents trained in traditional level setups as typical for ALE, agents' do not develop good generalization capabilities. The generalization capabilities can even be low if the training performances of agents are greatly promising [6]. Therefore, the usage of procedurally generated environments is encouraged by two major aspects: the large variety of levels [6, 48] and the insight that procedurally generated levels help the agent to generalize across levels [22]. Several benchmarks that use procedural generation were introduced in the last years [6, 8, 11, 16, 21, 26, 36, 46]. This thesis will focus on the ProcGen benchmark [6].

4. The ProcGen Benchmark

ProcGen [6] is a reinforcement learning benchmark with 16 unique environments. Each environment has the character of a mini-game, similar to the well-known Arcade Learning Environment (ALE) [3]. In each environment, a reinforcement learning agent takes the role of a player in a single-player game. Observations are $64 \times 64 \times 3$ Red-Green-Blue (RGB) images. The action an agent can perform in each environment is within a discrete action space providing 15 actions per environment. In some environments, no-op actions are added to obtain a common interface with the same number of actions in each environment.

Each environment has a deterministic state transition function. Therefore, being in a state s, and the agent choosing an action a, the outcome of the application of a in s will always be the same state s', with the same reward r. This determinism might lead the agent to learn which action is beneficial in a specific state instead of being able to transfer this to other similar yet different states. However, there is one environment, *chaser*, where the enemy moves (pseudo-)randomly. This (pseudo-)randomness is seed-dependent. For this environment, states that differ by the enemies' position can follow after an identical state s and action a being chosen. Additionally, most environments are partially observable. Therefore, despite having a deterministic state transition function, we can still argue that the partially observable environments are stochastic [23]. The agent does not see the complete state most of the time, so it does not have enough information to determine the state and its upcoming next state. Therefore, although the transition function is deterministic, from the agent's perspective, the environment behaves stochastically. Note that this is true only for environments with partial observability. Some environments, e.g., *heist*, are fully observable in the usual setting.

Instead of providing handcrafted levels in the environment, as in ALE and other RL benchmarks, each level is generated procedurally. Hence, the design of each level is highly seed-dependent, and each environment offers a great variety of different levels. Moreover, as they are generated on the fly, there is no limitation in the number of levels. The unlimited amount of different levels allows agents to see a great number of various instances of the same problem, meaning it has a lot of different situations to fine-tune their knowledge about the environment. However, it also leads to the issue of unsolvable levels. While ProcGen uses solvability constraints during level building to ensure that the generated levels are solvable, there still is a chance that the agent cannot solve some individual levels, i.e., it will eventually fail independently of the actions it chooses. For one of the 16 environments, the agent will even die after the first step in about 7% of the levels [6]. For the other environments, the amount of levels the agent cannot solve is expected to be less than 1% [6].



```
episode_return: 0.0
episode_steps: 91
level_seed: 1195121679
prev_level_complete: 0
prev_level_seed: 1195121679
press o to toggle overlay
```

```
episode_return: 0.0
episode_steps: 5
level_seed: 2093327821
prev_level_complete: 0
prev_level_seed: 2093327821
press o to toggle overlay
```

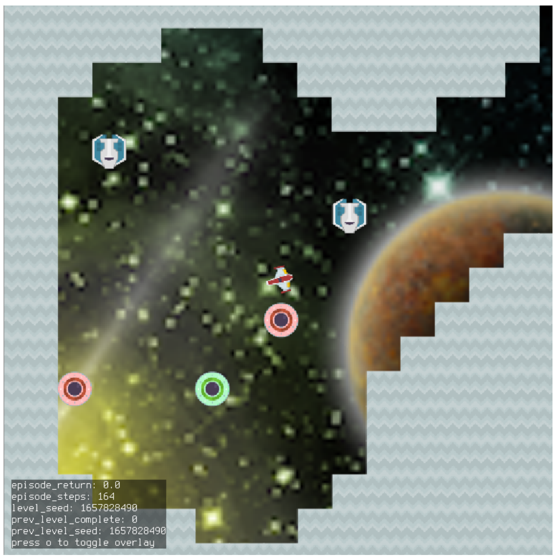
Figure 4.1.: Coinrun

4.1. Procedurally Generated Parts in ProcGen Environments

The procedural level generation impacts several aspects of the environment. Some aspects are relevant for all environments, while others differ between them. We discuss what aspects are part of the procedural generation.

Background and Assets. The graphical representation of environments consists of a background that can change in color and other visual details, as well as assets that determine the look of the player, different categories of enemies, and game objects like boxes, coins. ProcGen provides different versions for each of backgrounds and assets. In Figure 4.1 we can see two levels with different assets. The background and asset versions used in a level are determined during level generation. This leads to a great variety of the graphical aspects of the environment. An agent must quickly Figure out the game character, even if the environment looks different. It also needs to ignore diverse backgrounds and instead focus on the semantic changes in the environment.

Enemies and Objects. Some environments contain enemies the player has to defeat or avoid. In addition to that, some environments can also contain objects like boxes, coins, keys. Their location, whether an enemy is moving or static, and sometimes even their spawn times are part of the levels' procedural generation. Therefore, the agent needs to generalize its knowledge about enemies or objects in way it can use this knowledge in unseen levels with unknown enemy and object positions.



```
episode_return: 0.0
episode_steps: 164
level_seed: 1657828490
prev_level_complete: 0
prev_level_seed: 1657828490
press o to toggle overlay
```

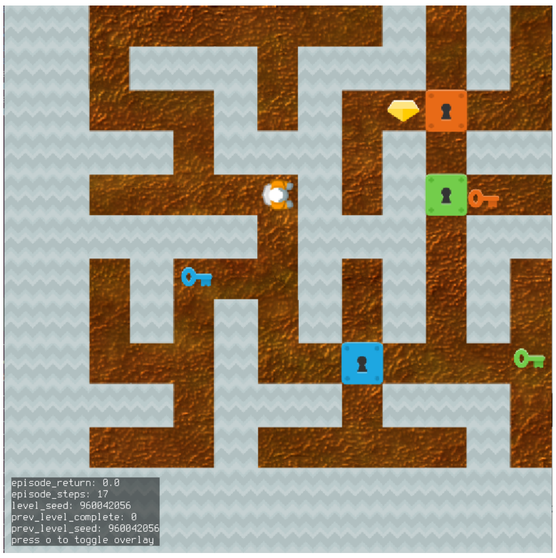
Figure 4.2.: Caveflyer

Platforms. Some game surroundings consist of multiple platforms the agent can walk on and jump between. See Figure 4.1 for an example. Their number, height, and size are part of this generation process. The more platforms a level has, the larger the map becomes.

Cave Maps. Some game surroundings are built with cave-like structures as in Figure 4.2. Using cellular automata [20], these cave structures are generated during the procedural generation process of a level. Therefore, each level has a unique cave, resulting in individual level maps.

Mazes. Similar to caves, mazes as in Figure 4.3, which are part of some game surroundings are generated during the creation process. These mazes are created using Kruskal's algorithm [27] and also look different at each level. This leads to a great variety of different mazes that an agent will have to solve in this benchmark.

Game Constants. Additionally, game constants like the health points of a boss and attacking sequences are part of the procedural generation.



```
episode_return: 0.0
episode_steps: 17
level_seed: 960042056
prev_level_complete: 0
prev_level_seed: 960042056
press o to toggle overlay
```

Figure 4.3.: Heist

4.2. Evaluating Agents on ProcGen

The ProcGen benchmark can be used for agent evaluation. In this section we discuss the different settings that can be used as well as special modes that can be used to evaluate advanced agent skills.

General Evaluation Settings. ProcGen provides different environmental settings for agent evaluation: in the default setting (distribution mode: *hard*), the generated levels are more likely to be harder to solve. Depending on the game, this can influence the map size, the number of enemies, and other aspects of the environment. In this setting, the agent will face challenging situations more often than easy ones. Sometimes, especially if computing capacities are limited, it can be beneficial to train the agent in a less challenging environment. In this case, the distribution mode can be set to *easy*. With this setting, the generated levels are mostly easier to solve. Therefore, the agent learns to solve the levels faster, which decreases the need for computation time. Note that the difference between the distribution modes *hard* and *easy* influences the probability distribution of hard and easy levels, not the level generation in general. In both settings, there will be both, hard and easy levels. Additionally, ProcGen supports distribution modes *exploration* and *memory* for some environments. Both settings provide special levels to investigate the exploration skills or the memory usage of an agent within the environment.

Generalization. In chapter 3, we discussed the importance of an agent's ability to generalize knowledge about the environment to similar yet slightly different situations.

ProcGen provides functionality to evaluate an agent's generalization performance: the agent can be trained on an arbitrarily large but fixed set of levels. During evaluation time, the agent then has to solve levels it did not see during training. This way, we can easily see how the agent performs in unknown situations: if it was able to generalize its insights, it would score similar to the training levels. Yet, if the score during training is high but low during evaluation on new levels, we can conclude that the agent overfits the training level set.

Sample Efficiency. Another essential property of agents is their sample efficiency. It describes how much knowledge agents are able to extract by only a few samples. In the different ProcGen environments, agents face a variety of different challenges they need to solve in order to be successful, either by getting a good score or by solving different levels. We can evaluate an agent's sample efficiency by training it within a challenging amount of time. It then needs to learn fast, which means it has to extract as much information per sample as possible to be successful.

Exploration. In reinforcement learning, the trade-off between exploitation and exploration is an important topic. An agent needs to exploit actions that worked well earlier to perform well in an environment and solve the given tasks. Yet, on the other hand, this hinders the agent from exploring the entire action space. Hence the agent might get stuck with actions from local maxima and not use the action spaces' full potential. A special mode of eight different environments in ProcGen helps to evaluate the exploring behavior of agents. In this mode, the level seed is fixed to generate one specific level. This level is the only one the agent sees during this training, but it is required to explore the complete map to solve this level successfully. Note that generalization performance cannot be evaluated in this mode due to the lack of multiple levels.

Memory. A second special mode makes it possible to evaluate the agents' capability to memorize essential aspects of the environment. In general, environments in ProcGen are designed in a way such that the agent has no or very little memory, and ProcGen's authors state that recurrent and non-recurrent algorithms can score similarly [6]. Yet it can be interesting to investigate how well an agent can utilize memory. Therefore, ProcGen provides an additional distribution mode that makes it possible to evaluate the memory skills of an agent: the memory mode, which is supported for six environments. This mode works by increasing the world size. If not already the case, the environment gets only partially observable. In two environments, *caveflyer* and *jumper*, restrictions to prevent dead-end paths are removed during level generation. These changes require the agent to use memory in order to solve the levels confidentially.

4.3. Game Mechanisms

Although ProcGens' 16 environments are unique games with different challenges, the environments can be grouped according to different aspects. This section discusses the possible groupings we will focus on during our analysis in chapter 6.

Non-friendly vs. Friendly. Most ProcGen environments are not peaceful. In these environments, levels can end early. This can happen if the agent collides with or gets shot by an enemy. Another possible threat to the agent is lethal obstacles. Colliding with these obstacles will also lead to an early end of the level. We use the term *enemy* in this context for *moving* or *shooting* entities that can kill the agent and *lethal obstacle* for *static* objects that lead to the death of a player when colliding with them. In extreme cases, like in *bigfish* or *fruitbot*, only a part of the enemies or obstacles are evil, while colliding with others actually leads to a reward. Additionally, ProcGen provides three friendly environments. In these environments, the agent can only succeed or time out. Table 4.1 lists the environments according to their type.

Game	Enemies	Obstacles	Friendly
bigfish	x	-	-
bossfight	x	x	-
caveflyer	x	x	-
chaser	x	-	-
climber	x	-	-
coinrun	x	x	-
dodgeball	x	x	-
fruitbot	-	x	-
heist	-	-	x
jumper	-	x	-
leaper	x	x	-
maze	-	-	x
miner	-	x	-
ninja	-	x	-
plunder	-	-	x
starpilot	x	x	-

Table 4.1.: Non-friendly and friendly environments in ProcGen

Main Challenges. The different environments provide several different sorts of challenges. We focus on five challenges: Moving to a specified goal, collecting specific objects, opening doors, avoiding enemies and lethal objects, destroying enemies or obstacles and surviving active attacks. The first three challenges are independent of enemies or obstacles and can also hold in friendly environments. They are listed in table 4.2. In contrast, the last three challenges can only occur in non-friendly environments. They are listed in table 4.3.

Game	Move to Goal	Collect	Open
bigfish	-	x	-
bossfight	-	-	-
caveflyer	x	-	-
chaser	-	x	-
climber	-	x	-
coinrun	x	x	-
dodgeball	x	-	-
fruitbot	x	x	x
heist	x	x	x
jumper	x	x	-
leaper	x	-	-
maze	x	x	-
miner	x	x	-
ninja	x	x	-
plunder	-	-	-
starpilot	-	-	-

Table 4.2.: Game challenges, possibly friendly

Surrounding. ProcGen provides different kinds of environmental surroundings. These are usually part of the procedural level generation. Some games like *coinrun* consist of multiple platforms. To be successful, the agent needs to move forward on these platforms. Often it has to be careful about enemies on platforms, or it has to collect objects placed on them. Also, the agent might fall down from a platform, sometimes leading to losing the level. Other games have maze structures in which the agent has to move around, e.g., in *heist*, the agent has to collect keys to open doors. To achieve this, the agent has to move within and explore a level's maze. Another possible environmental structure is the cave. The agent usually has to look for the goal in these caves, such as in *caveflyer*. In some environments, caves contain platforms on which the agent can walk. Other environments have walls that the agent must not walk against but do not have other structures besides these walls. Another possible surrounding is without a concrete structure, e.g., in *bigfish*, there are no platforms or walls. In some of these open surroundings, the agent has to avoid lethal objects, e.g., falling rocks in *miner*. Table 4.4 lists the different surrounding types.

Game	Avoid	Fight	Survive
bigfish	x	-	-
bossfight	x	x	x
caveflyer	x	-	-
chaser	x	-	-
climber	x	-	-
coinrun	x	-	-
dodgeball	x	x	x
fruitbot	x	-	-
heist	-	-	-
jumper	x	-	-
leaper	x	-	-
maze	-	-	-
miner	x	-	-
ninja	x	-	-
plunder	-	x	-
starpilot	x	x	x

Table 4.3.: Game challenges, non-friendly

Platforms	Mazes	Caves	Walls	Open
climber	chaser	caveflyer	dodgeball	bigfish
coinrun	heist	jumper [1]	fruitbot	bossfight
jumper	maze	ninja[1]		leaper
ninja				miner
				plunder
				starpilot

Table 4.4.: Possible surroundings in ProcGen environments

Full vs. Partial Observability. As discussed at the beginning of this chapter, some environments are fully observable, i.e., the agent can observe the entire environmental state. Contrary, some environments are only partially observable, i.e., the agent cannot observe the whole map at a single point in time. The partitioning of the fully and partially observable environments is listed in table 4.5.

Negative Rewards and Penalties. Most games only provide positive rewards. Some of them possibly end when colliding with lethal obstacles or being killed by an enemy. However, there are two exceptions: *fruitbot* provides negative rewards for collecting unhealthy objects instead of fruit. Further, *plunder* provides penalties for shooting

[1]Note that this environment is listed twice because the cave contains platforms. Only on those platforms the agent can move through the cave

Full obs.	Partial obs.
bigfish	caveflyer
bossfight	climber
chaser	coinrun
dodgeball	fruitbot
heist	jumper
leaper	ninja
maze	starpilot
miner	
plunder	

Table 4.5.: Observation spaces in ProcGen

friendly ships instead of enemy ones. This penalty decreases the remaining time to solve the level. Both are interesting categories.

Reward Structures. The reward structure differs between the environments. We detect three different structures that we can group the environments with: very sparse rewards, where the agent only gets a reward when successfully solving a level, and two types of intermediate rewards. In the first intermediate reward category, the agent gets rewarded for one thing, e.g., collecting coins or killing enemies. The other category also provides intermediate rewards, but they are provided for different actions than those that successfully end a level. An example of this category is the game *caveflyer*: the final and most significant reward is provided for finding the goal. However, during the running level, the agent can also collect smaller rewards by killing enemies. The categorization of the reward structure is shown in table 4.6.

Level-end only	Intermediate, identical	Intermediate, different
coinrun	bigfish	caveflyer
heist	bossfight	dodgeball
jumper	chaser	fruitbot
leaper	climber	miner
maze	plunder	
ninja	starpilot	

Table 4.6.: Reward structures in ProcGen

5. Experiments

This chapter explains and shows details regarding the performed experiments. In section 5.1, we discuss the four agents that are trained and evaluated in the experiments. The two different setups of the experiments are discussed in section 5.2. The agents' hyperparameters, as well as how they have been obtained, are presented in section 5.3. Additional information about the reproducibility of the experiments is provided in section 5.4. Finally, we provide an overview of our results in section 5.5. The final analysis of the experiments will take place in chapter 6.

5.1. Agents

This thesis aims to compare multiple deep reinforcement learning algorithms according to their training and generalization performance and classify game mechanisms that the agents solve particularly well or poorly. To this end, four different agents were trained in the experiments based on three algorithms. The following paragraphs discuss the different agents. All agents use the Adam optimizer [24].

***REINFORCE*.** The first agent is an implementation of the episodic REINFORCE algorithm by Williams [45]. The implementation originates from the open-source library ChainerRL [12]. ChainerRL is a deep reinforcement learning library that provides open-source implementations of well-known DRL algorithms to enable reproducible research results. Specifically, we use the implementation of the Preferred Reinforcement Learning (PRFL) library, which is a direct successor of ChainerRL. Instead of Chainer networks [44, 43], as in ChainerRL, PFRL algorithms are compatible with neural networks implemented with PyTorch [38].

***PPO*.** The second agent originates from the ProcGen introductory paper [6]. It is a PPO implementation published in the openAI baseline repository [9]. The implementation is optimized for GPU usage.

DQN^M. For the third agent, we use an implementation of the DQN algorithm provided by Gros et al. [15].

DQN^P. The fourth agent is based on a different DQN implementation. This agents' DQN implementation stems from ChainerRLs' PyTorch library PFRL [12]. While, in theory, the same algorithm should produce identical results, it has been demonstrated that different implementations can differ a lot, presumably by missing details in the publication of the algorithms [17]. Combined with a hint that DQN does not work well on ProcGen [34], it is interesting to investigate this behavior further.

5.1.1. Network

All agents use the same network proposed by Cobbe et al. in the ProcGen introduction [6]. This network was introduced with the IMPALA agent [10] and compared against the NatureCNN network [32]. Experiments demonstrated that the IMPALA network outperforms NatureCNN both in sampling efficiency and generalization performance [6]. Cobbe et al. omit the long short-term memory (LSTM) part of the IMPALA network. The decision is probably based on the fact that ProcGen was designed so that agents do not need much memory. An exception to this is the *memory mode*, as discussed in chapter 4.2. Further experiments showed that on ProcGen, the LSTM part does not improve performance in general. In some environments, it additionally led to unstable training progress [6].

5.2. Setup

We performed the experiments in two different settings for each agent: an *easy* and a *hard* setting. Details of the easy setting are presented in section 5.2.1, and the details for the hard setting are shown in section 5.2.2. In each setting, agents are trained for a specified amount of steps instead of episodes. We decided on this because it matches the training process in the ProcGen introduction paper [6]. Additionally, Machado et al. argue for a step-based setup instead an episodic setup: Often, if an agent performs well, it also takes more steps within one episode. Being allowed to perform more steps means that agents that learn fast have access to more observations than slow learning agents. Making less observations can be a disadvantage for slow learning agents [29]. In each episode, an agent faces one level, which is sampled at the beginning of the episode. The episode ends this level, i.e., either because the agent won the level, it is defeated by an enemy, or the maximal amount of steps per episode are taken. The last value is given by the environment itself. The experiments are run in both settings on all 16 ProcGen environments on three seeds each. The results of the different seeds are then averaged.

5.2.1. Easy Setting

The first setting for our experiments is the *easy setting*. In this setting, each agent is trained for 3 million time steps. During this training phase, the agent has access to 200 different levels. Additionally, the distribution mode of each environment is set to *easy*. The sampled levels are, on average, less challenging than in other distribution modes. Therefore agents are expected to learn faster [6]. The easy distribution mode was introduced to enable users of ProcGen with less computing capabilities to run experiments with fewer resources.

5.2.2. Hard Setting

We refer to the second setting as the *hard setting*. In this setting, each agent is trained for 12 million time steps with 500 differently sampled levels the agent has to solve during the training process. In the hard setting, the distribution mode of each environment is set to *hard*. Cobbe et al. introduced this distribution mode as the default distribution mode [6]. We expect the agents to perform similarly as in the easy setting.

5.2.3. Final Note on the Settings

Cobbe et al. allowed the agent to train for 200 million steps. While this is more than in our setting, this came not with a specific reason but rather because this amount of steps is used frequently on ALE [6]. An additional argument by Cobbe et al. was that their *PPO* agent needed roughly 24 hours for 200 million steps, which they claimed to be an acceptable amount of time [6]. We figured out that our three other agents are much slower in computation time, e.g., the *REINFORCE* agent took roughly one day for 12 million steps on the *heist* environment[1]. Due to limited computing capabilities, we restricted the training steps to 12 million in the hard setting and 3 million in the easy setting. The relation of training steps between easy and hard distribution modes stays the same as in Cobbe et al. [6]. In contrast to the decision of time steps, where the authors followed known ranges, the amount of levels in each distribution mode comes with a reason. During experiments, Cobbe et al. figured out that for their *PPO* agent, this was the number of levels needed to impact the generalization performance [6].

5.2.4. Training and Generalization Performance

For each agent, the training performance is measured as the mean score over the last few thousand steps during training in the environment. Additionally, the generalization

[1]The agent run on a machine with a V100 GPU and an Intel Xeon CPU.

performance, as discussed in chapter 3, is measured. To this end, after a fixed amount of steps, the agent has to solve unknown levels in an evaluation environment. The evaluation performance is tracked like the training performance. We subtract the evaluation scores from the training scores to calculate the generalization performance. While in the evaluation environment, the agent will not continue learning and only access unknown levels.

5.3. Hyperparameter

In this section, we discuss the different hyperparameters of each agent and how we obtained them.

5.3.1. *PPO*

With the publication of ProcGen, Cobbe et al. also published the results of a PPO agent on the benchmark [6], including the hyperparameters tuned by them. Table 5.1 lists the hyperparameters. We stick to these hyperparameters.

Hyperparamater	Value
γ	0.999
λ	0.95
Minibatch size	8
Entropy bonus β	0.01
Clip range	0.2
Reward normalization	yes
Learning rate	5×10^{-4}
Loss coefficient	0.5
Max grad norm	0.5

Table 5.1.: PPO hyperparameter

5.3.2. *REINFORCE*

Due to time and computing capability constraints, we reuse the tuned hyperparameters from the *PPO* agent. The hyperparameter relevant and used for *REINFORCE* are listed in table 5.2.

Hyperparamater	Value
Entropy bonus β	0.01
Learning rate	5×10^{-4}
Max grad norm	0.5
Minibatch size	8

Table 5.2.: REINFORCE hyperparameter

5.3.3. DQN^M

For DQN, we did not have any hyperparameters we could confidentially reuse. Therefore, we tuned the hyperparameters ourselves. Table 5.3 lists the hyperparameters. We performed the hyperparameter tuning on four environments: *coinrun*, *fruitbot*, *heist*, and *starpilot*. We selected four environments that provide different game mechanics and challenges for the agent. We tuned the parameters in the easy setting discussed in section 5.2.1 on three different random seeds per environment. Due to time and compute capacity, we did not tune the full cross product of all combinations of hyperparameters. Instead, we started with a specific parameter, tuned it, fixed this parameter to be the best value, and then tuned the following parameter. The hyperparameter tuning only considers training performance. The evaluation runs in unknown environments are not taken into account.

Hyperparamater	Value
ϵ start	1.0
ϵ end	0.01
ϵ decay	0.9
Learning rate	5×10^{-6}
γ	0.999
τ	0.01
Batch size	32
Buffer size	50000

Table 5.3.: DQN hyperparameter

5.3.4. DQN^P

DQN^M and DQN^P implement the same algorithm. Hence, we use the same hyperparameter for both agents as listed in Table 5.3.

5.3.5. Final Note

The agents based on PFRL implementations have an additional hyperparameter *reward-scale-factor*. It controls the scaling of rewards provided by the environment. It speeds up the learning process of agents [12]. Despite this hyperparameter seeming to have a small impact in the first experiments, we decided to fix it to 1, i.e., omit to scale. We took this decision mainly for two reasons:

1. Usually, the hyperparameters control algorithmic aspects of the agent rather than the environmental part. Especially because in reinforcement learning, the environment is defined as everything that the agent cannot control [41]. It seems a bit counterintuitive to add a hyperparameter for agents that modifies the observation provided by the environment.

2. First experiments hinted that the hyperparameter is different for the agents. While we could have added this additional hyperparameter to the other two agents to make it fairer, the training progress would not be comparable as the results would have different scales.

Especially the combination of these two arguments made us omit the hyperparameter.

5.4. Reproducibility

A common problem in reinforcement learning is that it is hard to understand how researchers achieved their results, even if an agent's hyperparameters are provided. We provide everything necessary in this section to enable our readers to understand how results were achieved.
If not stated otherwise, the experiments were conducted using Python 3.9.13, PyTorch 1.9.1, NumPy 1.22.4, gym 0.15.4, and ProcGen 0.10.7.
The implementations of agents $REINFORCE$, and DQN^P were taken from the ChainerRL library. To this end, we used PFRL version 0.3.0.
The DQN^M agent relies on RLMate [14]. In the experiments, we used RLMate version 0.1.0.
All agents above used a PyTorch implementation of the IMPALA network. The results are GPU-dependent.
To achieve reproducibility on identical GPUs, in the PyTorch backend, the option *deterministic* and *benchmark* for the deep neural network library cuDNN need to be set to $True$ and $False$, respectively. The first option ensures that only deterministic functions are used on the GPU. The second option abstains from optimizing algorithm decisions on the GPU whenever multiple algorithms usually exist. We tried this, and the results were roughly 4x slower, i.e., a single run took roughly 24 hours for 3 million steps instead of roughly 7-8 hours. At the same time, we do not expect significant changes in

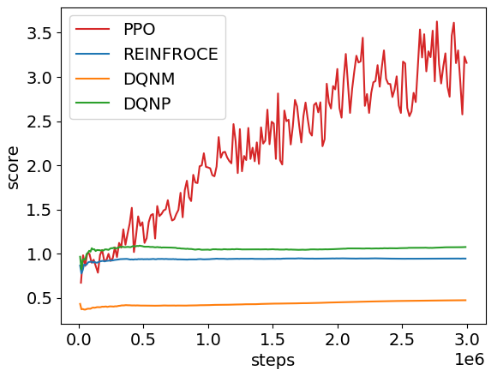
the results. Therefore, for the scope of this thesis, we omit full reproducibility. Due to compatibility issues, the experiments with the *PPO* agent were performed using Python 3.7.3. Besides this, we stuck to the Tensorflow [1] implementation of the IMPALA network, as reprogramming it would have been too complex. To this end, we used Tensorflow 1.15.0.

5.5. Results

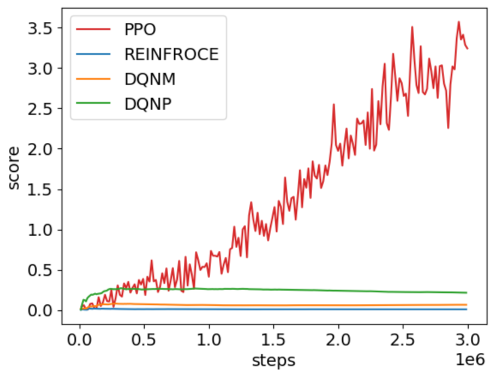
This section presents the results of all four agents in all settings. We show the training and generalization performance in different graphics for better readability. We performed the experiments on three seeds and plotted the average performance of the three runs each. The training and generalization performance for each agent in combination are shown in Appendix A.1 - A.8.

5.5.1. Easy Setting - Training Performance

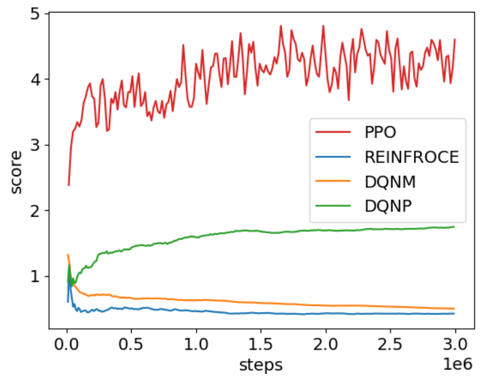
Figure 5.1 shows the training results in the easy setting for all four agents in all 16 environments. The *PPO* agent outperforms all three other agents in all environments. The only exception is the environment *plunder*, where the *REINFORCE* agent performs similarly to the *PPO* agent. Except for the environments *bossfight*, *caveflyer*, and *fruitbot*, all agents outperform the DQN^M agent.



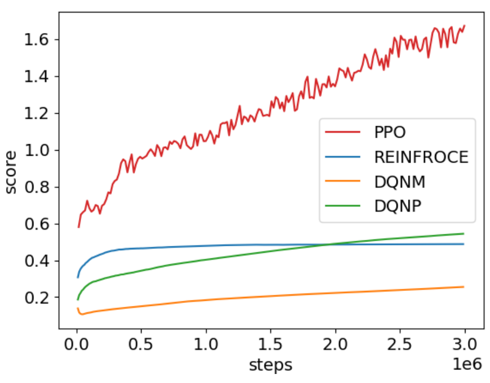
(a) bigfish



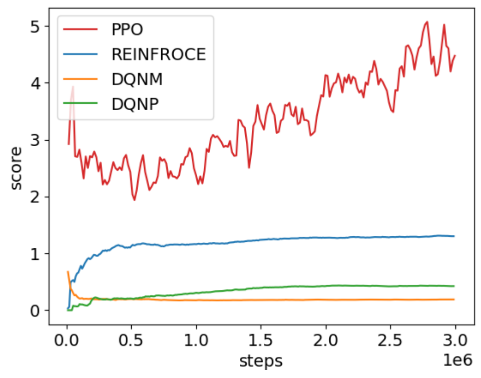
(b) bossfight



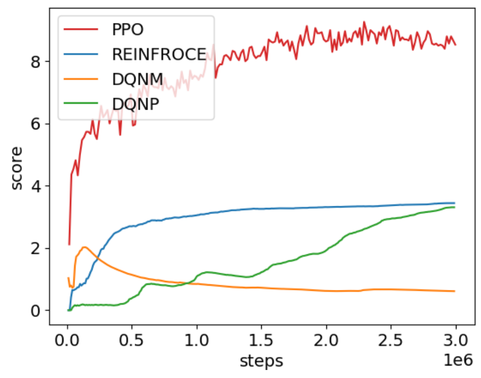
(c) cavefyler



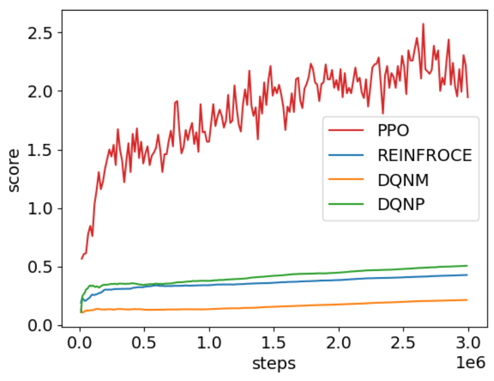
(d) chaser



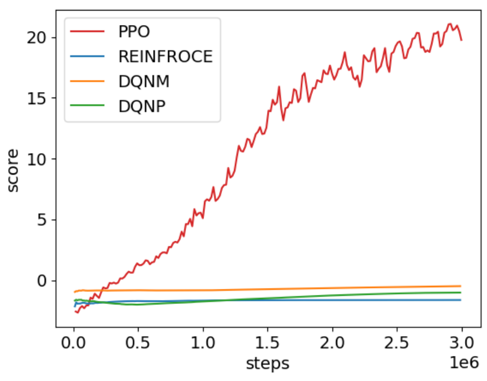
(e) climber



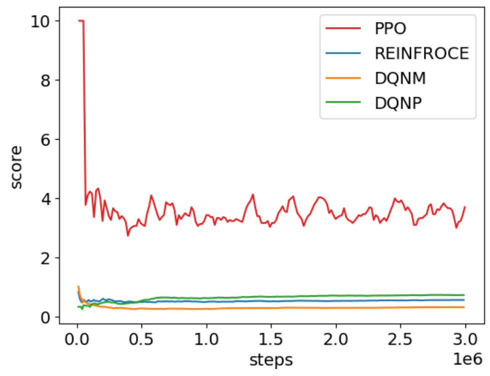
(f) coinrun



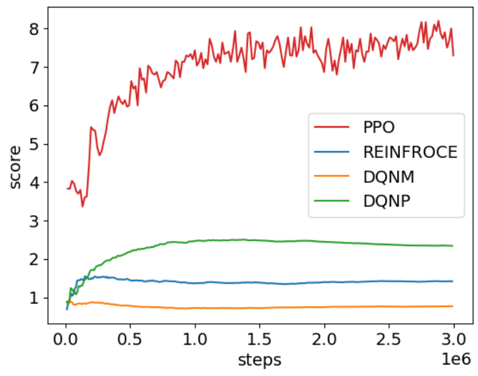
(g) dodgeball



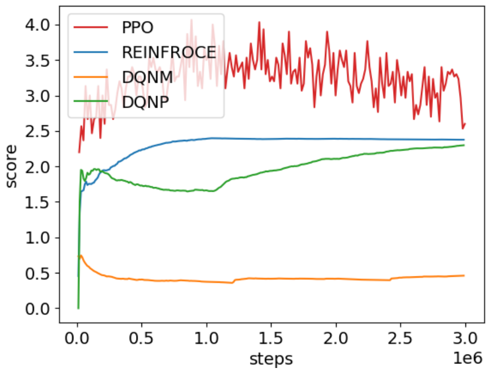
(h) fruitbot



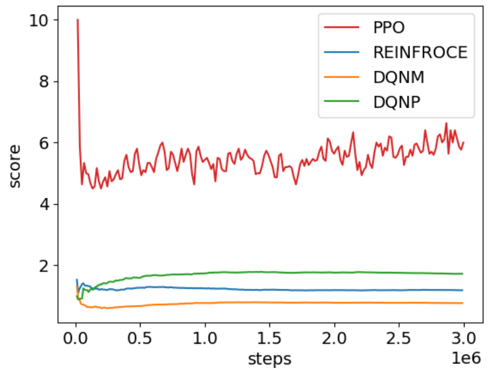
(i) heist



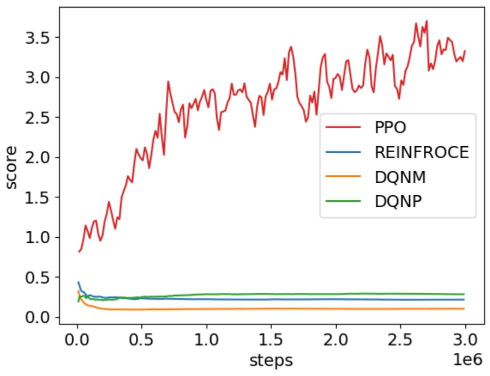
(j) jumper



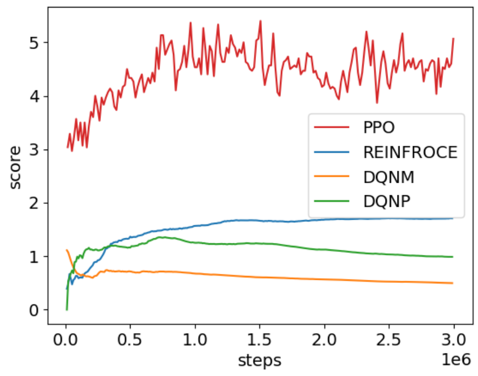
(k) leaper



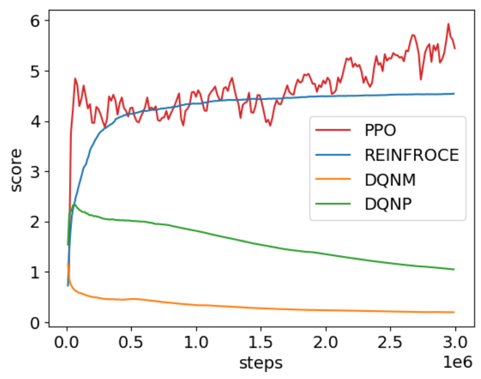
(l) maze



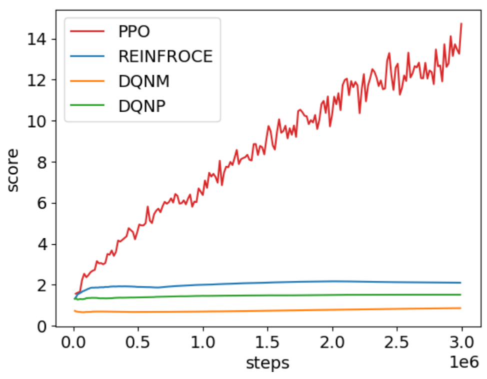
(m) miner



(n) ninja



(o) plunder

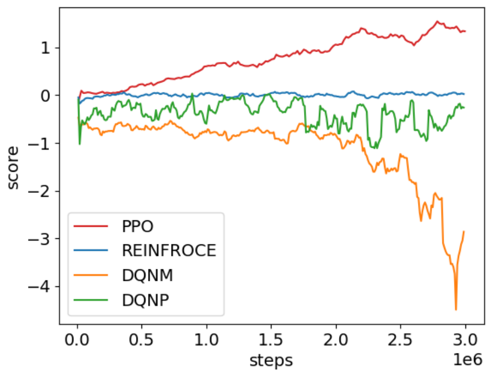


(p) starpilot

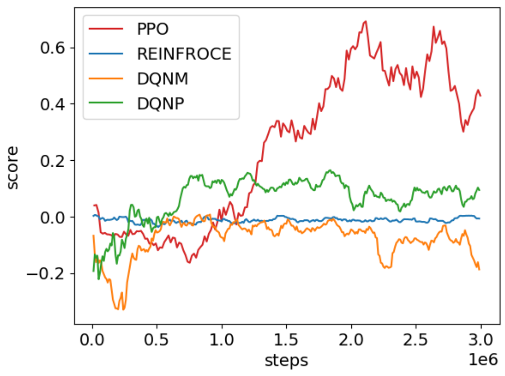
Figure 5.1.: Training performance - easy setting

5.5.2. Easy Setting - Generalization Performance

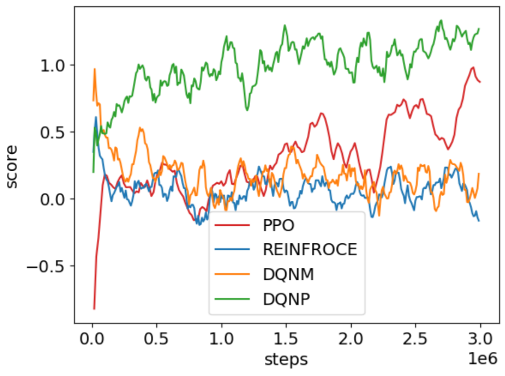
Figure 5.2 shows the generalization performance results in the easy setting for all four agents in all 16 environments. As the generalization performances were a lot more unstable than the training performances, we smoothened the visualization by averaging up to the 10 last results. We see that the *PPO* and DQN^P agents often show the highest generalization performances, whereas the DQN^M and *REINFORCE* agents often show a generalization performance around zero or even a negative performance. The latter indicates that the agent performs better in the evaluation than in the training environment.



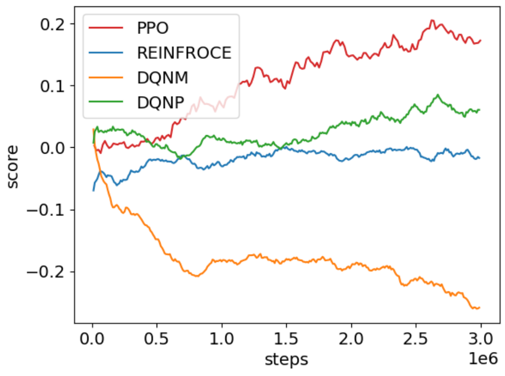
(a) bigfish



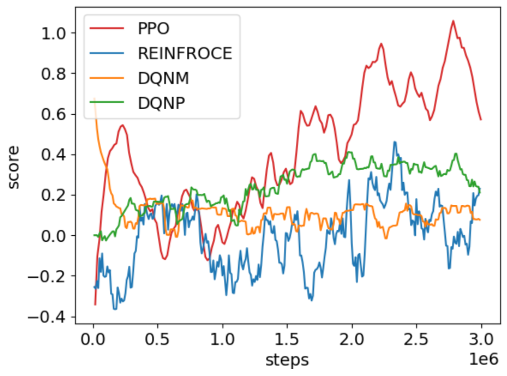
(b) bossfight



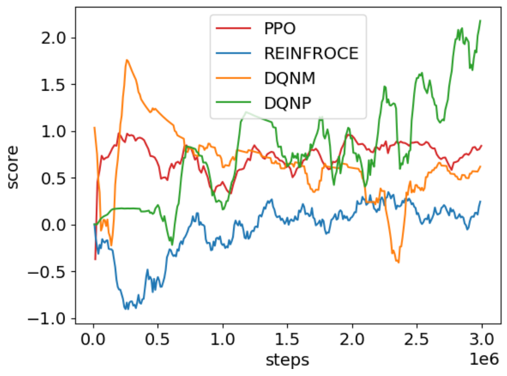
(c) cavefyler



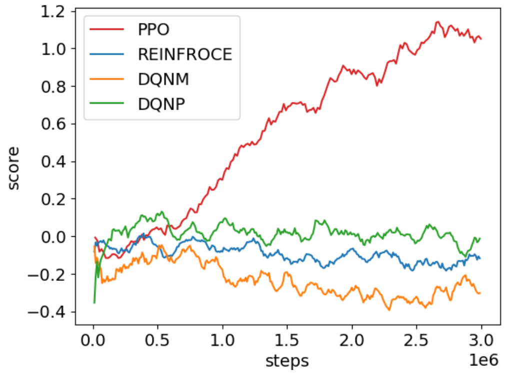
(d) chaser



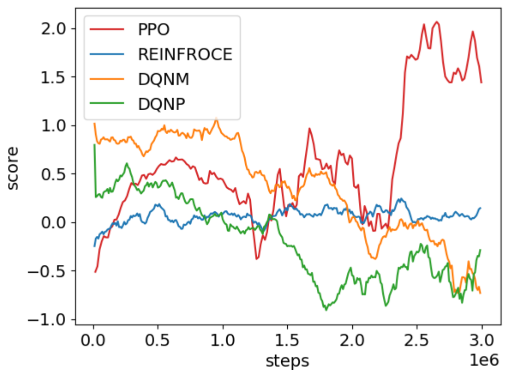
(e) climber



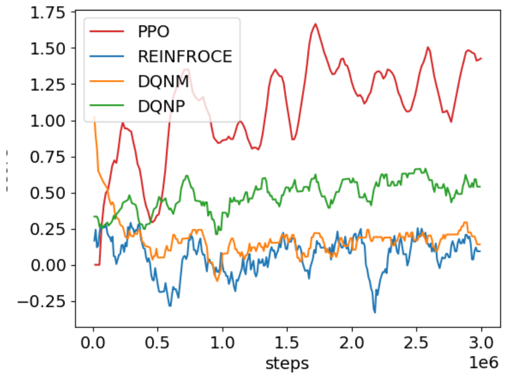
(f) coinrun



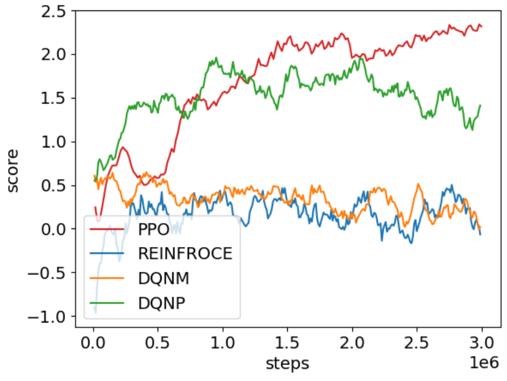
(g) dodgeball



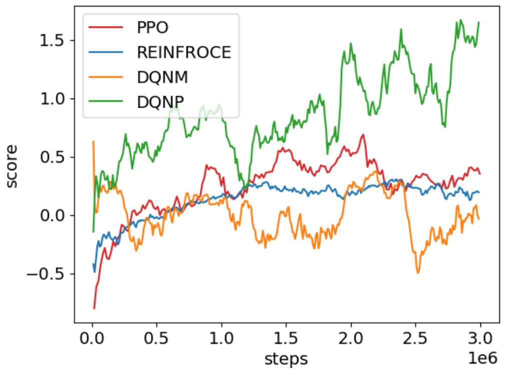
(h) fruitbot



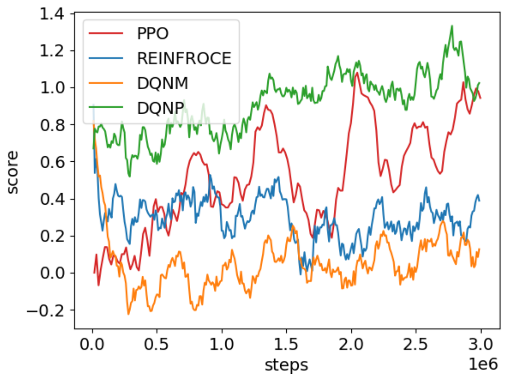
(i) heist



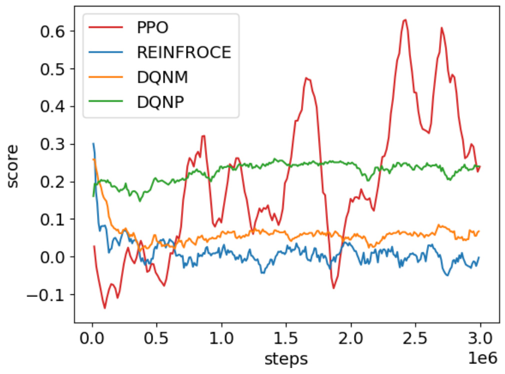
(j) jumper



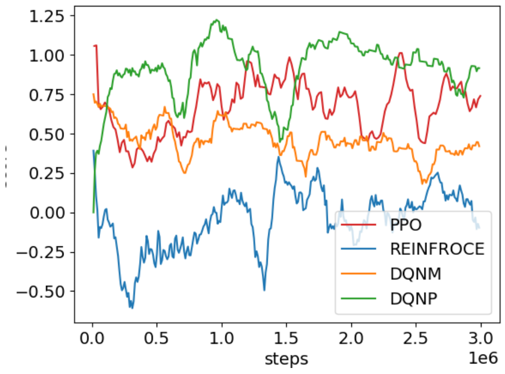
(k) leaper



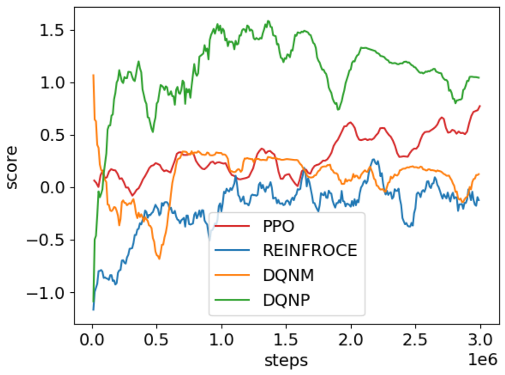
(l) maze



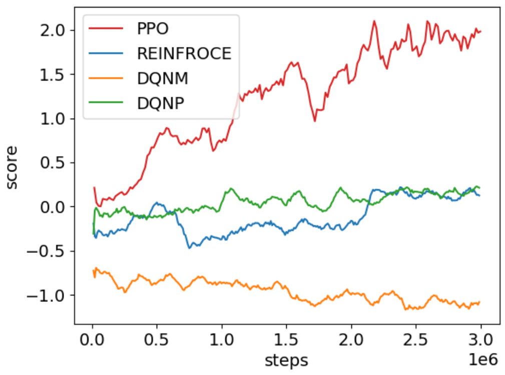
(m) miner



(n) ninja



(o) plunder

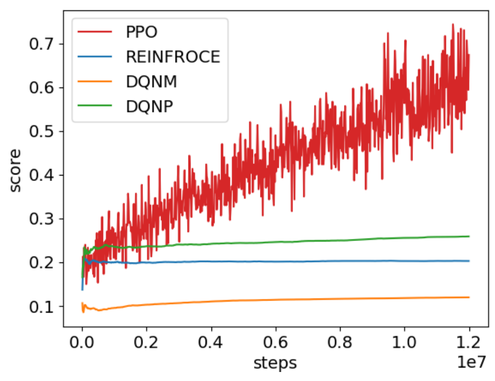


(p) starpilot

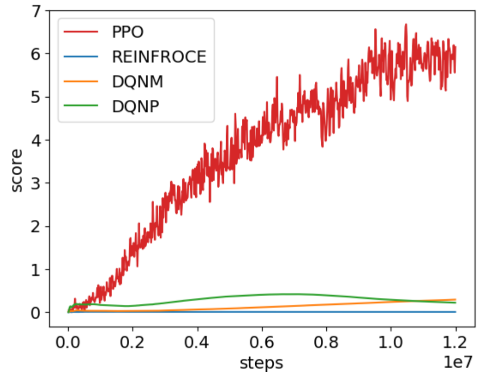
Figure 5.2.: Generalization performance - easy setting

5.5.3. Hard Setting - Training Performance

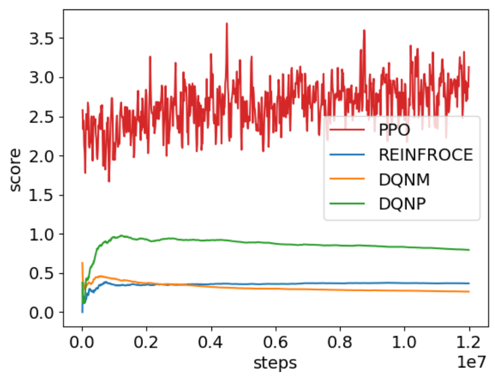
Figure 5.3 shows the training results in the hard setting for all four agents in all 16 environments. The results are pretty similar to the results from the easy setting. Again, in all environments except for *plunder*, *PPO* outperforms the other agents. However, all agents tend to score less.



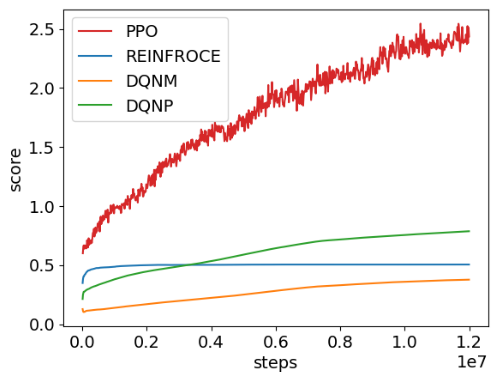
(a) bigfish



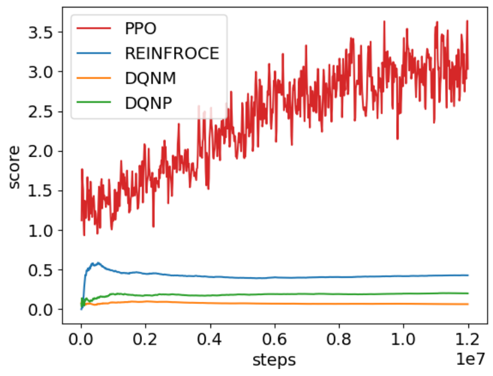
(b) bossfight



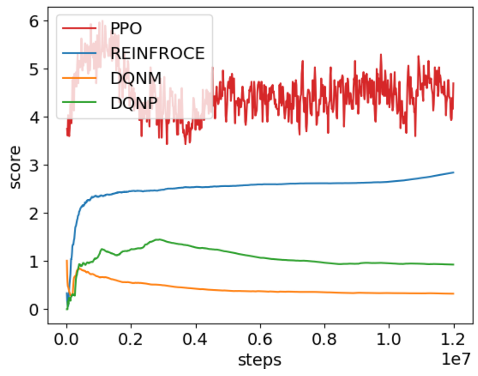
(c) cavefyler



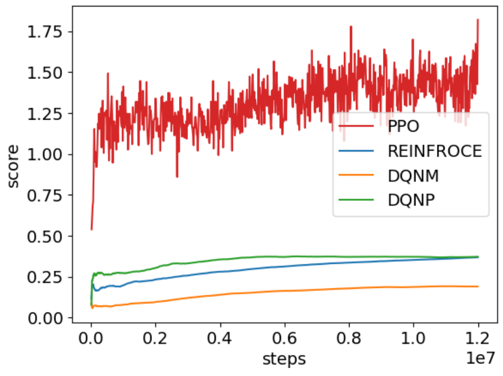
(d) chaser



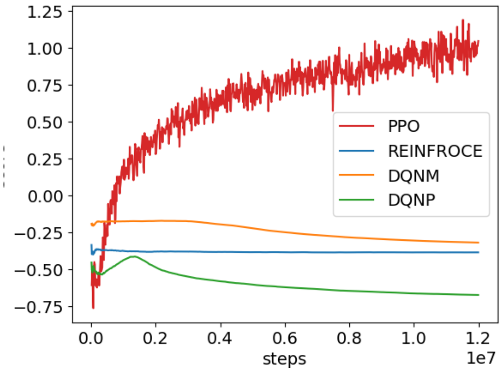
(e) climber



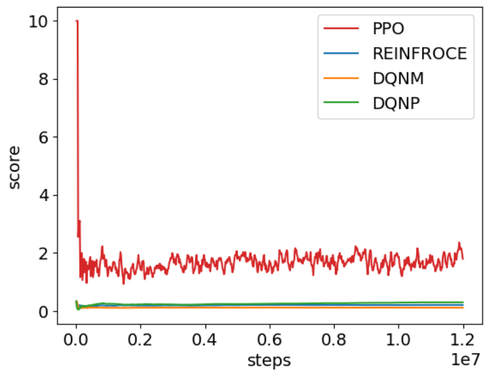
(f) coinrun



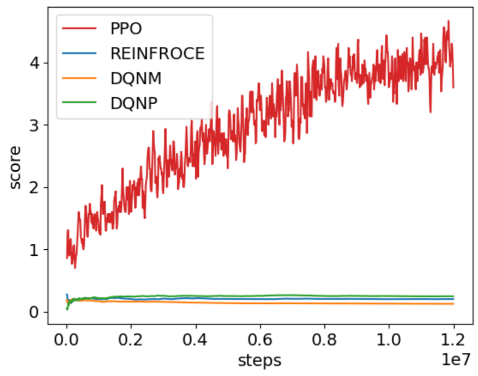
(g) dodgeball



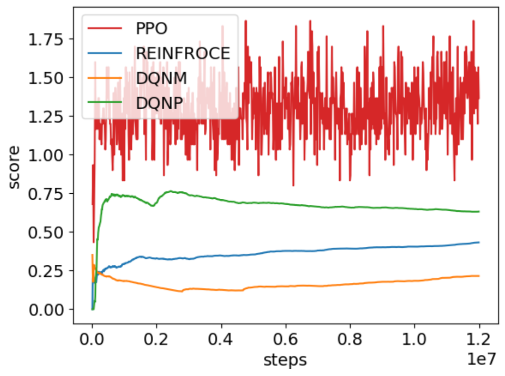
(h) fruitbot



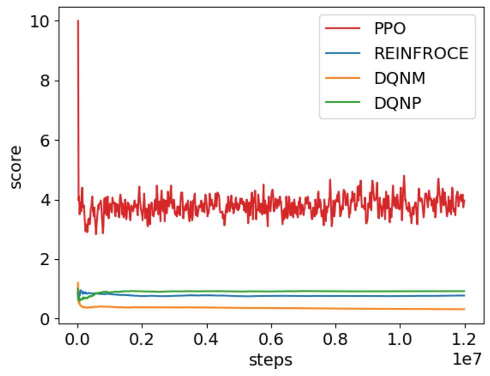
(i) heist



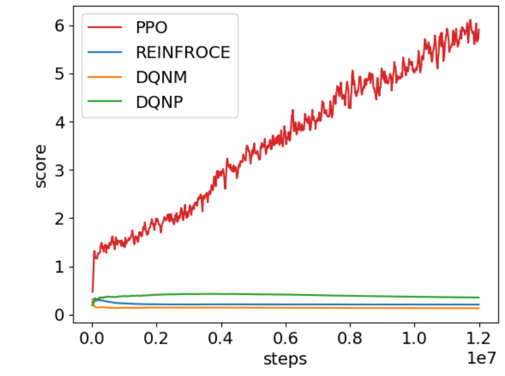
(j) jumper



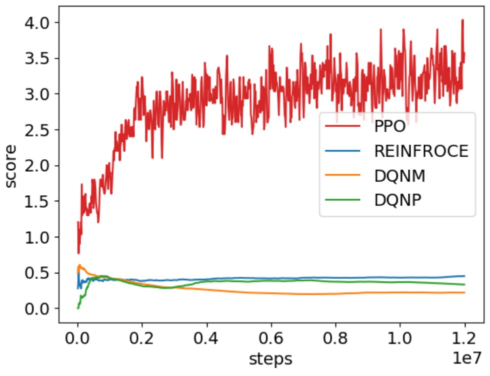
(k) leaper



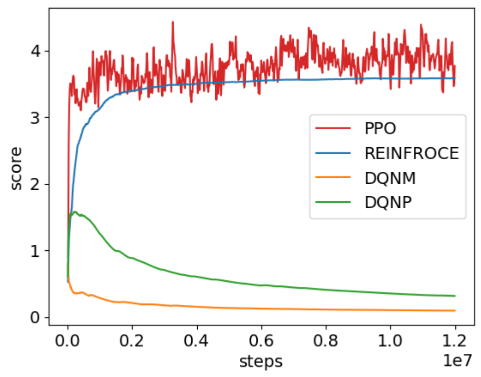
(l) maze



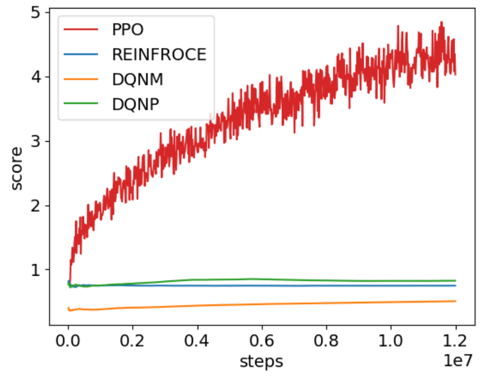
(m) miner



(n) ninja



(o) plunder

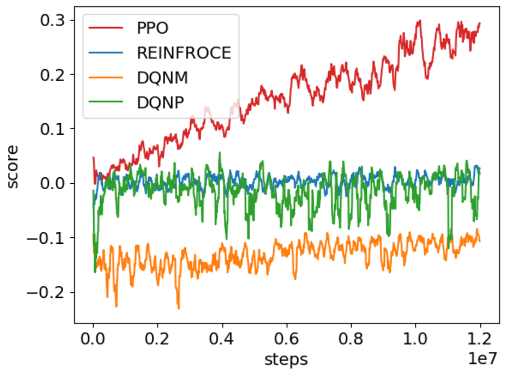


(p) starpilot

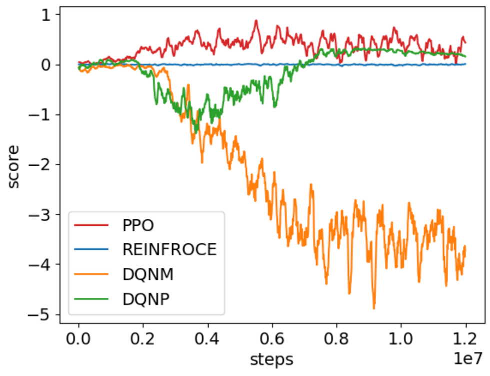
Figure 5.3.: Training performance - hard setting

5.5.4. Hard Setting - Generalization Performance

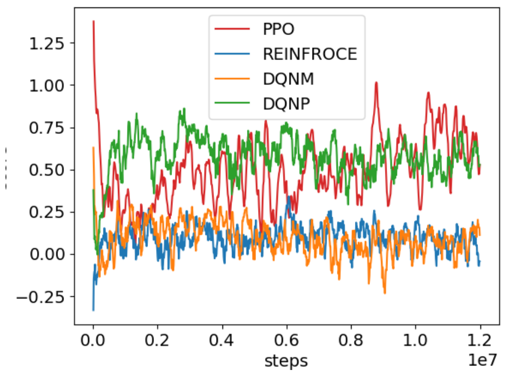
Figure 5.4 shows the generalization performance results in the hard setting for all four agents in all 16 environments. As for the easy setting, we smoothened the results by averaging up to the last 10 results. The generalization performances are similar to the results in the easy setting. However, DQN^P seems to show generalization performances around zero more often.



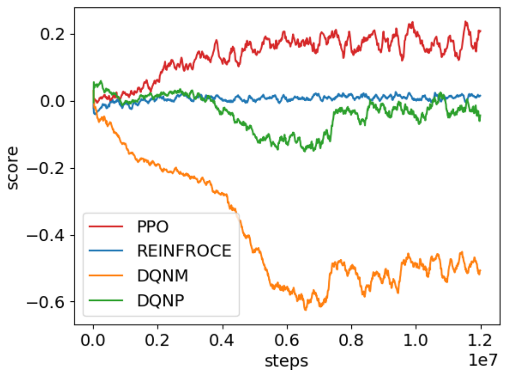
(a) bigfish



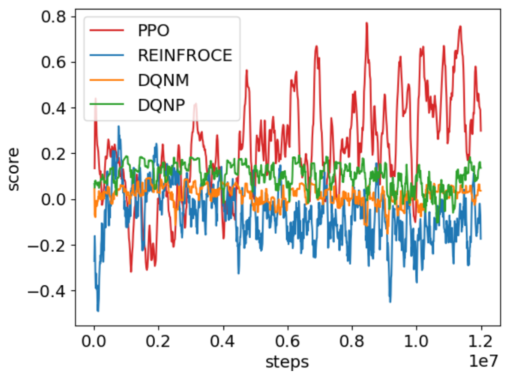
(b) bossfight



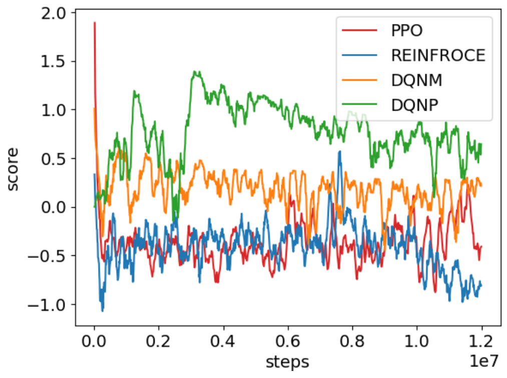
(c) cavefyler



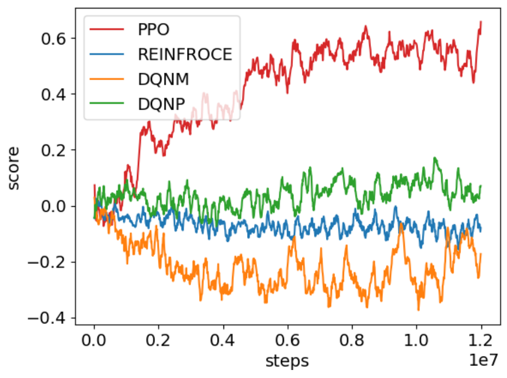
(d) chaser



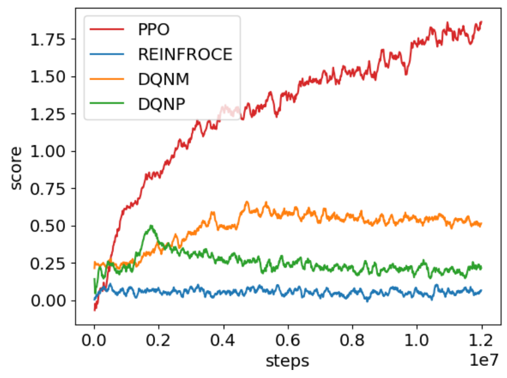
(e) climber



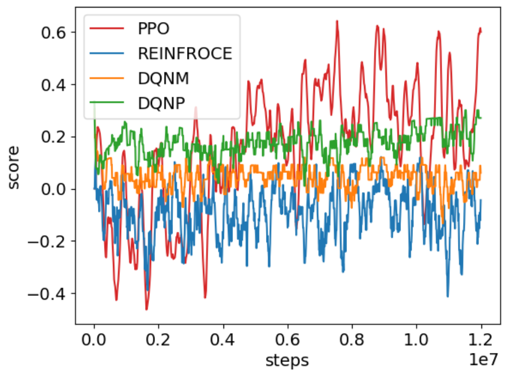
(f) coinrun



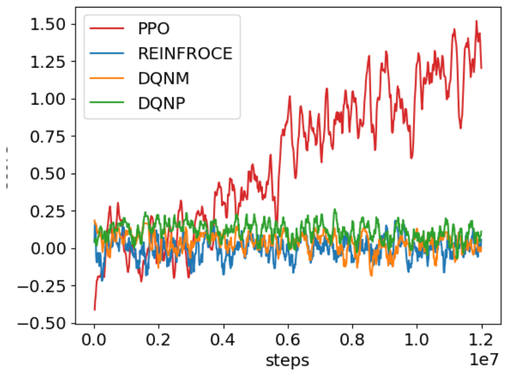
(g) dodgeball



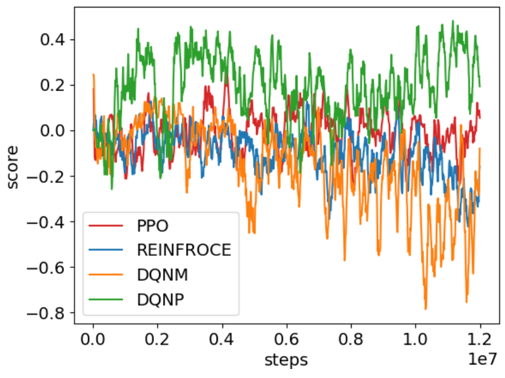
(h) fruitbot



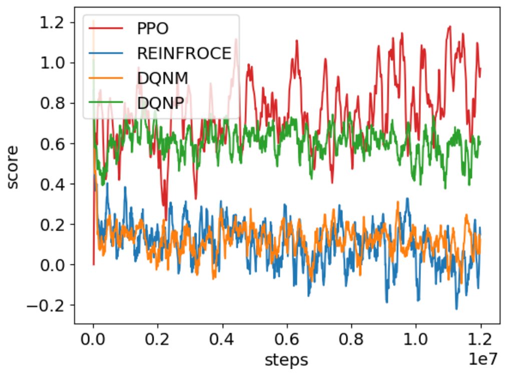
(i) heist



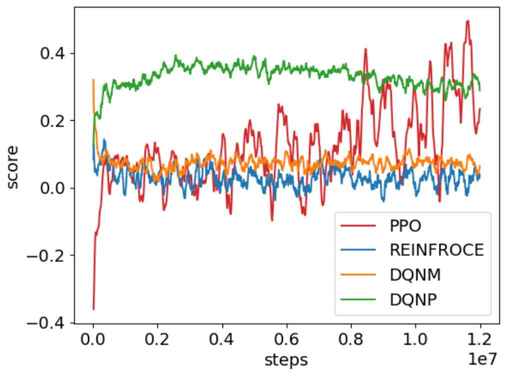
(j) jumper



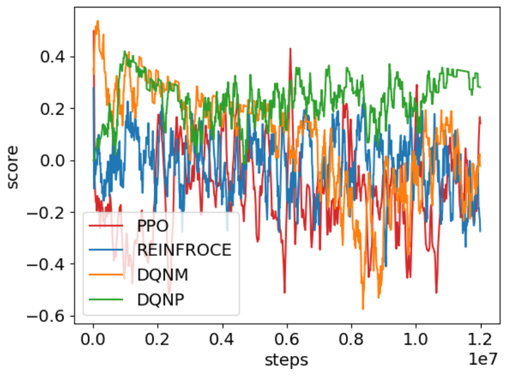
(k) leaper



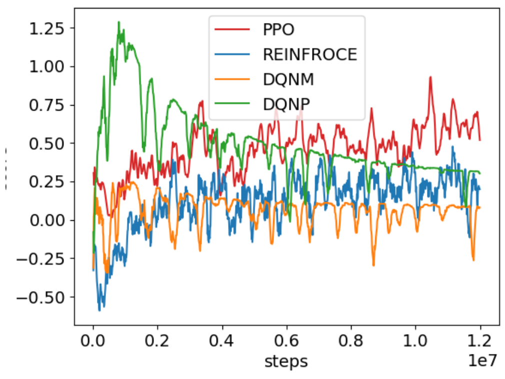
(l) maze



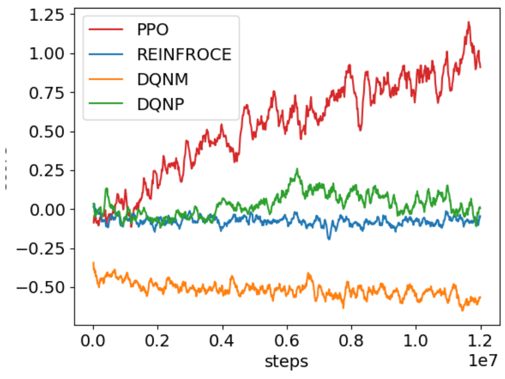
(m) miner



(n) ninja



(o) plunder



(p) starpilot

Figure 5.4.: Generalization performance - hard setting

6. Analysis

In chapter 5, we already discussed the experiments. This chapter analyzes the results under four different aspects. The training performance of the agents is analyzed in section 6.1. The generalization performance of each agent is analyzed in section 6.2. Additionally, we analyze the impact of different game mechanics, as discussed in 4.3 in section 7. Finally, we analyze the influence of the two different settings, as discussed in 5.2 in section 6.3.
The analysis will focus on the easy setting. The only exception is section 6.3, which discusses the differences in performances between the easy and hard settings.

6.1. Training Performance

As a first step of the analysis, we look at the general impression of each agent's training progress. The training performance, as pictured in chapter 5.5, does neither consider the maximal reachable score, nor a trivial score that should be easy to achieve. To better rate the results, we consider these maximal and trivial scores per environment in this part of the analysis. Table 6.1 lists each environment's maximal and trivial scores with an easy distribution mode. We use the values provided by Cobbe et al. [6]. The maximal scores were determined using different ways. For the environments *bigfish*, *bossfight*, *coinrun*, *dodgeball*, *heist*, *jumper*, *leaper*, *maze*, *miner*, *ninja*, and *plunder*, the maximal rewards were computed. For *caveflyer*, *chaser*, and *climber*, the scores were empirically determined. The values for *fruitbot* and *starpilot* were determined by training a *PPO* agent for eight billion time steps [6]. Additionally, we take a look at the trivial scores. These were obtained by training an agent with masked observations [6].

The scores between the four agents often differ greatly. For the training performance analysis, we consider the results depicted in Figure 5.1 in Chapter 5.5.1, and the close-ups for each agent in Appendix A.1, A.3, A.5, and A.7.

6.1.1. *PPO*

In general, the *PPO* agent shows good learning progresses across environments. Especially in *bigfish*, *bossfight*, *fruitbot*, and *starpilot*, the training performance increases significantly with more training steps. A bit less impressive, either due to the learning curve itself or the amount of score progress, but still good and noticeable progress is visible in *chaser*, *climber*, *coinrun*, *jumper*, and *miner*. In the environments *dodgeball* and *plunder*, there is some but no impressive increase in training performance over time.

Environment	Max Score	Trivial Score
bigfish	40	1
bossfight	13	0.5
caveflyer	12	3.5
chaser	13	0.5
climber	12.6	2
coinrun	10	5
dodgeball	19	1.5
fruitbot	32.4	-1.5
heist	10	3.5
jumper	10	3
leaper	10	3
maze	10	5
miner	13	1.5
ninja	10	3.5
plunder	30	4.5
starpilot	64	2.5

Table 6.1.: Maximal and trivial scores for each ProcGen environment with easy distribution mode

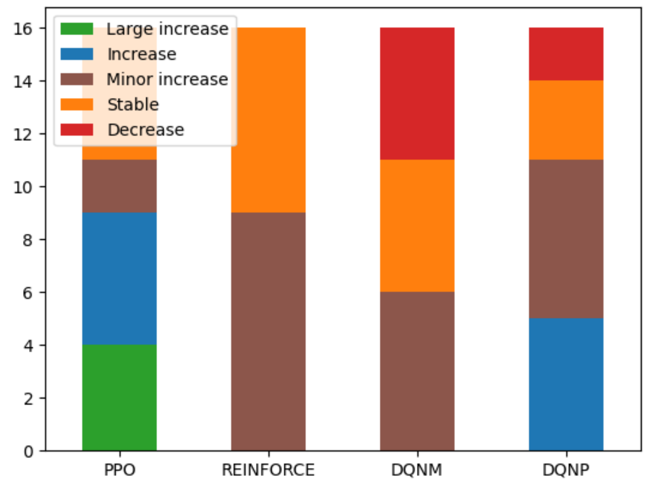
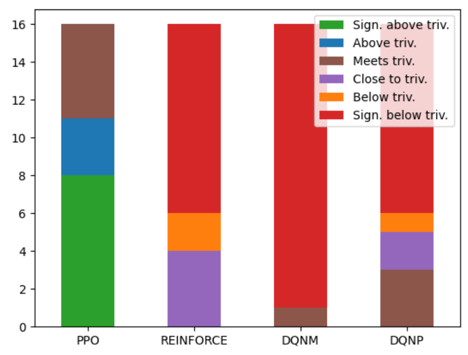
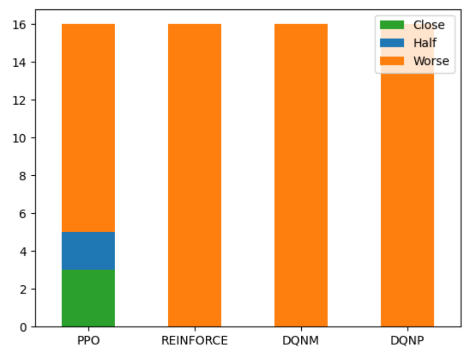


Figure 6.1.: Training progress



(a) Trivial Score



(b) Maximal Score

Figure 6.2.: Scores compared to trivial and maximal score

The training performance is relatively constant in *caveflyer*, *heist*, *maze*, *ninja*, and *leaper*. The correspoding statistics is depicted in Figure 6.1.
As shown in Figure 6.2, the *PPO* agent meets the trivial scores in all environments. In *caveflyer*, *miner*, and *ninja*, the training score even gets visibly better than the trivial score. The training scores for *bigfish*, *bossfight*, *chaser*, *starpilot*, *climber*, and *jumper* are even better, compared to their trivial score. Very well is the score for *fruitbot*. In *coinrun*, the score even becomes close to the maximal achievable score. However, compared to the maximal achievable scores, the agent's performance is only significant in *coinrun*, *fruitbot*, and *jumper*. In *maze* and *ninja*, the agent achieves at least roughly half of the possible score. A summary is provided by Figure 6.2.

6.1.2. *REINFORCE*

In general, as visible in Figure 6.1, the *REINFORCE* agent's training progress is much more limited. In some environments, the progress is utterly stable after the first few thousand steps, with no or minimal improvements in training scores. The agent shows performance progress in the environments *chaser*, *climber*, *coinrun*, *dodgeball*, *fruitbot*, *leaper*, *ninja*, *plunder*, and *starpilot*. It is pretty constant for all seven other environments.
In none of the environments, the agent meets the trivial score. However, for *bigfish*, *chaser*, *fruitbot*, and *plunder*, the *REINFORCE* agent nearly meets the trivial score. A bit further from the trivial score is the agent in the environments *leaper* and *starpilot*. In all other ten environments, the agents' training performance is significantly below the trivial score. Nevertheless, for *climber*, *coinrun*, *dodgeball*, and *ninja*, it seems the agent is still learning, but slowly. In the other environments, it possibly got stuck in a locally optimal policy. These results are concluded in Figure 6.2.

6.1.3. DQN^M

In general, the training progress of the DQN^M agent does not seem very promising. The achieved scores remain pretty constant from the scores from the first few thousand steps. In the environments *caveflyer*, *climber*, *coinrun*, *ninja*, and *plunder*, the performance even decreases during the training process. However, some training progress is visible in *chaser*, *dodgeball*, *fruitbot*, *leaper*, *maze* and *starpilot*. Although the improvements are minor, with more training time, the agent could become better. In *fruitbot*, the agent even outperforms the *REINFORCE* and DQN^P agent. These results are summarized in Figure 6.1. Additionally, the agent meets the trivial score in *fruitbot*. However, the agent does not exceed the trivial score often. In all other environments, its score stays below the trivial score, as visualized in Figure 6.2.

6.1.4. DQN^P

The DQN^P agent shows promising training progress for several environments. This is the case for the environments *caveflyer*, *coinrun*, *fruitbot*, *maze*, and *jumper*. The agent improves slightly during the training process in *chaser*, *climber*, *dodgeball*, *heist*, *leaper*, and *miner*. At the same time, the agents' performance is relatively stable in *bigfish*, *bossfight*, and *starpilot*. However, in *ninja* and *plunder*, the training performance even decreases. Results are summarized in Figure 6.1.
The agent can meet the trivial score in the environments *bigfish*, *chaser*, and *fruitbot*. In *bossfight* and *leaper*, it nearly meets the trivial score. In *leaper*, the learning process visibly continues, so with more training time, it would probably meet the trivial score. Not too far from the trivial score is the agents' performance in *starpilot*. In all other environments, the agent misses the trivial score significantly, as depicted in Figure 6.2.

6.1.5. Conclusion

Undoubtedly, the *PPO* agent outperforms all other agents' training progresses significantly. The only exception is the environment *plunder*, where the *PPO* and the *REINFORCE* agent reach a similar training score. Difficulties for DQN agents in ProcGen have already been observed earlier [6, 34].

In *plunder*, *PPO* and *REINFORCE* reach the trivial score. Interestingly, *plunder* is, together with *climber*, one of the only environments where the policy based agents clearly outperform the value based agents during the full training progress. *PPO* reaches the trivial score in all other environments. This is no surprise, as the trivial score is based on a *PPO* agent trained with the observations being masked out [6]. In many environments, *PPO* even exceeds the trivial score. In *coinrun*, it even gets close to the optimal score. The other agents often have trouble reaching even the trivial score.

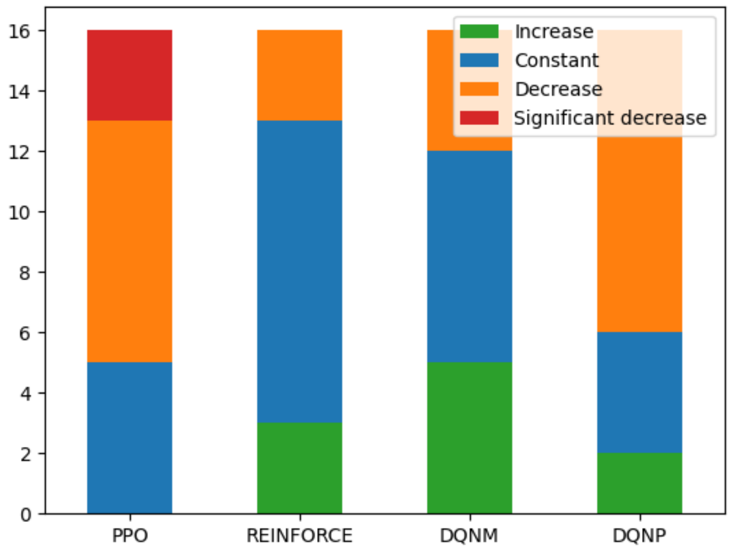


Figure 6.3.: Evaluation performance

However, only considering the training progress without comparing their reached scores, we see similarities between the agents. All agents show some training progress in *chaser*, *dodgeball*, and *fruitbot*. In *chaser* and *fruitbot*, all agents reach or nearly reach the trivial score, *PPO* even gets a quite good overall score. However, in *dodgeball*, no agent reaches impressive results. Three agents showed training progress in *climber*, *coinrun*, *leaper*, and *starpilot*.
Interestingly, DQN^M and DQN^P had different tendencies in three environments: *caveflyer*, *climber*, and *coinrun*. Here, DQN^P showed training progress, whereas DQN^M even showed negative training progress. This strengthens the observations made by Henderson et al., which indicate that implementation of the same algorithm and using identical hyperparameters, as discussed in chapter 5.3.4, can still lead to different results [17]. In the environments *caveflyer*, *heist*, and *ninja*, at most one agent showed noticeable training progress. In *fruitbot*, all agents either reached a good score (*PPO*) or (nearly) met the trivial score (*REINFORCE*, DQN^M, DQN^P). Except for DQN^M, the same holds in the environments *bigfish*, *chaser*, and *pilot*.

6.2. Generalization Performance

This section analyzes the agents' generalization performance. To this end, we consider the formula discussed in chapter 3.2, namely the difference between training performance and evaluation performance. Therefore, the generalization performance can maximally

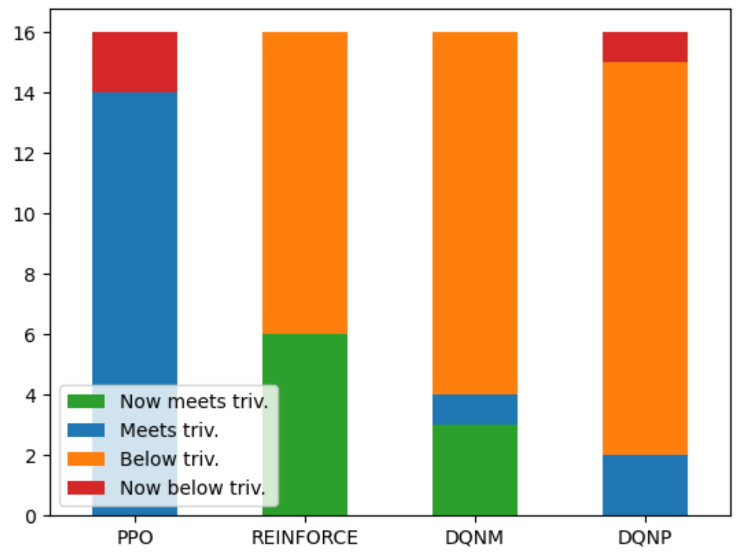


Figure 6.4.: Evaluation scores compared to trivial score

be the same as the training score, indicating that an agent does not yield rewards in the evaluation environment. On the other hand, a negative generalization performance indicates that an agent performs better in the evaluation environment than in the training environment. An exception to this is the environment *fruitbot*, where negative rewards are possible. Ideally, an agent has a generalization performance around zero. This means the agent performs equally in the training and evaluation environment.
For the generalization performance analysis, we consider the results depicted in Figure 5.2 in chapter 5.5.2, and the close-ups for each agent in Appendix A.1, A.3, A.5, and A.7. Often the visualizations in the appendix allow deeper insights, as we can see the generalization performance compared to the agents' training performance. We argue that the same generalization performance weights less in an environment where the agent achieves high values. In contrast, it is more dramatic in environments with a lower performance in general.

6.2.1. *PPO*

In general, the training progress and the evaluation progress look quite similar. However, compared to the progress during training, in the environments *bigfish*, *dodgeball*, and *jumper*, the generalization performance increases visibly, i.e., the performance in the evaluation environments do not keep up with the learning progress in the corresponding training environment. The agents' generalization performance increases slightly in the environments *bossfight*, *caveflyer*, *chaser*, *heist*, *maze*, *ninja*, *plunder*, and *starpilot*. The

generalization performance is roughly but not wholly zero in the other environments, i.e, the agent scores similar in training and evaluation. In general, the generalization performance of the *PPO* agent is non-negative, i.e., the training performance is always better or equal to the evaluation performance, as visible in Figure 6.3.
With the proceeding training progress, the generalization performance increases in nearly all environments, which means the agent cannot transfer all its knowledge from the training environment to unknown levels.
For nearly all environments, the agent still meets the trivial scores, as summarized in Figure 6.4. However, in *bigfish* and *jumper*, the performance gets significantly closer to the trivial score. In *dodgeball* and *heist*, the agent cannot meet the trivial score in the evaluation environment.

6.2.2. *REINFORCE*

In general, the generalization performance of the *REINFORCE* agent is roughly around zero, with a tendency towards negative values. Hence, the agent performs equally or better in the evaluation environment. Especially in environments where the agent showed training progress, the agent achieves a promising generalization performance.
In the environments *dodgeball*, *plunder*, and *starpilot*, the evaluation performance exceeds the training performance. Contrary, in *fruitbot*, *leaper*, and *maze*, the evaluation performance is visibly worse than the training performance. In all other environments, the performances are nearly identical. This indicates that the learned policy is stable even at unknown levels. A summary can be found in Figure 6.3. However, as *REINFORCE* did not score too well in training, a generalization performance of roughly zero or slightly below zero also indicates that the agent does not score too well in general. Yet, in the environments *bigfish*, *chaser*, *climber*, *fruitbot*, *plunder*, and *starpilot*, the agent achieves the trivial score in evaluation environment in several levels. Figure 6.4 summarizes these insights.

6.2.3. DQN^M

As depicted in Figure 6.3, the generalization performance of the DQN^M agent has a tendency towards a negative value. This means the agent performs better in the evaluation environment than in the training environment. This is surprising, as it does not learn in those environments. Especially as there is seldomly really noticeable training progress, we are astonished by these results. Especially in *bigfish*, *chaser*, *dodgeball*, *fruitbot*, and *starpilot*, the evaluation performance progresses with training time. In *fruitbot*, the evaluation performance increases even significantly with proceeding training. The evaluation performance is visibly worse in *coinrun*, *jumper*, *miner*, and *ninja*.
While the training performance only meets the trivial score in the environment *fruitbot*, the agent is more successful in the evaluation environment. Here, it meets the trivial score several times in *bigfish*, *bossfight*, and *fruitbot*. In *caveflyer*, the agent even learns

to meet the trivial score regularly towards the end of the training process. In *starpilot* only slightly misses the trivial score. A summary of the trivial scores in generalization is depicted in Figure 6.4

6.2.4. DQN^P

In general, the generalization performance of the DQN^P agent is not as good as that of the DQN^M and *REINFORCE* agents. The agents' generalization performance is summarized in Figure 6.3.
In *bigfish*, the evaluation performance is better than the training performance. In *fruitbot*, the evaluation performance exceeds the training performance after half of the training progress. The performances are similar to the training performance in *chaser*, *dodgeball*, *fruitbot*, and *starpilot*. However, in *chaser*, the agent meets the trivial score in training but not in the evaluation environment. Figure 6.4 shows the summary of trivial scores in the generalization environments. In *caveflyer*, *climber*, *coinrun*, *heist*, *jumper*, *leaper*, *maze*, *miner*, *ninja*, and *plunder*, the evaluation performance is visibly worse, i.e., the agent does not generalize well or at all in those environments.

6.2.5. Conclusion

The generalization performance of *PPO* and the other three agents is hard to compare. This is because *PPO* achieves solid scores, whereas the other agents mostly only score minimally. By the definition of the generalization performance, *REINFORCE* and both DQN agents cannot get a high, i.e., undesirable, generalization performance. The risk for *PPO* to achieve a worse generalization performance is much higher, as the training performance is much better. However, we try to analyze each agent's generalization performance. To this end, we focus on the performances between the environments for each agent.

PPO shows a decreased evaluation performance in most environments but with varying impacts. In contrast, *REINFORCE*, and especially DQN^M, show a tendency towards a negative generalization performance. This means the agents perform better in the evaluation environment than in the training environment. A possible explanation for this might be that these agents learning progress improves in only some of the training levels, but not enough to improve the overall training score. In the evaluation, where many more levels are sampled, these insights might turn out to be beneficial. Similar to *PPO*, the DQN^P agent shows a tendency toward a decreased evaluation performance. However, the evaluation performance increases compared to the training performance in two environments. In these environments, the evaluation performance is also increased for DQN^M. However, it is interesting that the generalization behavior of both DQN agents differs greatly. This is yet another hint supporting the observation by Henderson et al. [17].

In *jumper*, *maze*, and *ninja*, three agents show an increasing generalization performance. The evaluation performance of the *PPO* agent decreases close to the trivial score, yet meeting it in two environments. In another two environments, it does not meet the trivial score in the evaluation environment anymore. The DQN^P agent misses a trivial score that was met in training environments. Surprisingly, *REINFORCE* and DQN^M can meet trivial scores in the evaluation environment that they do not meet during training.

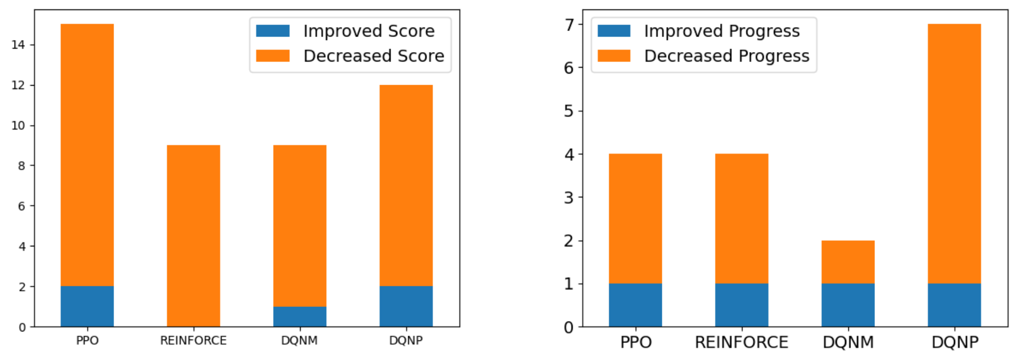
Although *PPO* mostly shows a worse generalization performance than the other agents, its scores in the evaluation environment are still higher. Therefore, when assessing the performances of agents, it is essential to consider both the evaluation performance as well as the generalization performance. Finally, it needs to be mentioned that the number of levels in the training environment was determined based on the performance of a *PPO* agent [6]. Whether other algorithms need more or fewer levels until the generalization gap closes needs further investigation.

6.3. Easy vs. Hard Setting

As discussed in chapter 5.2, the experiments were performed in two different settings. The main difference between the settings is the distribution mode, which is set to easy in one setting and hard in the other, i.e., in the easy setting, the sampled levels are easier to solve in general, whereas in the hard setting, they are harder to solve, on average. The number of steps and training levels is adjusted to the distribution mode. The idea behind the different modes is to provide a mode that requires fewer computing resources [6]. This section inspects the results in both settings. We compare the training performances from Figures 5.1 and 5.3 and generalization performances as seen in Figures 5.2 and 5.4 from chapter 5.5.

Training. In many environments, the scores reached during training in the hard setting decrease for all or nearly all agents. This is the case in *bigfish*, *fruitbot*, *heist*, *leaper*, *maze*, *ninja*, *plunder*, and *starpilot*. In *caveflyer*, *chaser*, *climber*, *coinrun*, *dodgeball*, and *jumper*, two agents score lower or show worse training progress than in the easy setting, respectively. In *miner*, *PPO* performs better. Also, in *bossfight*, both DQN agents perform better, and *PPO* even doubles its reached scores.
However, whether an agent showed a good or not training performance, as well as that *PPO* outperforms the other agents, remain the same. Also, the changes in scores are not dramatic. While the performances are generally worse than in the easy setting, we still consider the settings to show similar results. A summary of the differences between easy and hard training results is provided in Figure 6.5.

Generalization. *REINFORCE* does not show visible changes in its generalization performance in any of the environments. DQN^M has an increased generalization per-



(a) Scores (b) Training Progress

Figure 6.5.: Differences between easy and hard training

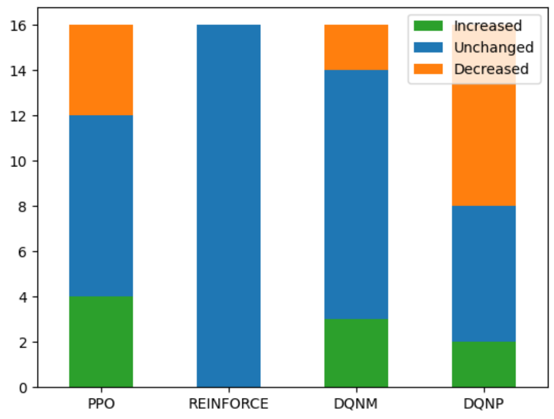


Figure 6.6.: Differences between easy and hard generalization performance

formance in *bigfish*, *fruitbot*, and *starpilot*. In *bigfish* and *starpilot*, the generalization performance remains negative, i.e., it still performs better in the evaluation environment. However, *PPO* and DQN^P show better generalization performances in multiple environments. For *PPO*, this is true in four environments, while it also has an increased generalization performance in four environments. DQN^P shows a better generalization performance in eight of the 16 environments while only showing an increased generalization performance in two environments. This could be an indicator that DQN^P benefits from the longer training time.
However, we do not face dramatic changes similar to the training progress. As for the training performances, we consider the settings to show similar results. A summary of the differences between easy and hard generalization performances is provided in Figure 6.6.

7. Characterization of Game Mechanisms

In chapter 4.3, we discussed several game mechanisms. To this end, we grouped the environments into different categories and discussed their properties. This chapter investigates whether any of the categories of mechanisms impact the performance of one of the agents.

7.1. *PPO*

Environments without enemies and lethal obstacles are seldomly among the best or most impressive achievements of the *PPO* agent. Also the training progress in friendly environments is less impressive than in non-friendly ones. Additionally, as summarized in Figure 7.1, in non-friendly environments, the generalization performance is sometimes constant, whereas in friendly environments it is always positive, i.e., the performance in evaluation levels decreases visibly. However, the results do not indicate a strong preference of the agent toward non-friendly environments. Within the non-friendly challenges, the generalization performance is only constant in some environments where the agent has to avoid enemies, as depicted in Figure 7.2. We do not see another relation between environments with different non-friendly challenges. Within the different friendly challenges, the agent seems to perform well in environments where it has to collect objects. This is especially true for the training progress, as summarized in Figure 7.3. The agent is less successful in moving towards a goal in environments without intermediate rewards or with barriers it has to open. However, moving towards a goal while having intermediate rewards seems to provide a good learning environment for the agent. Compared to the maximal reward, the agent is most successful in this kind of environment. Yet, this only holds for the training progress. We did not see such a pattern for the agents' generalization performance.
However, the agents performances seem to be affected by the surrounding. Figure 7.4 shows a summary of this impact. The agent shows the best training performances in environments with platforms, walls, or open environments. In maze-structured environments or when the environment contains walls, the generalization performance increases, i.e., the training performance improves while the evaluation performance does not keep up with this progress. This behavior is also seen in some, but not all, open environments.
Except for *maze*, which is a fully observable environment, the agent scores best compared to the maximal score in the partially observable environments. Additionally, we see an increase in generalization performance in fully observable environments. Full tendencies are summarized in Figure 7.5. However, we assume that the reason does not lie in observability itself. I.e., all maze-structured environments, where the agent also has an

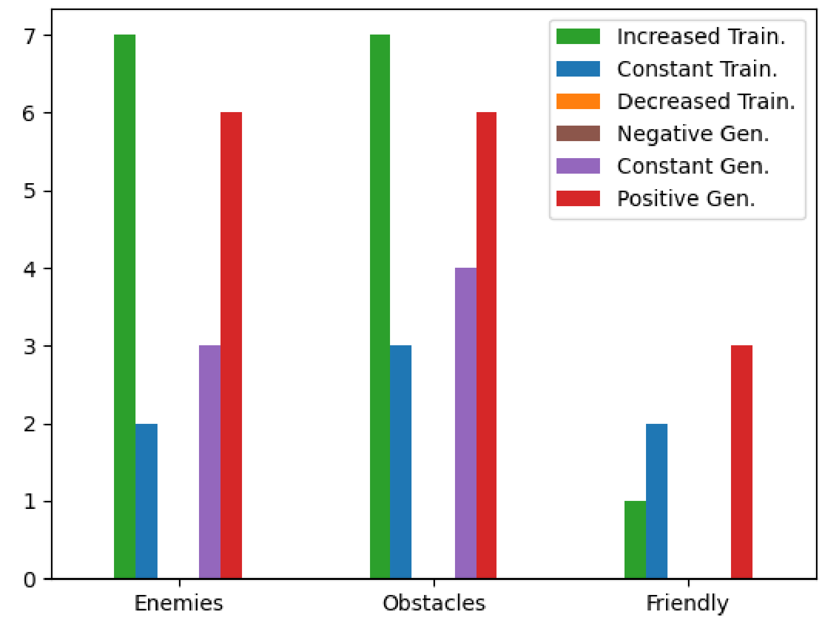


Figure 7.1.: Friendly vs. non-friendly, *PPO*

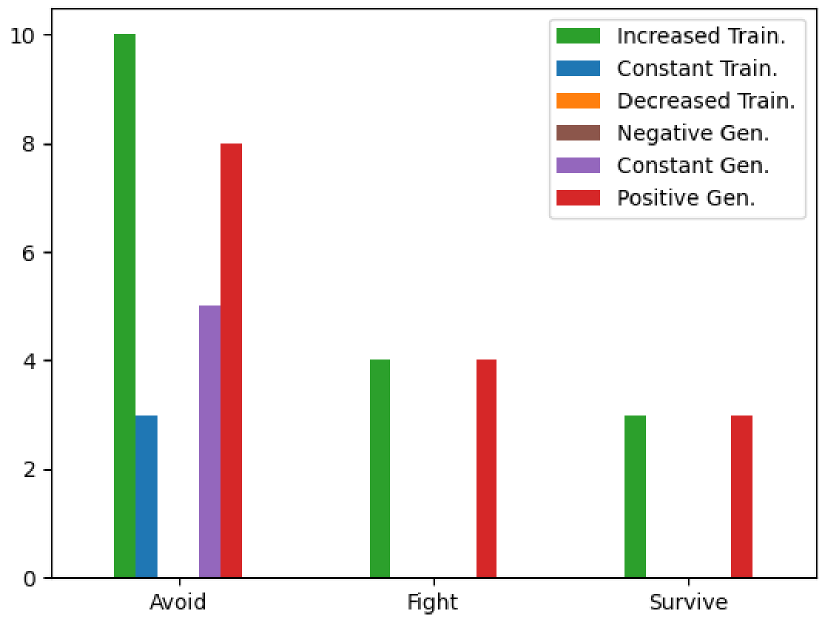


Figure 7.2.: Non-friendly challenges, *PPO*

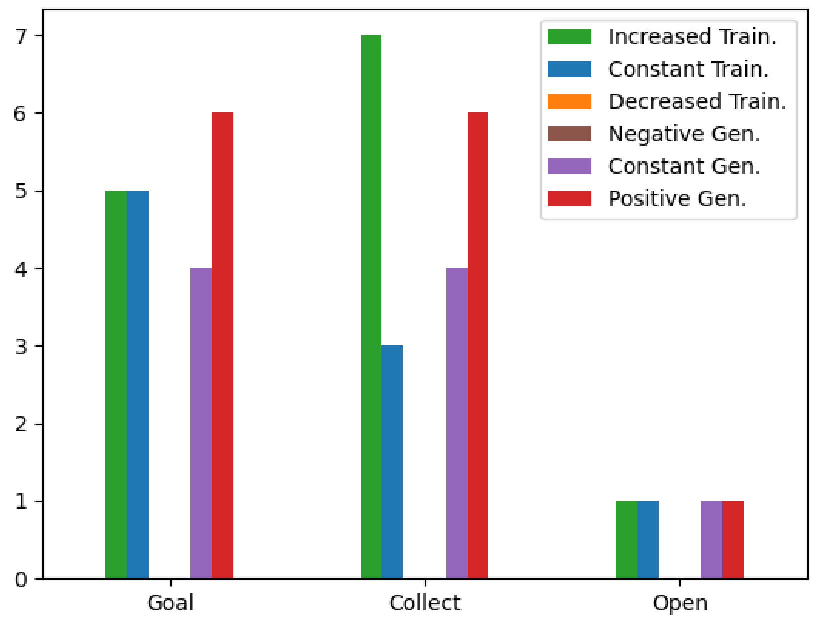


Figure 7.3.: Friendly challenges, *PPO*

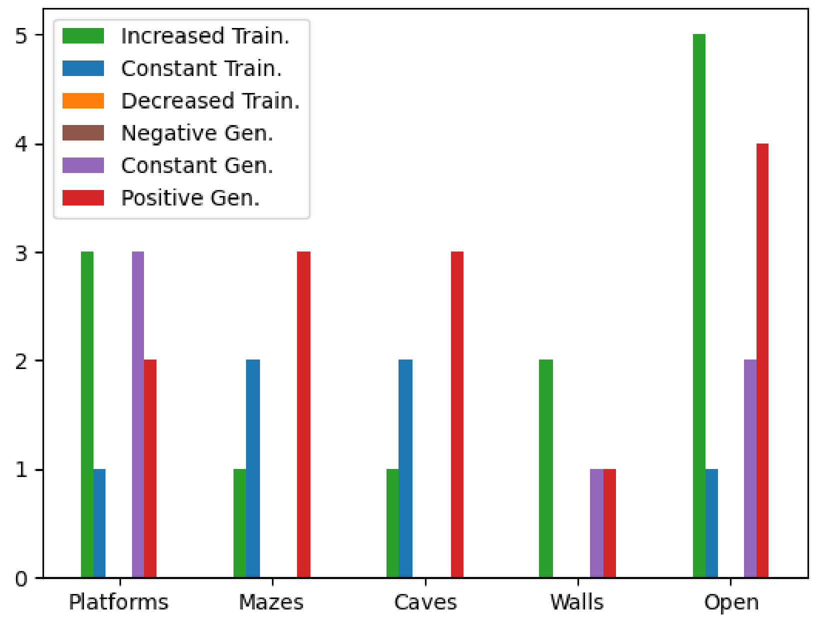


Figure 7.4.: Different surroundings, *PPO*

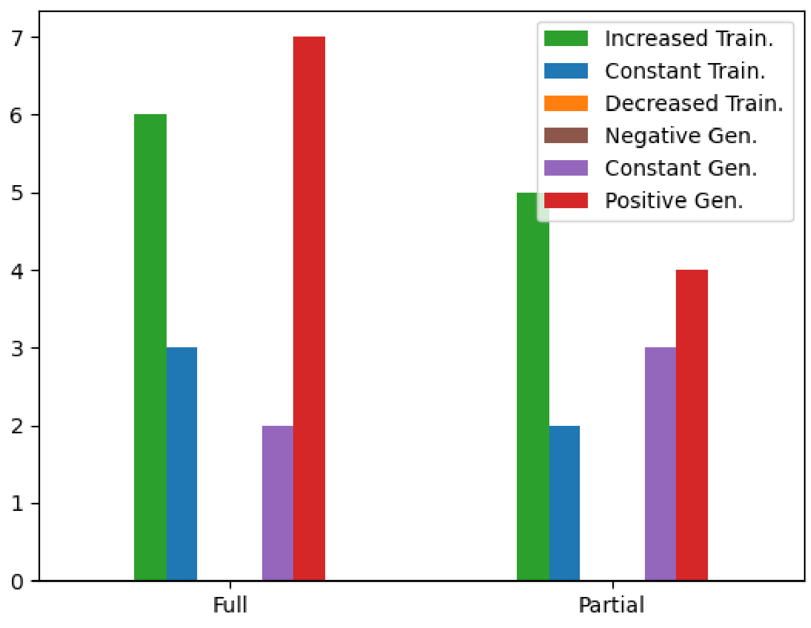


Figure 7.5.: Fully vs. partially observable, *PPO*

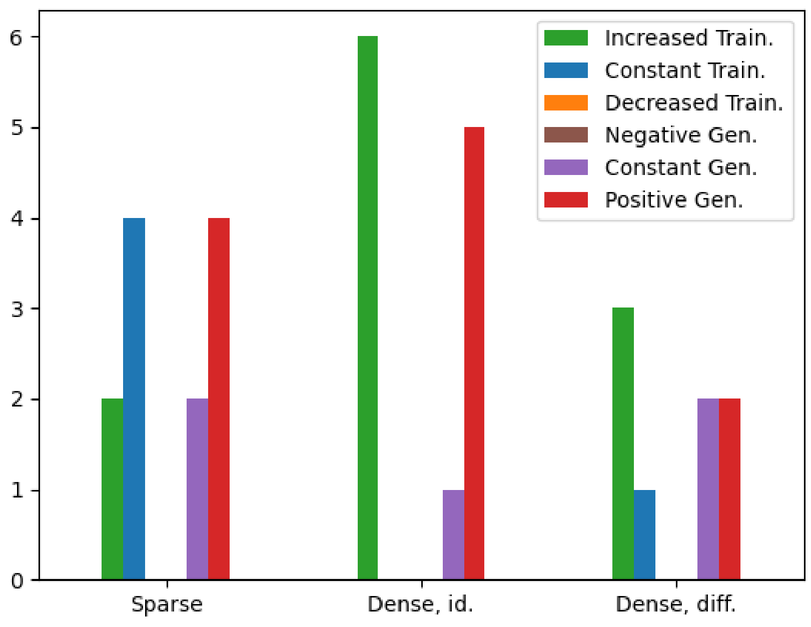


Figure 7.6.: Reward structures, *PPO*

increased generalization performance, are fully observable.
The negative reward in *fruitbot* does not hinder the agent from performing well in this environment. While it does not perform ideally in this environment, compared to other environments without a negative reward, it still performs well regarding the maximal achievable score. In *plunder*, the agent only meets the trivial score. We assume that the agent understands how to score in *plunder*, but it does not understand the time penalty for similar actions.
The most impressive training progress by the *PPO* agent is in environments providing intermediate rewards, except for *coinrun* and *jumper*. The agents' training progress here is also good, but they only provide level-end rewards. The agent also scores significantly above trivial scores in dense reward environments. However, scores close to the maximal score are mostly seen in the level-end reward environments. We assume this contradiction comes from the fact that in level-end environments, there is no room for many rewards, i.e., whenever the agent does not solve a level successfully, this reduces the score visibly. However, in dense reward environments, a successful level can nullify a bad performance in another level compared to the trivial score. Yet, this reduces the overall score enough to distance the score from the maximal achievable score.
However, the generalization performance in dense reward environments with identical results is worst, as depicted in Figure 7.6.

7.2. *REINFORCE*

As seen in Figure 7.7, there is no indication that the existence or absence of enemies and lethal obstacles influences the agents' performance. However, contrary to friendly environments, the generalization performance is constant in most non-friendly environments.
Additionally, we see no hints that the non-friendly enemy challenges influence the agents' training performance. However, the *REINFORCE* agent shows a good evaluation performance in two of three environments where the agent has to fight or survive enemies, as well as in environments where the agent has to avoid enemies. The latter is visible in Figure 7.8. In another environment where the agent has to shoot enemies but is friendly in the sense that the enemies will not attack the agent, it also performs relatively well. While there is still one exception, *bossfight*, we assume the agent is good at learning to attack enemies if intermediate rewards are provided. Another challenge that seems to be easier to learn for the agent is collecting. All environments where the agent nearly meets the trivial score fall into this or the previously discussed category. However, some environments with the task of collecting objects still are challenging for the agent. The agents' generalization performance is quite constant in environments where it has to collect or move towards a goal, as depicted in Figure 7.9.
The different kinds of surroundings in the environment do not seem to impact the agents' performance significantly, as depicted in Figure 7.10. However, in environments containing platforms or walls, the training progress is mostly increased, whereas in other we see more neutral training processes. In open environments, and environments

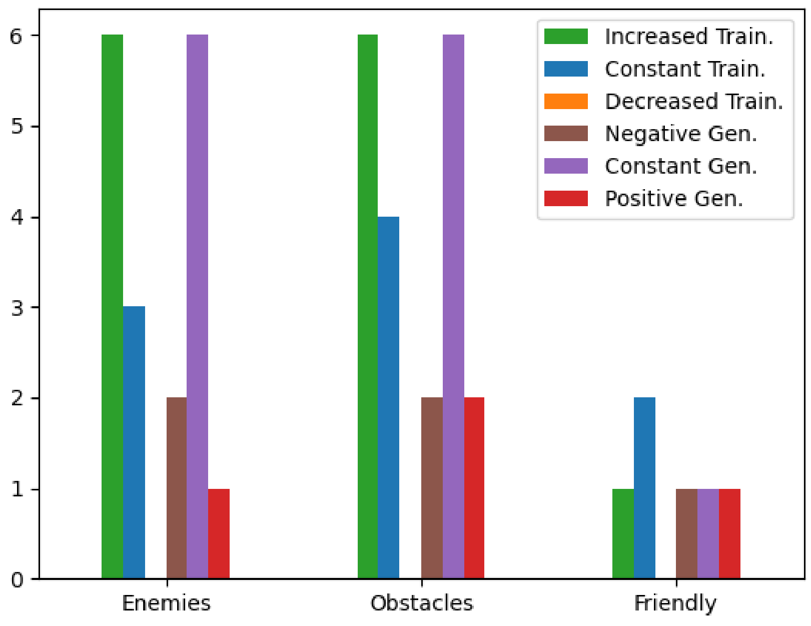


Figure 7.7.: Friendly vs. non-friendly, *REINFORCE*

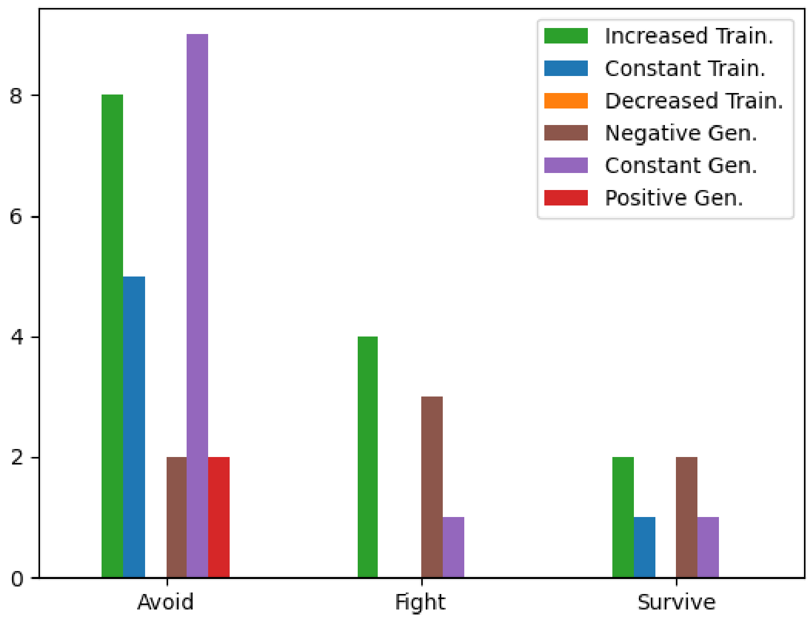


Figure 7.8.: Non-friendly challenges, *REINFORCE*

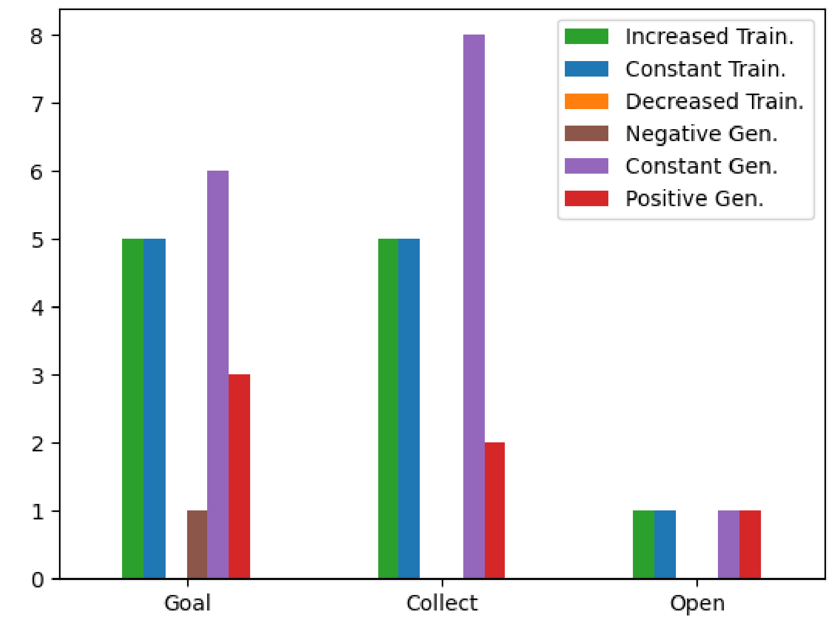


Figure 7.9.: Friendly challenges, *REINFORCE*

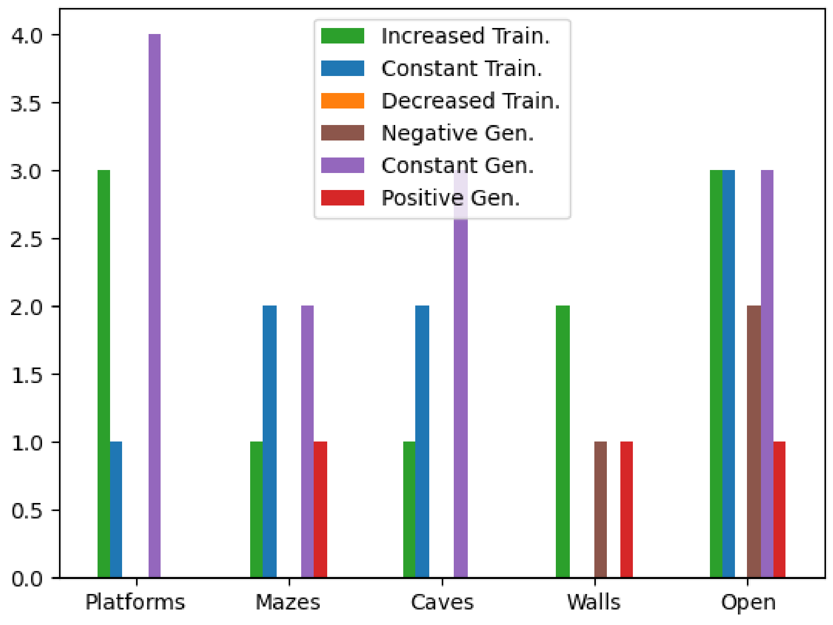


Figure 7.10.: Different surroundings, *REINFORCE*

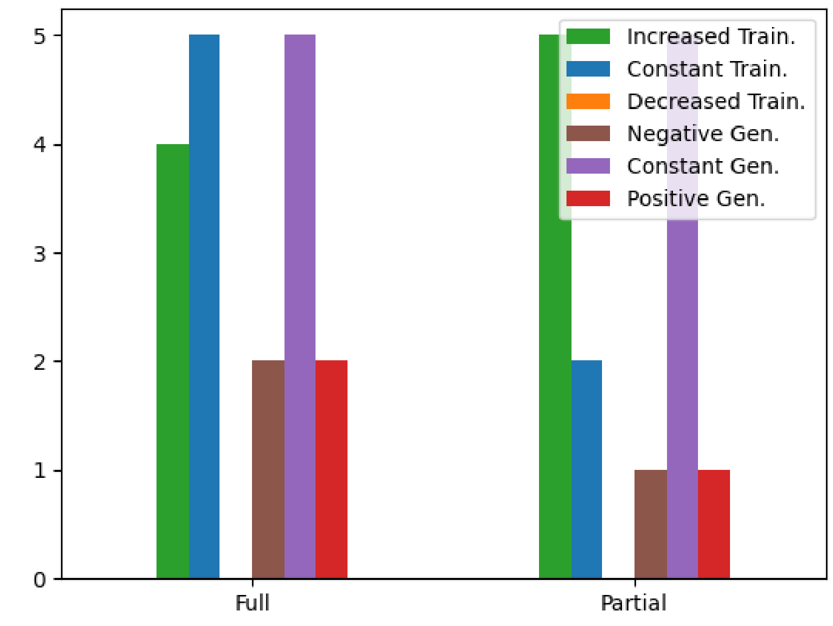


Figure 7.11.: Fully vs. partially observable, *REINFORCE*

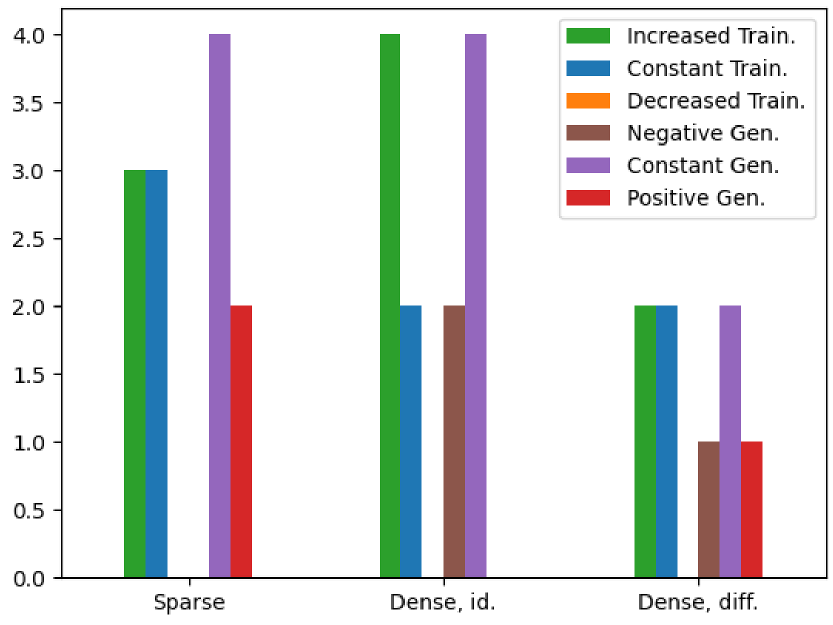


Figure 7.12.: Reward structures, *REINFORCE*

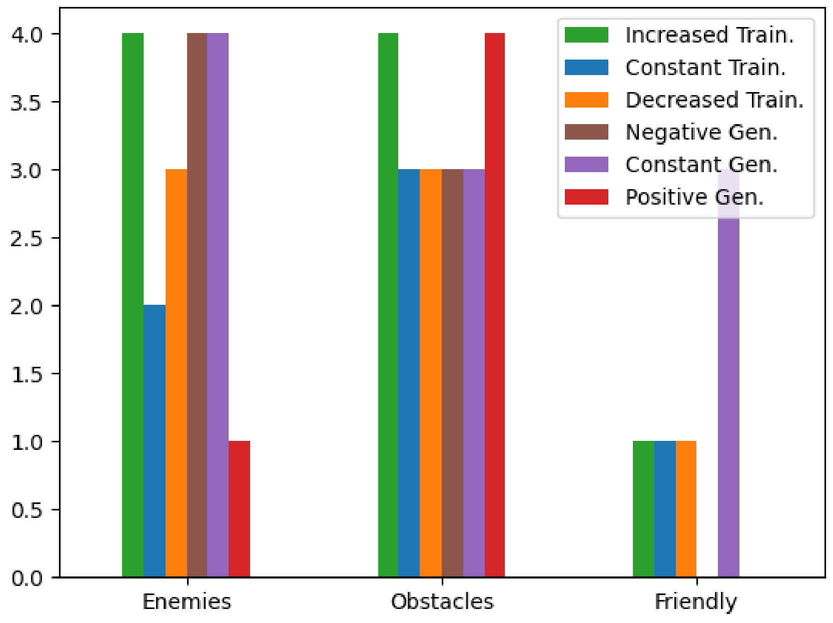


Figure 7.13.: Friendly vs. non-friendly, DQN^M

containing walls, we observe negative generalization performances.
In partially observable environments, the agent shows a better training performance. The generalization performances are comparable in fully and partially observable environments. The statistics is shown in Figure 7.11
The agent meets the trivial score in *fruitbot*, which is the environment providing negative rewards. However, the agents' score, as well as the trivial score, are both negative. In *plunder*, we observe the same as for the *PPO* agent.
Intermediate rewards are provided in all environments where the agent scores well. The agents' evaluation performance only overtakes the training performance in intermediate reward-providing environments, while the evaluation performance in level-end reward environments gets worse than the training performance. Also, all environments where the agent meets the trivial score in the evaluation environment have a dense reward structure. Apart from these aspects, the performances seem comparable according to the tendencies in training and generalization performance, as shown in Figure 7.12.

7.3. DQN^M

The existence or absence of enemies and lethal obstacles does not seem to impact the agents' training performance directly. However, in non-friendly environments, the agent

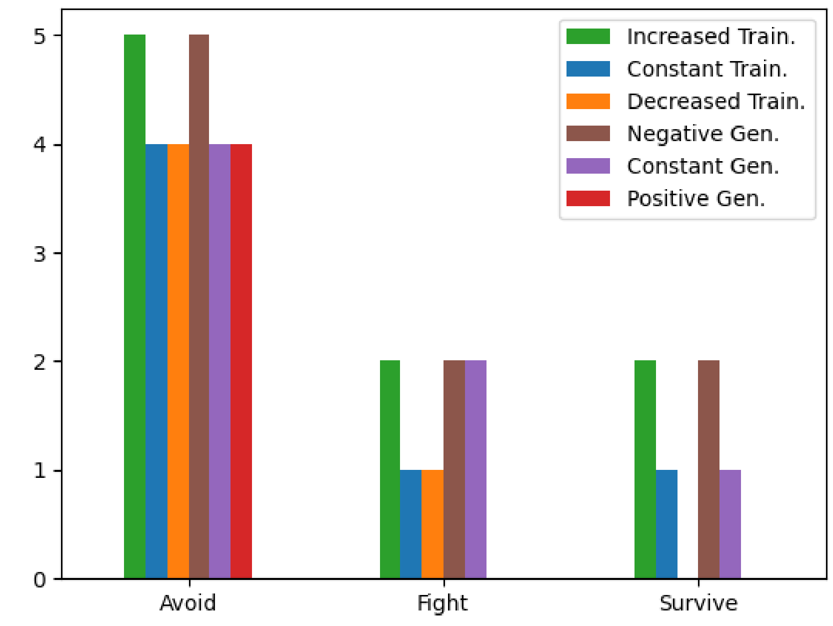


Figure 7.14.: Non-friendly challenges, DQN^M

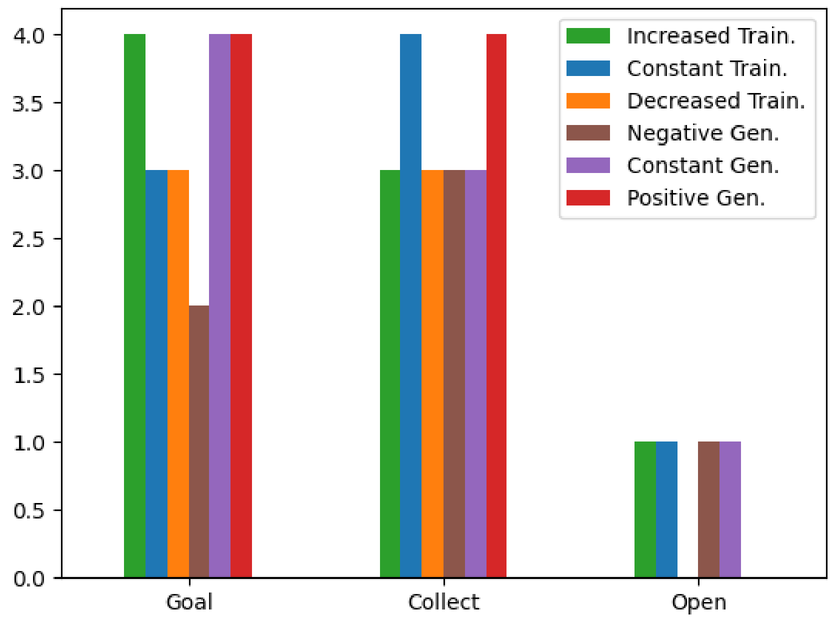


Figure 7.15.: Friendly challenges, DQN^M

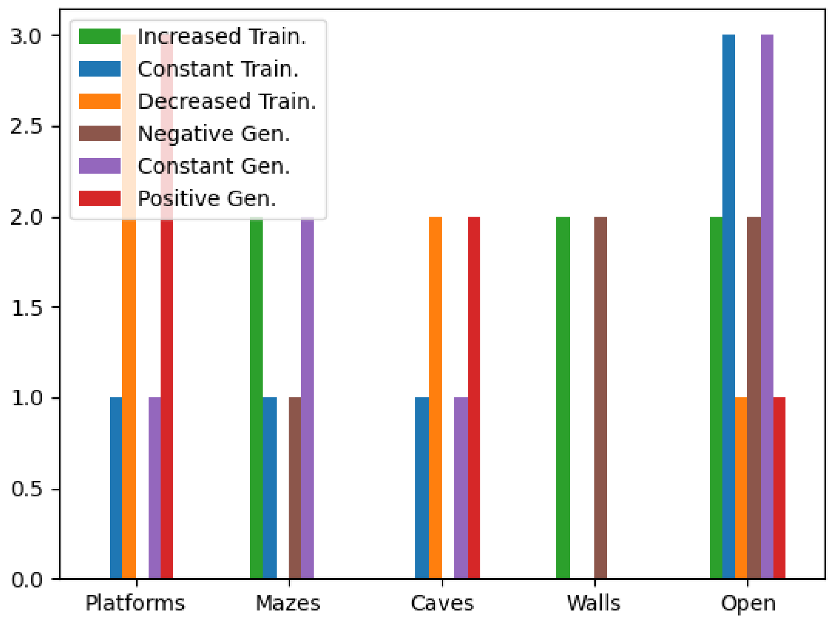


Figure 7.16.: Different surroundings, DQN^M

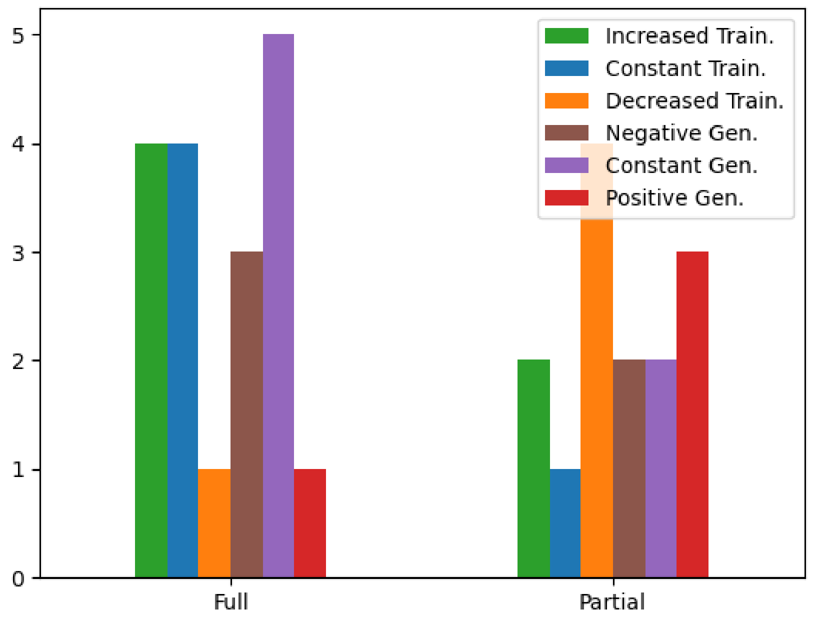


Figure 7.17.: Fully vs. partially observable, DQN^M

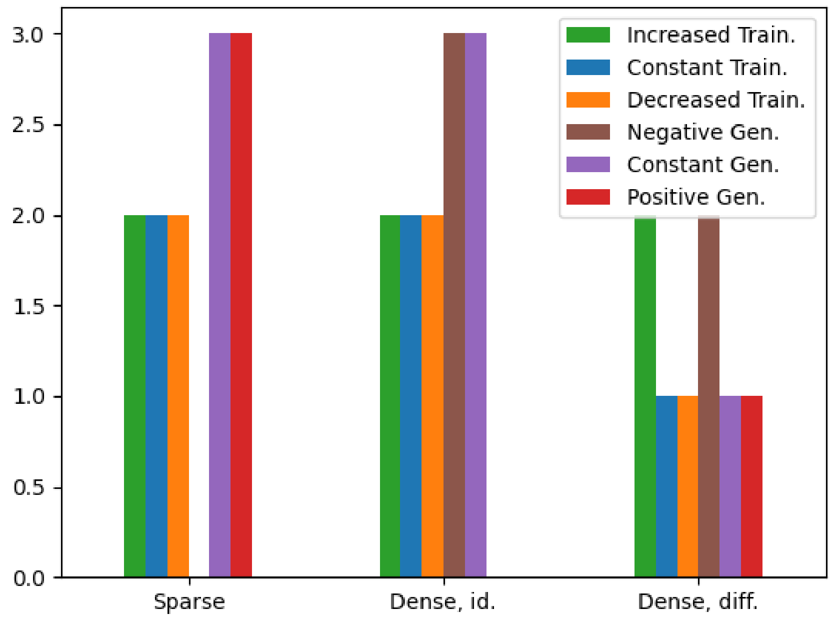


Figure 7.18.: Reward structures, DQN^M

generalizes worse than in friendly environments. Yet, it is noticeable, that only in friendly environments, the agent never shows a decreasing training performance. Figure 7.13 shows the full distribution of the agent's training and generalization progresses.
However, in two of three environments where the agent has to fight enemies actively, the agent (nearly) meets the trivial score. Additionally, in this category, for two environments, the evaluation performance exceeds the training performance in addition to noticeable training progress. This hints that this category could be easier to learn for this agent. As depicted in Figure 7.14, in environments where the agent has to avoid enemies, it does not show decreasing training progress. We do not observe a tendency toward environments where the agent has to collect objects, as visible in Figure 7.15. This is in contrast to the prior discussed agents, which had increased performance in this group of environments. For the DQN^M agent, we observe both improving and decreasing training progress in environments where it has to collect objects, as well as in environments where the agent does not have to collect objects. In general, the performance seems to be less challenge dependent as for other agents. Figures 7.14 and 7.15 show mostly balanced tendencies.
In environments containing walls, the agent shows better training and evaluation progress than in the other categories of the environmental surrounding. In maze-structured environments, the agent also shows good training and generalization performances. Especially in environments containing platforms, we either see a decrease in performance during training or the evaluation performance is not good. As depicted in Figure 7.16,

the performances in open environments are quite diverse.
In Figure 7.17, we can see a small tendency towards fully observable environments. The agent generalizes a bit better in this category of environments. Additionally, the environments in which the training progress decreases are mostly partially observable environments.
In total, the agent performs best in *fruitbot*, i.e., in the environment providing negative rewards. However, although the agent performs best here, the score does not entirely indicate that the agent learns to interpret this negative reward. In *plunder*, the agent does not perform so well. Given the low score, we guess this is not correlated to the penalty, which only hinders achieving good scores but not from scoring in general.
Figure 7.18 shows the training and generalization progresses for the different reward structures. The training progress is mostly similar between the categories: the agent shows increased, constant or decreased progresses in all categories. An increased evaluation performance, including meeting or nearly meeting the trivial score, is most visible in environments that provide intermediate rewards. The agent even reaches negative generalization performances here, i.e., it performs in the full range of the environments' levels even better than in the training levels only. In environments with very sparse rewards, i.e., only at level-end, the agent shows a constant or decreased evaluation performance.

7.4. DQN^P

As depicted in Figure 7.19, the agent has a decreased training performance in one environment containing obstacles and one friendly environment. In non-friendly environments, it sometimes shows a constant, i.e., neither increasing nor decreasing performance. In all categories, the agent has an increased training progress several times. However, only in non-friendly environments the agent generalizes well in some environments.
The agent nearly met the trivial score in two of three environments where it had to avoid enemies to survive and fight enemies. Contrary, in environments where the agent only needs to avoid enemies, we do not see such a tendency. Here, the agent even shows positive generalization performances, as visible in Figure 7.20. Additionally, environments that combine the need to collect and move towards a goal seem to benefit the agents' performance. Minor training progress is visible for environments with only one of these two challenges. A summary is shown in Figure 7.21.
Within surroundings containing walls or mazes, the agent mostly showed training progress. In open environments, the training was rather stable or only slightly progressing, containing one environment with a decrease in training performance. The agent usually showed some progress in environments containing platforms, with one exception. The evaluation performance was worse in environments with caves or platforms. The full statistics for impact of surrounds in depicted in Figure 7.22.
In contrast to our other agents, that showed improved training or generalization performance in the partially observable environments, this agents' performances seem independent of the observability. The corresponding statistics are provided in Figure

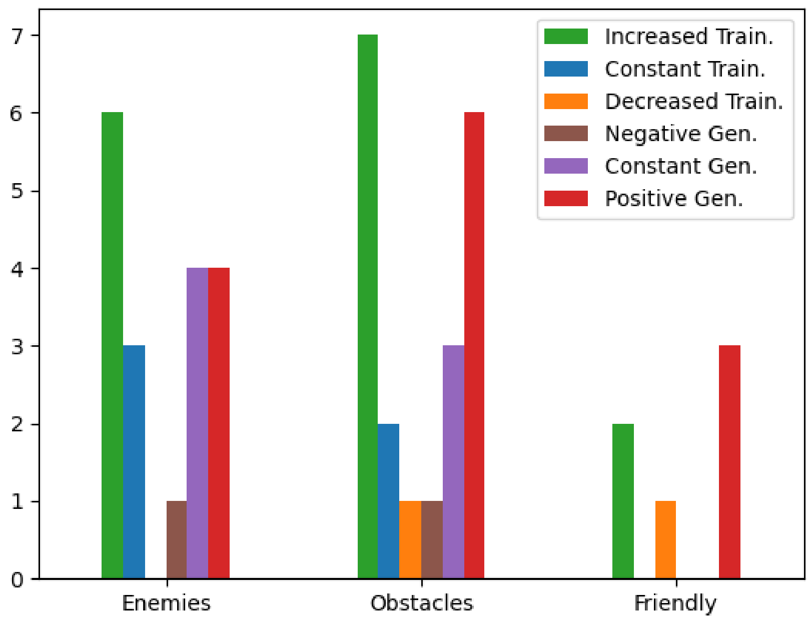


Figure 7.19.: Friendly vs. non-friendly, DQN^P

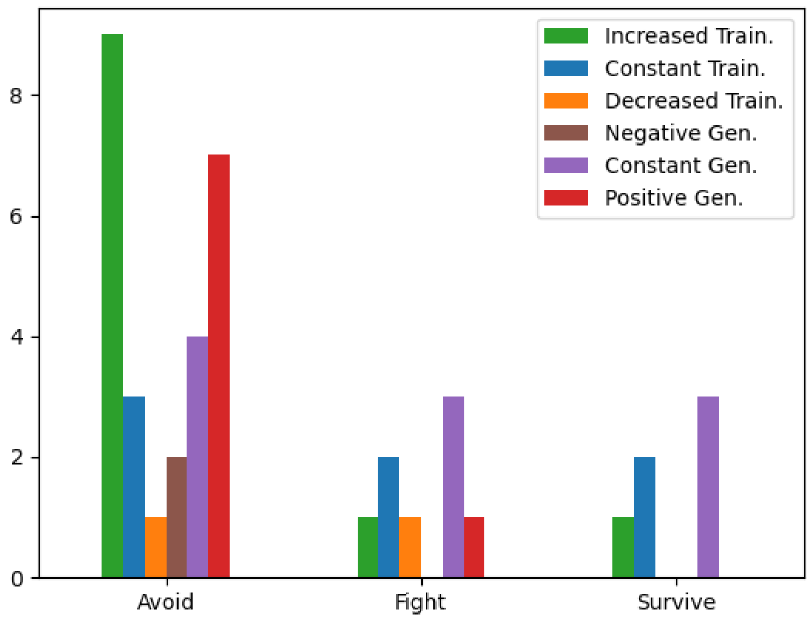


Figure 7.20.: Non-friendly challenges, DQN^P

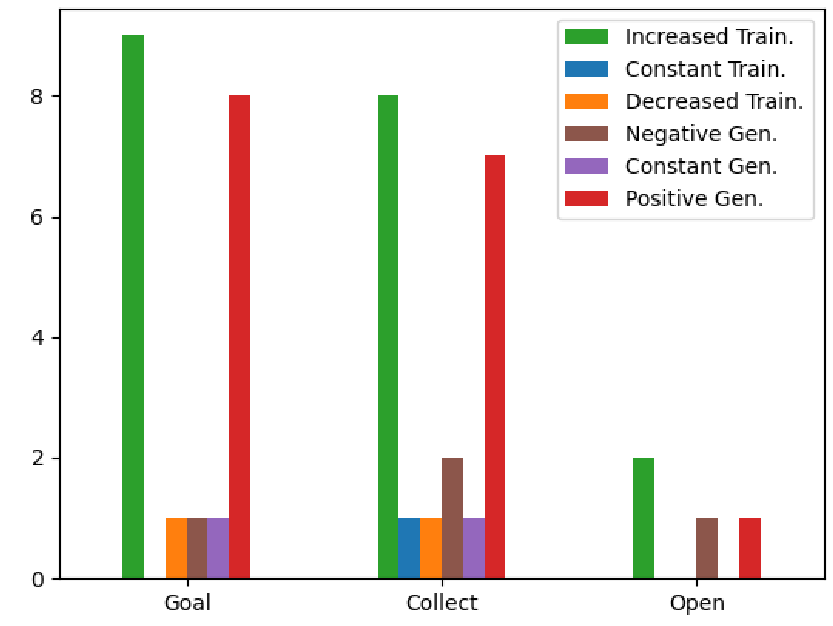


Figure 7.21.: Friendly challenges, DQN^P

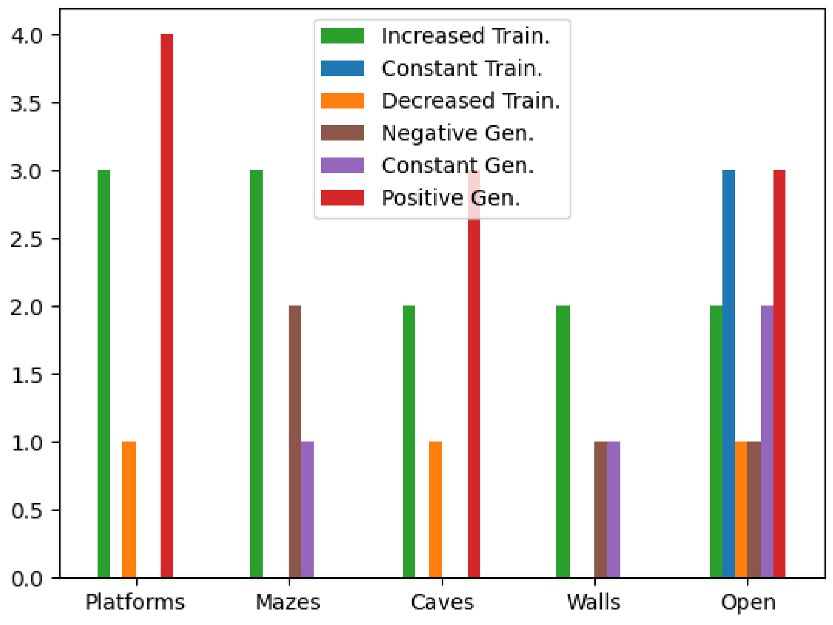


Figure 7.22.: Different surroundings, DQN^P

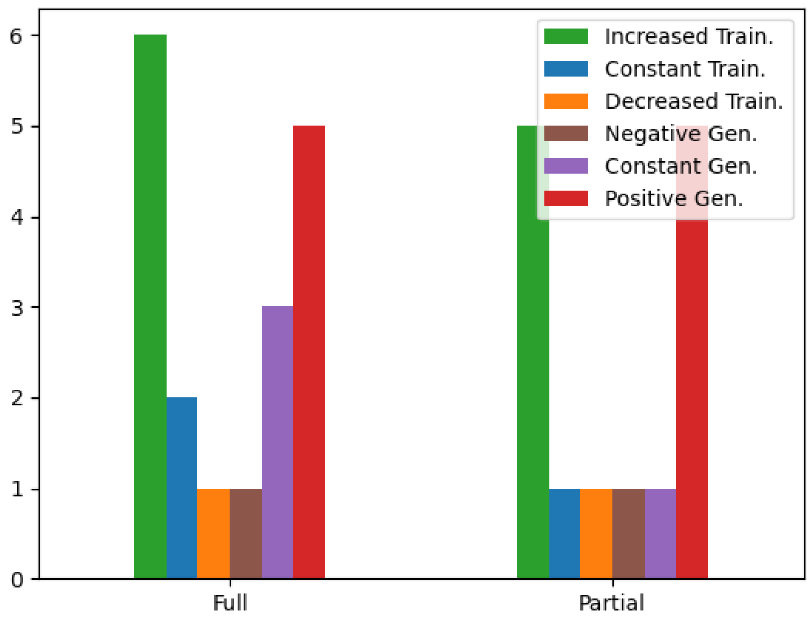


Figure 7.23.: Fully vs. partially observable, DQN^P

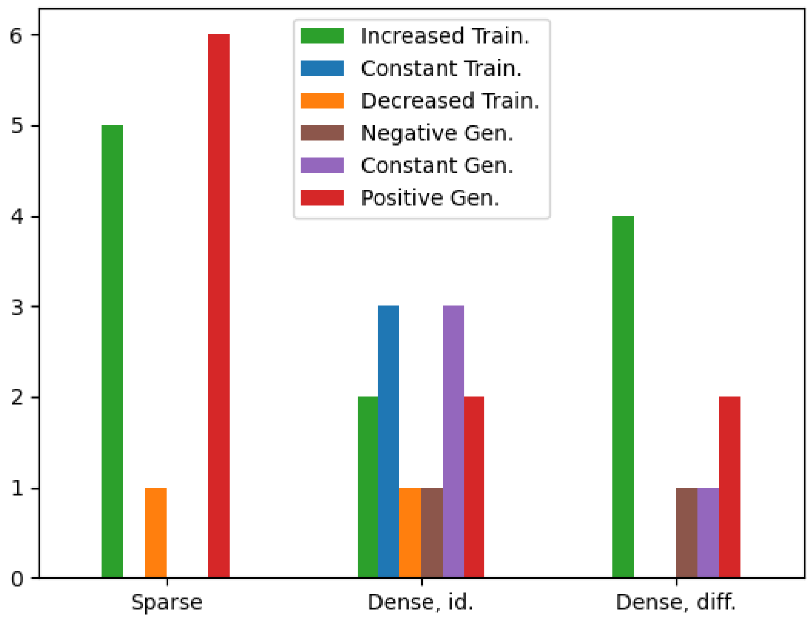


Figure 7.24.: Reward structures, DQN^P

7.23.
In the only environment providing negative rewards, *fruitbot*, the agent performed quite well compared to other environments. This is not the case for the environment *plunder*, which provides penalties.
The reward structure seems to have an impact on the agents' performance. Most of the environments where the agent meets or nearly meets the trivial score provide intermediate rewards, whereas, in level-end only reward environments, the agent performs less well, compared to the trivial score. However, the training progress in sparse reward and dense reward with different rewards, is mostly increased, whereas in dense reward with identical reward settings, the training progress is also neutral quite some times. Nevertheless, in sparse reward settings, the agent fails to generalize. Figure 7.24 visualizes the training and generalization processes in the different reward settings.

7.5. Conclusion

The absence or existence of enemies and lethal objects seems to have a low impact on the agents. Especially the generalization performances are better in non-friendly environments. However, ProcGen only provides three friendly environments. For a final conclusion, it would be interesting to train the agents in more friendly environments.
In environments where the agent has to collect objects or fight enemies, three of four agents showed improved training and evaluation results compared to their performance in other environments. All agents had their best performances in environments providing intermediate rewards, i.e., environments with a dense reward structure. *PPO* and DQN^P seem to prefer partially observable environments. In contrast to dense reward-providing environments, where the improvement in performance is understandable, we do not understand why this is the case. However, we assume this is rather by coincidence, i.e., that those environments are easier to solve for other reasons.
The different environmental surroundings seem to have an impact, but each agent is impacted differently. However, surroundings with walls have a positive impact on DQN^M and DQN^P. We observe that in the only environment providing penalties, *fruitbot*, all agents have one of their best performances. Yet, except for *PPO*, the scores in training and evaluation are still rather low, i.e., it is possible that the agents do not fully understand the negative reward. In this case, the good performance would, similar to the observability, be based on other circumstances.
Besides the problem of the low scores, only one environment provides these negative rewards, i.e., we cannot draw conclusions based on *fruitbot* only. Similarly, we observe a similar problem for non-friendly environments, different surroundings, and partially even the challenges. These categories only provide three to four different environments. Therefore, it is hard to draw universal conclusions.

8. Conclusion and Future Work

This final chapter concludes this thesis in section 8.1. Section 8.2. provides inspiration for additional future work.

8.1. Conclusion

The goal of this thesis was to compare multiple deep reinforcement learning algorithms. To this end, we trained four agents representing three algorithms in all 16 ProcGen environments in two different settings. We discussed each agent's training and generalization performance and investigated how different game mechanisms influence the agents' performance. Additionally, we discussed whether a more challenging setting impacts the performances. Our results show that *PPO* has the best training and evaluation scores. *PPO* scored pretty close to the maximal achievable score in a few environments. The number of environments where *PPO* did not score so well is rather small. Contrary, all other three agents had trouble meeting even the trivial score. However, *REINFORCE* and DQN^M showed excellent generalization performance. Surprisingly, both agents showed better performance in the evaluation than in the training environment. This led them to achieve the trivial score more often during evaluation than during training. We are not certain why this is the case for these two agents, however, a possible explanation is that they learn to solve a few levels of the training set. Low scores then are still possible if the knowledge from those levels does not improve the performance in most training levels. However, when these challenges are represented in the newly sampled evaluation levels, then the evaluation score can exceed the training score.
Additionally, we also deepened the observations by Henderson et al., who observed that the same algorithm in different implementations could get differing results [17]. Differing performances were visible for both DQN agents, where DQN^M showed a visibly worse training performance but generalized well, whereas the DQN^P agent had better training yet worse evaluation performance.
Our analysis showed that some game mechanisms, such as the reward structure, impact all agents. Others, e.g., challenges like collecting objects, increased the performance of most agents. On the other hand, different environmental surroundings impacted each agent differently. While all agents had a good performance, compared to their results in other environments, in the environment providing negative rewards, a single instance seems insufficient to draw general assumptions. Further investigations into this direction might be interesting.

8.2. Future Work

In a competition in the ProcGen benchmark, some teams modified the IMPALA network [34]. Seemingly, this changes improved the agents' performances. A structured investigation on this could be interesting.

Furthermore, due to time constraints, we only could train four agents. However, comparing the results to more different agents would help to further assess the performances and gain additional knowledge on agents' potentialities on ProcGen. A first attempt could be Rainbow. A Rainbow agent was evaluated in the work by Cobbe et al., yet only in regards to sampling efficiency [6]. An investigation of Rainbow's generalization performance was omitted. But also agents based on other algorithms are of interest.

Also, we assume that an epsilon-greedy exploration strategy, as used for DQN, is not optimal in ProcGen. As most environments contain multiple no-op actions, only choosing random actions for exploration does not seem beneficial. Especially in sparse reward settings, more complex exploration strategies have been shown to be beneficial on ALE [5, 37]. Therefore, we expect that the performance of our agents could be increased by using an enhanced exploration strategy.

We performed experiments in two different settings. One with the distribution mode *easy*, and one with distribution mode *hard*. However, as discussed in chapter 4.2, ProcGen provides additional modes. These modes additionally challenge an agents' exploration or memory skills. Training and evaluating the agents in additional settings with these modes could further improve the understanding of these agents' performances.

Lastly, it would be interesting to investigate our observations in regard to the game mechanisms in other environments, too. This is especially the case for categories with only a few representatives, as for the negative reward structure or environments providing implicit penalties.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Leemon C Baird III. Advantage updating. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

[3] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[4] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents (extended abstract). *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4148–4152, 2015.

[5] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018.

[6] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning, 2019.

[7] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1282–1289. PMLR, 09–15 Jun 2019.

[8] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games, 2018.

[9] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[10] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih,

Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

[11] Meire Fortunato, Melissa Tan, Ryan Faulkner, Steven Hansen, Adrià Puigdomènech Badia, Gavin Buttimore, Charles Deck, Joel Z Leibo, and Charles Blundell. Generalization of reinforcement learners with working and episodic memory. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[12] Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] T. P. Gros and J. Groß. Rlmate. `https://pypi.org/project/rlmate/`. Accessed: 2022-08-30.

[15] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep statistical model checking. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 96–114, Cham, 2020. Springer International Publishing.

[16] Danijar Hafner. Benchmarking the spectrum of agent capabilities, 2021.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[18] Maximilian Igl, Kamil Ciosek, Yingzhen Li, Sebastian Tschiatschek, Cheng Zhang, Sam Devlin, and Katja Hofmann. Generalization in reinforcement learning with selective noise injection and information bottleneck. *Advances in neural information processing systems*, 32, 2019.

[19] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[20] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA, 2010. Association for Computing Machinery.

[21] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 2684–2691. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[22] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation, 2018.

[23] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *arXiv preprint arXiv:2111.09794*, 2021.

[26] Dimitrios I. Koutras, Athanasios Ch. Kapoutsis, Angelos A. Amanatiadis, and Elias B. Kosmatopoulos. Marsexplorer: Exploration of unknown terrains via deep reinforcement learning and procedurally generated environments, 2021.

[27] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[28] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[29] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents, 2017.

[30] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents, 2017.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[34] Sharada Mohanty, Jyotish Poonganam, Adrien Gaidon, Andrey Kolobov, Blake Wulfe, Dipam Chakraborty, Gražvydas Šemetulskis, João Schapke, Jonas Kubilius, Jurgis Pašukonis, Linas Klimas, Matthew Hausknecht, Patrick MacAlpine, Quang Nhat Tran, Thomas Tumiel, Xiaocheng Tang, Xinwei Chen, Christopher Hesse, Jacob Hilton, William Hebgen Guss, Sahika Genc, John Schulman, and Karl Cobbe. Measuring sample efficiency and generalization in reinforcement learning benchmarks: Neurips 2020 procgen benchmark, 2021.

[35] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning, 2015.

[36] Open Ended Learning Team, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard, and Wojciech Marian Czarnecki. Open-ended learning leads to generally capable agents, 2021.

[37] Georg Ostrovski, Marc G. Bellemare, Aaron van den Oord, and Remi Munos. Count-based exploration with neural density models, 2017.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[39] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[42] Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of self-interpretable agents. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 414–424, 2020.

[43] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011. ACM, 2019.

[44] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[45] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[46] Cheng Xue, Vimukthini Pinto, Chathura Gamage, Ekaterina Nikonova, Peng Zhang, and Jochen Renz. Phy-q: A testbed for physical reasoning, 2021.

[47] Amy Zhang, Nicolas Ballas, and Joelle Pineau. A dissection of overfitting and generalization in continuous reinforcement learning, 2018.

[48] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning, 2018.

[49] Chenyang Zhao, Olivier Sigaud, Freek Stulp, and Timothy M Hospedales. Investigating generalisation in continuous deep reinforcement learning. *arXiv preprint arXiv:1902.07015*, 2019.

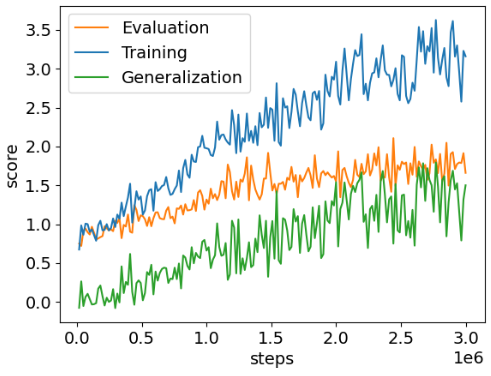
Appendices

A. Training and Generalization for Each Agent

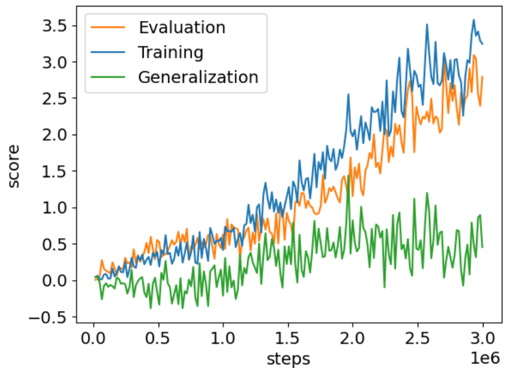
We provide each agent's training, evaluation, and generalization performance in both settings. It is based on the same data as the visualization in Figures 5.1-5.4 in chapter 5.5. However, we found it helpful to gain further knowledge about each agent's performance.

A.1. *PPO* - Easy

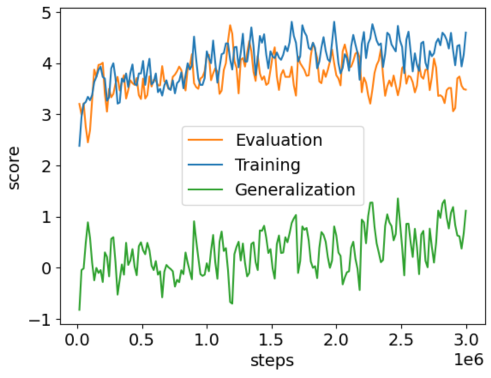
Figure A.1 shows the training, evaluation, and generalization performance of the *PPO* agent for all 16 environments in the easy setting.



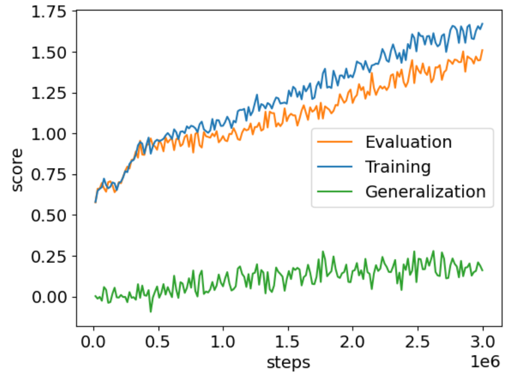
(a) bigfish



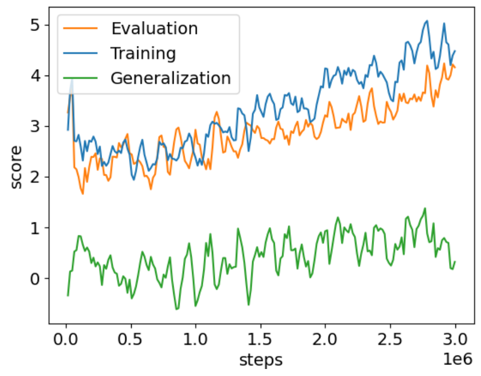
(b) bossfight



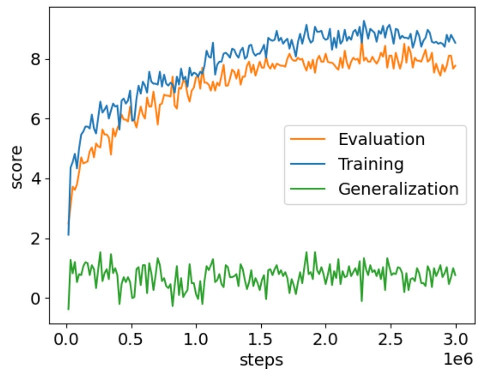
(c) cavefyler



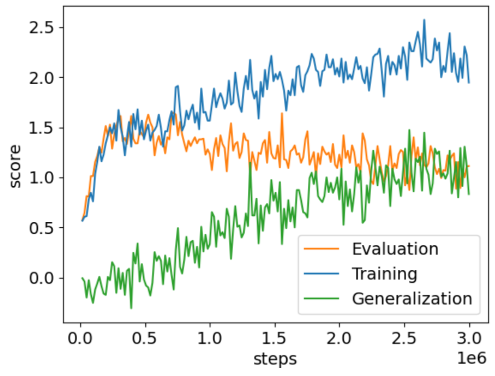
(d) chaser



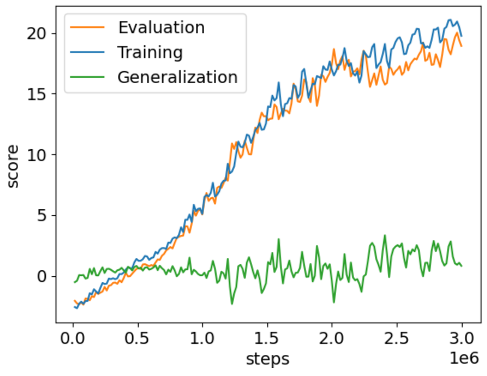
(e) climber



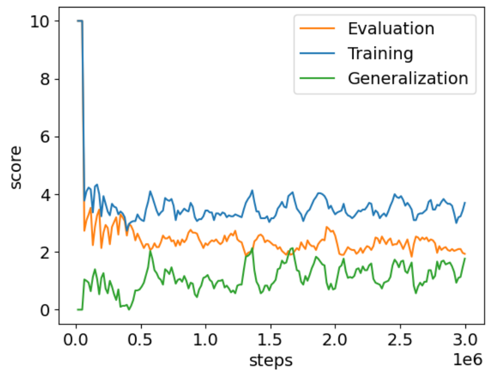
(f) coinrun



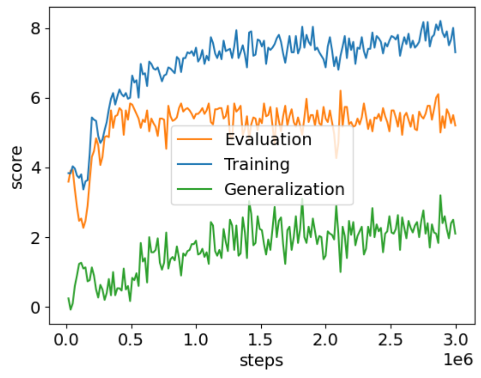
(g) dodgeball



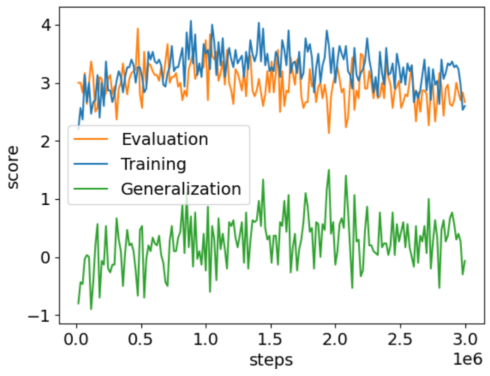
(h) fruitbot



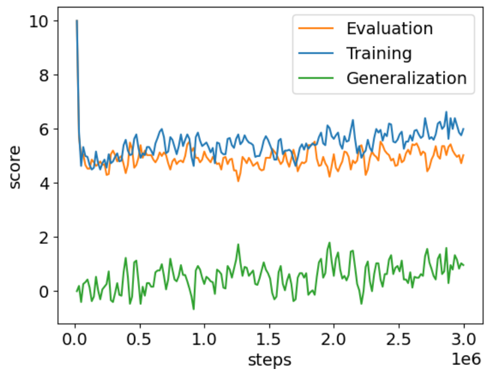
(i) heist



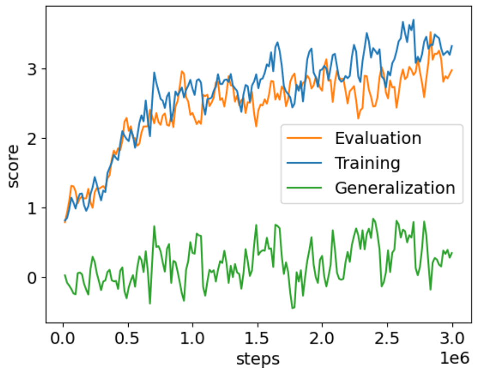
(j) jumper



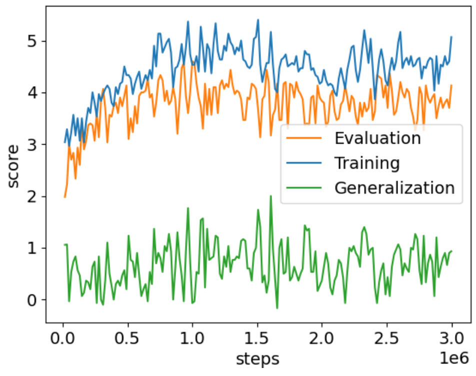
(k) leaper



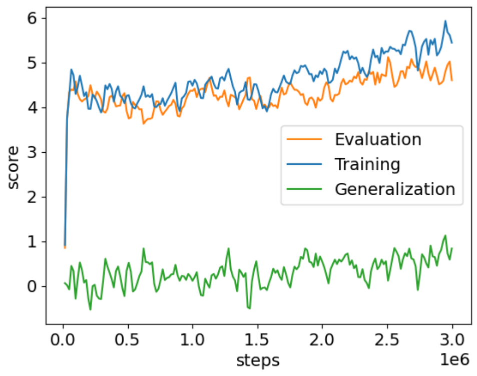
(l) maze



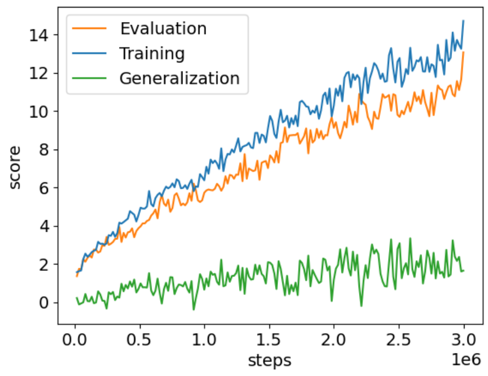
(m) miner



(n) ninja



(o) plunder

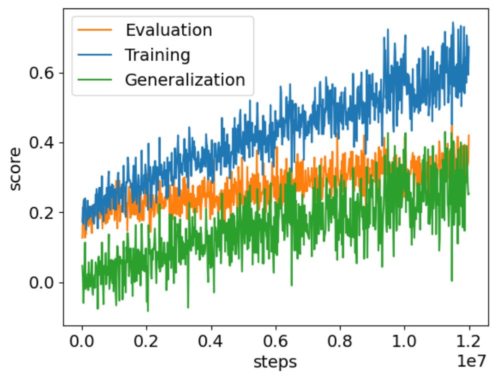


(p) starpilot

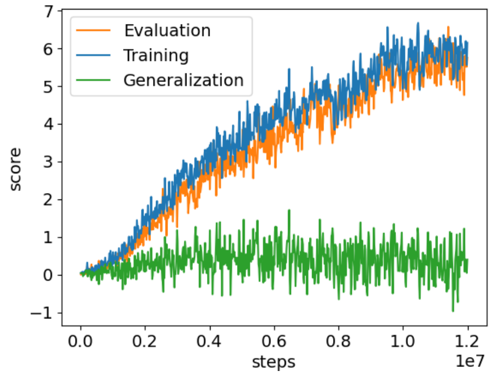
Figure A.1.: *PPO* performance - easy setting

A.2. *PPO* - Hard

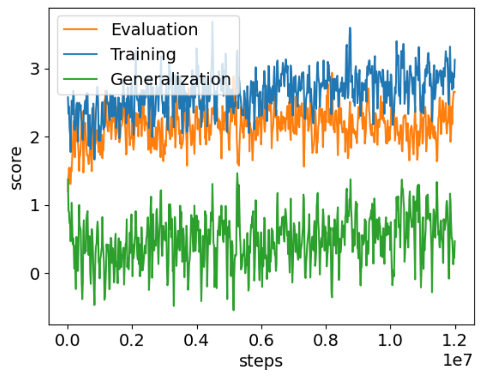
Figure A.2 shows the training, evaluation, and generalization performance of the *PPO* agent for all 16 environments in the hard setting.



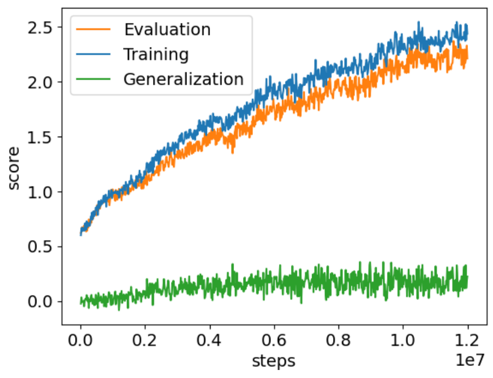
(a) bigfish



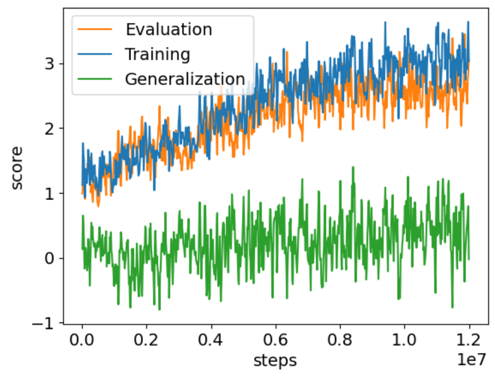
(b) bossfight



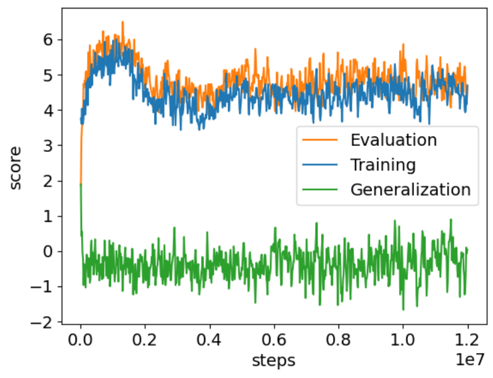
(c) cavefyler



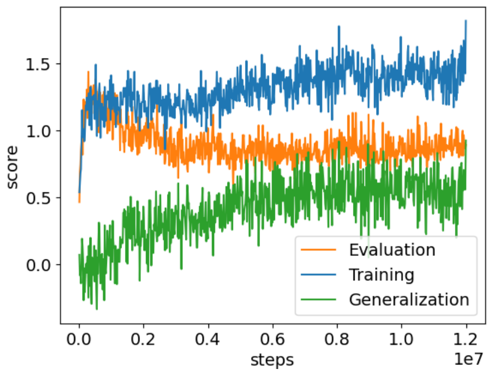
(d) chaser



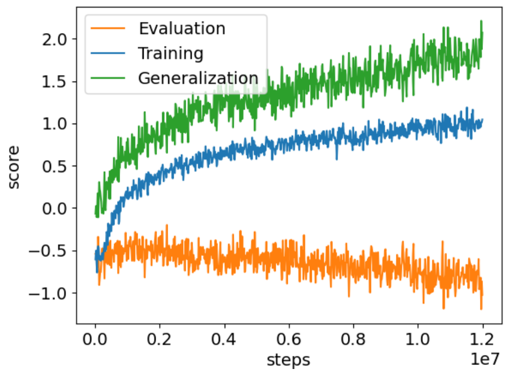
(e) climber



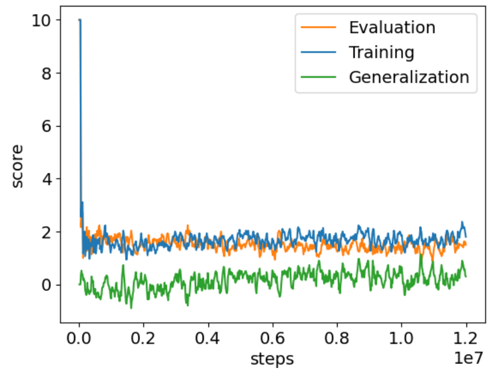
(f) coinrun



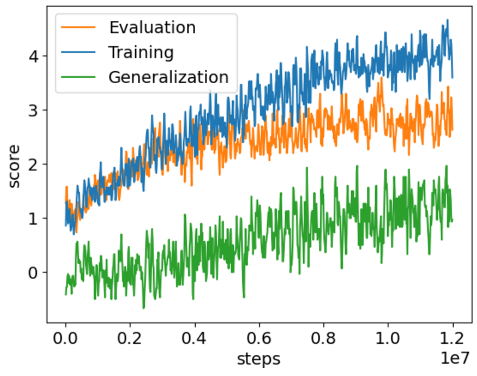
(g) dodgeball



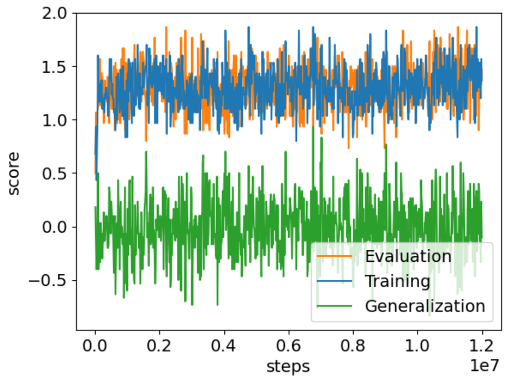
(h) fruitbot



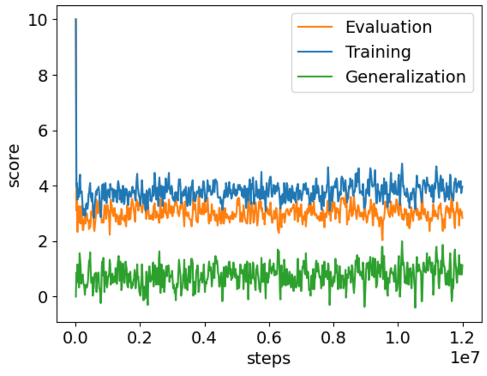
(i) heist



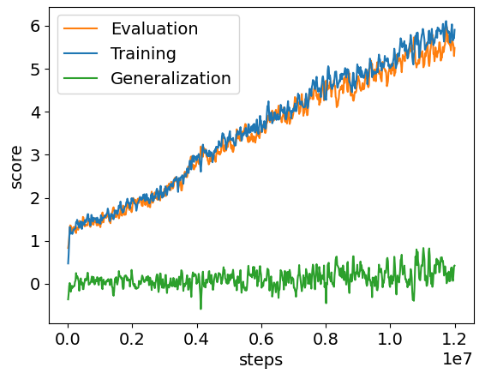
(j) jumper



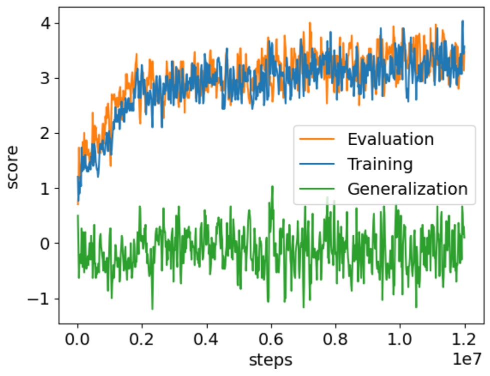
(k) leaper



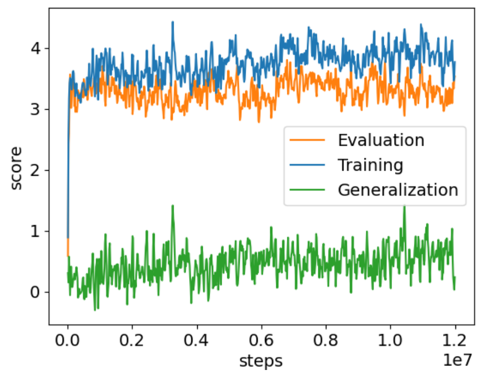
(l) maze



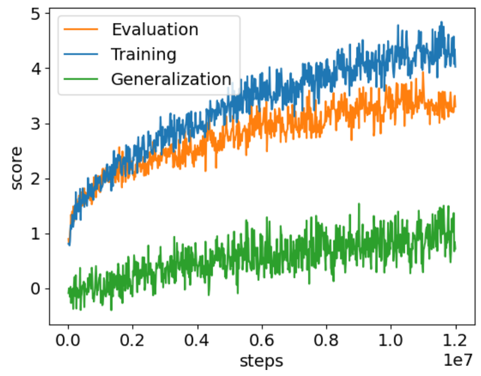
(m) miner



(n) ninja



(o) plunder

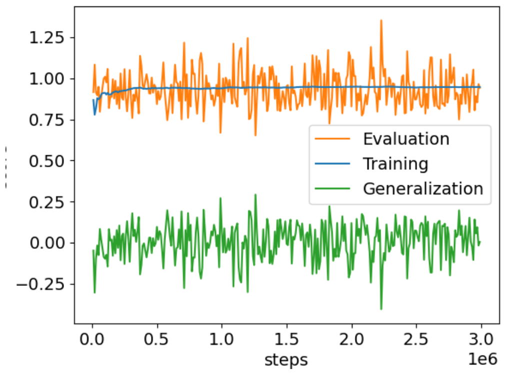


(p) starpilot

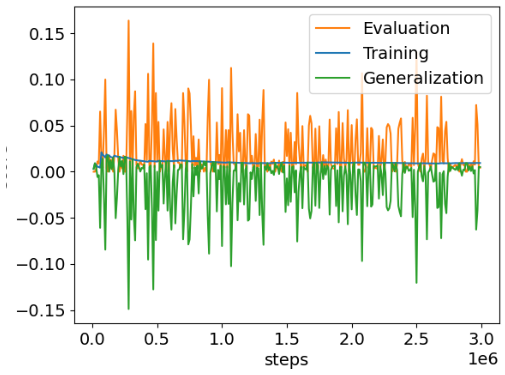
Figure A.2.: *PPO* performance - hard setting

A.3. *REINFORCE* - Easy

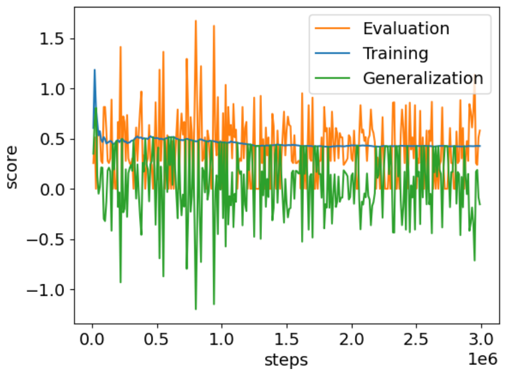
Figure A.3 shows the training, evaluation, and generalization performance of the *REINFORCE* agent for all 16 environments in the easy setting.



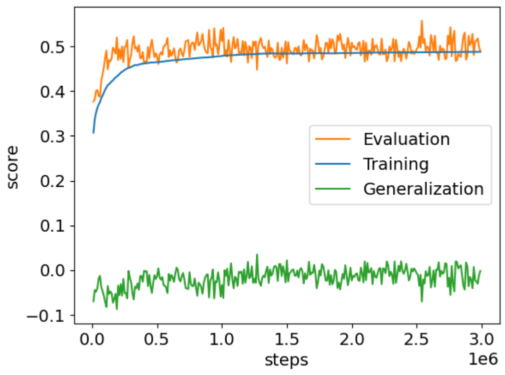
(a) bigfish



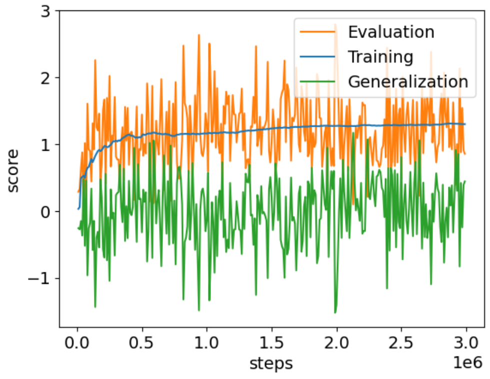
(b) bossfight



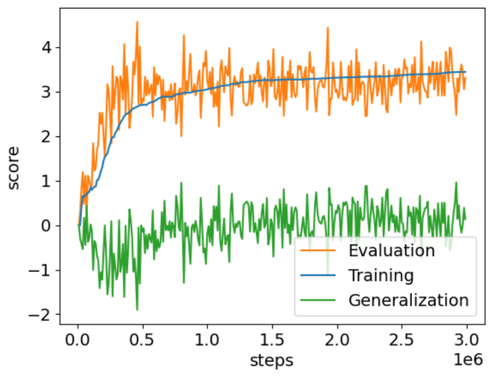
(c) cavefyler



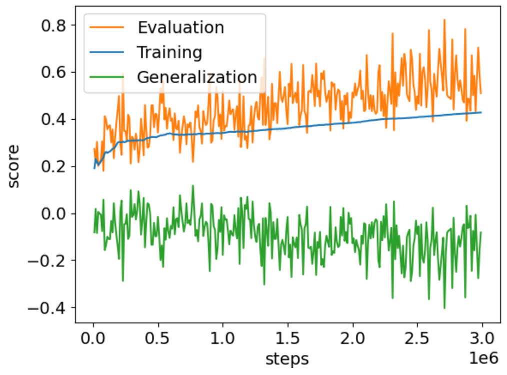
(d) chaser



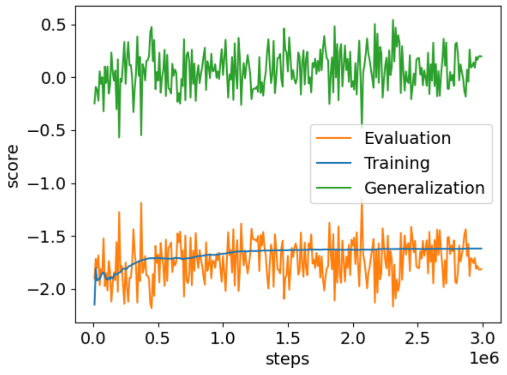
(e) climber



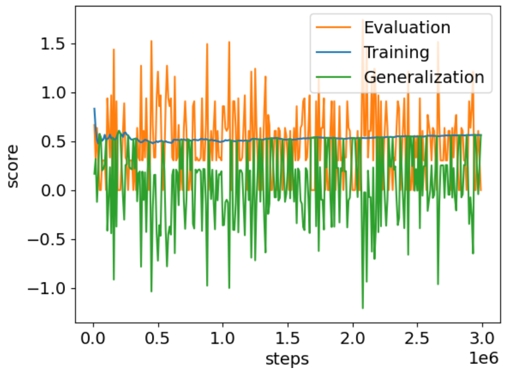
(f) coinrun



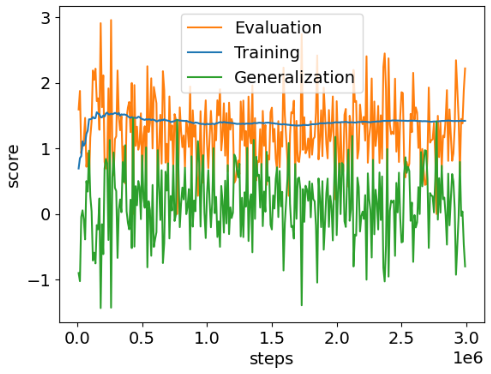
(g) dodgeball



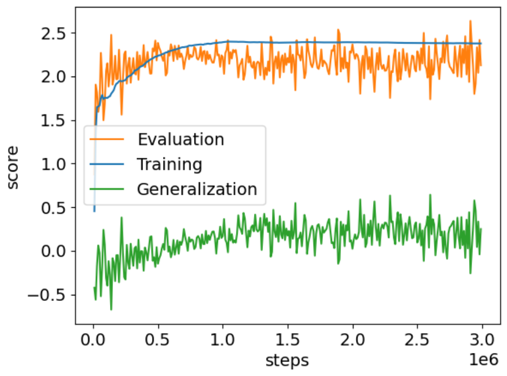
(h) fruitbot



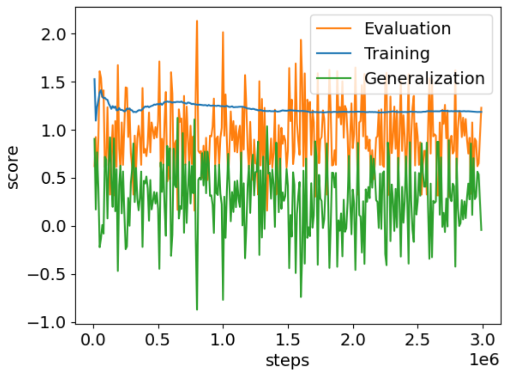
(i) heist



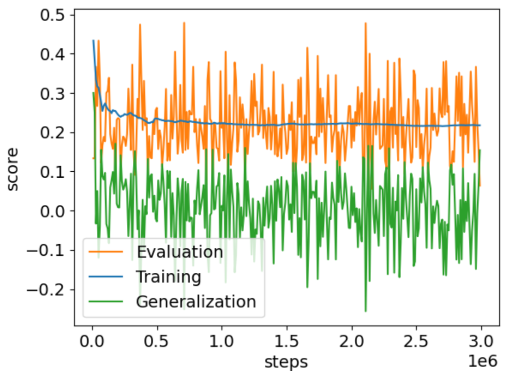
(j) jumper



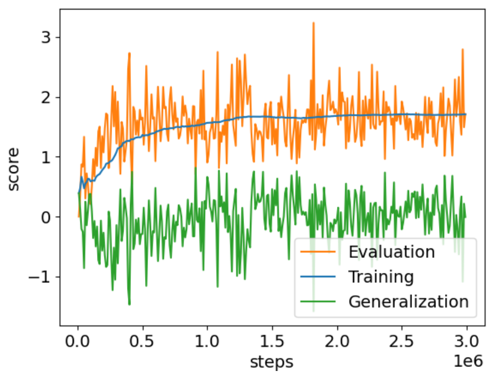
(k) leaper



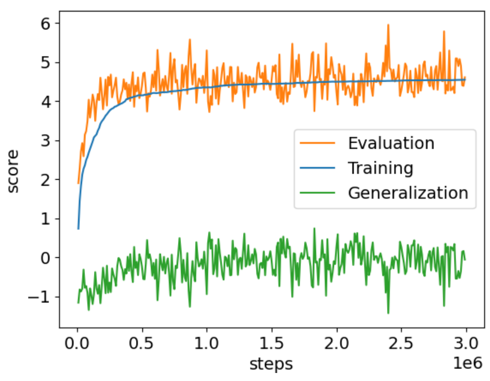
(l) maze



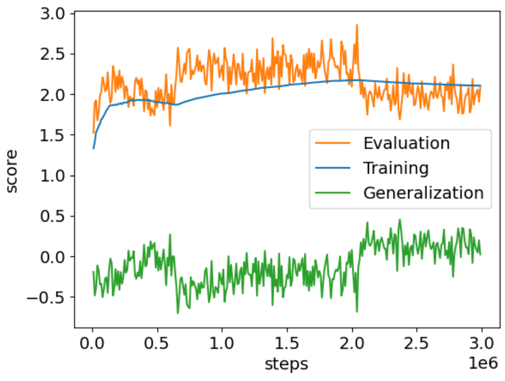
(m) miner



(n) ninja



(o) plunder

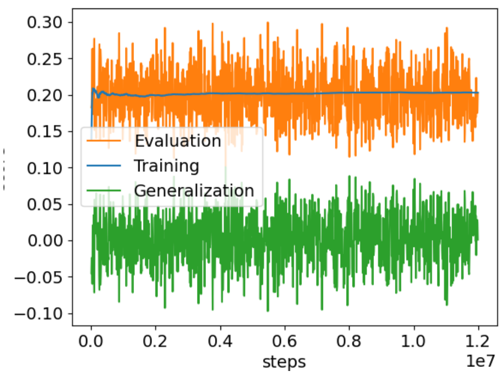


(p) starpilot

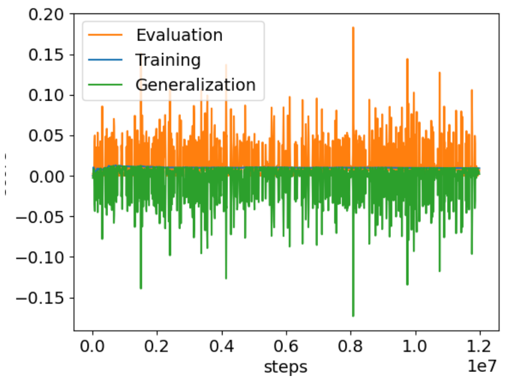
Figure A.3.: *REINFORCE* performance - easy setting

A.4. *REINFORCE* - Hard

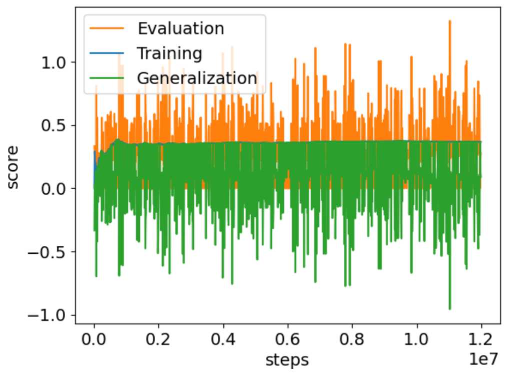
Figure A.4 shows the training, evaluation, and generalization performance of the *REINFORCE* agent for all 16 environments in the easy setting.



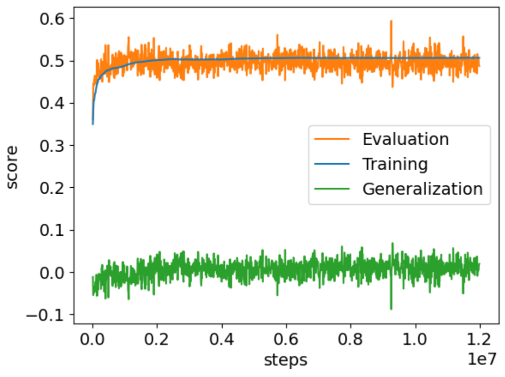
(a) bigfish



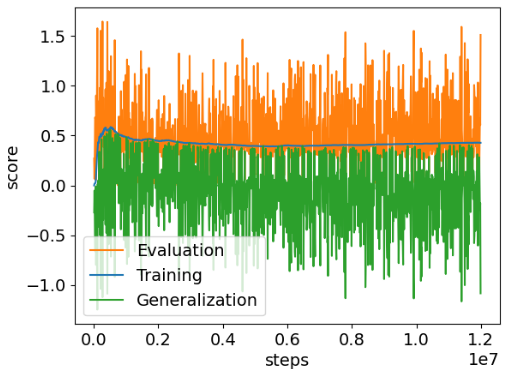
(b) bossfight



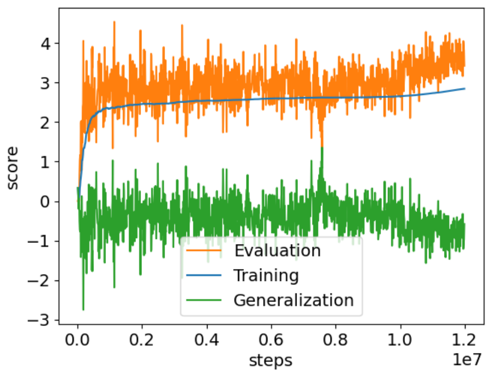
(c) cavefyler



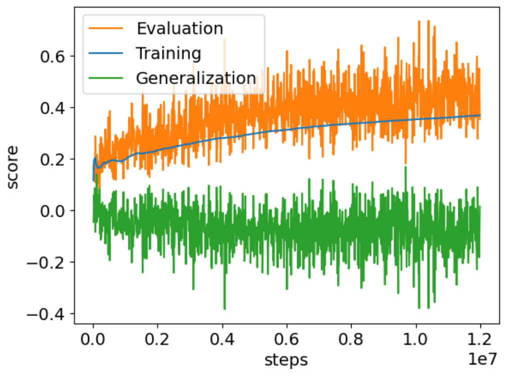
(d) chaser



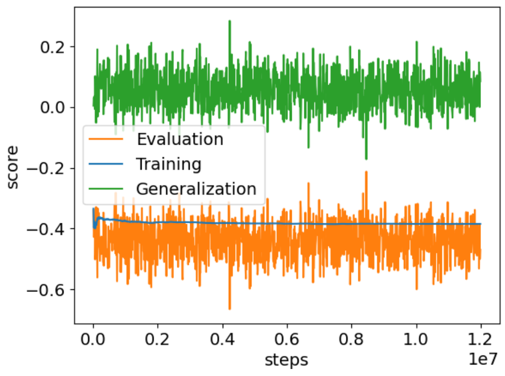
(e) climber



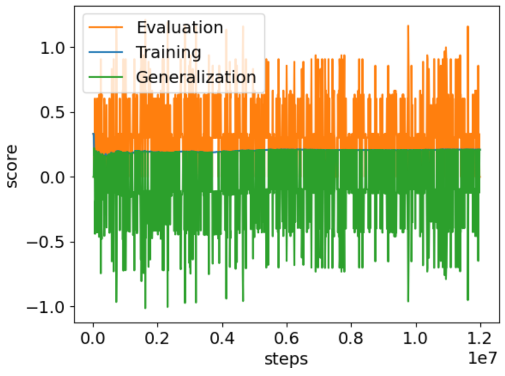
(f) coinrun



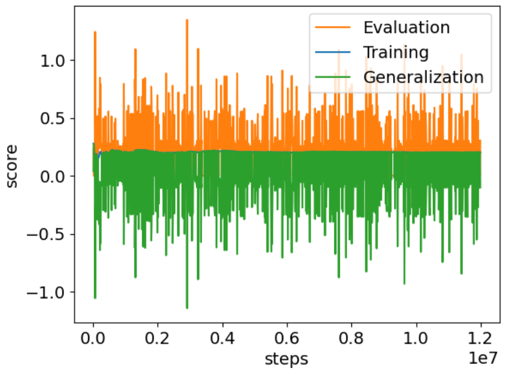
(g) dodgeball



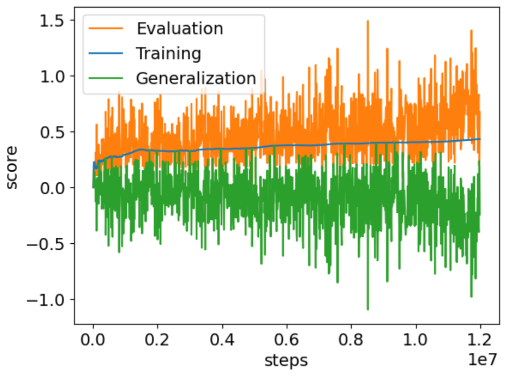
(h) fruitbot



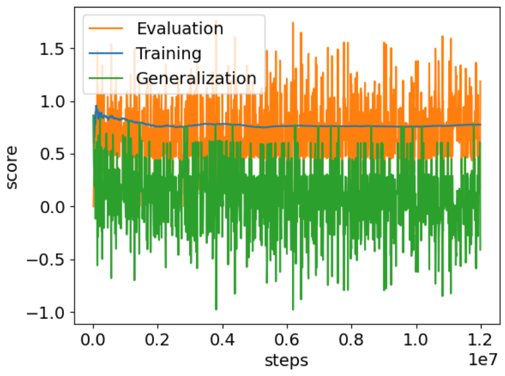
(i) heist



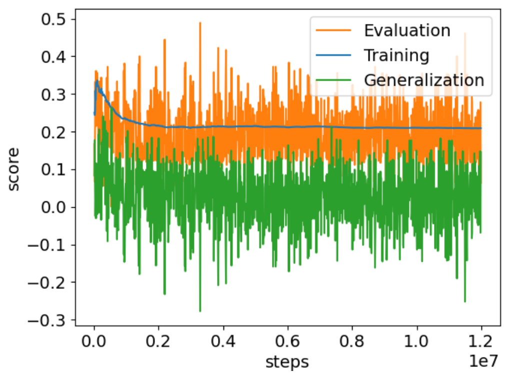
(j) jumper



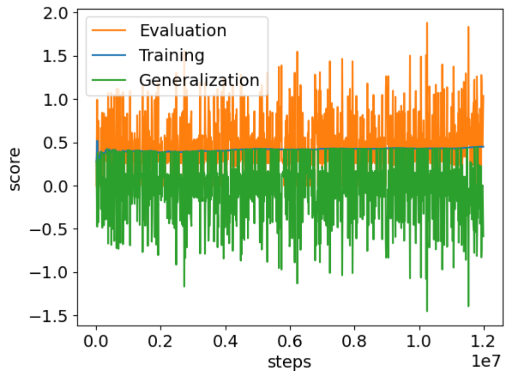
(k) leaper



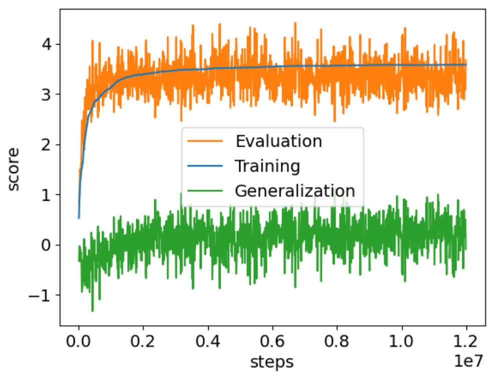
(l) maze



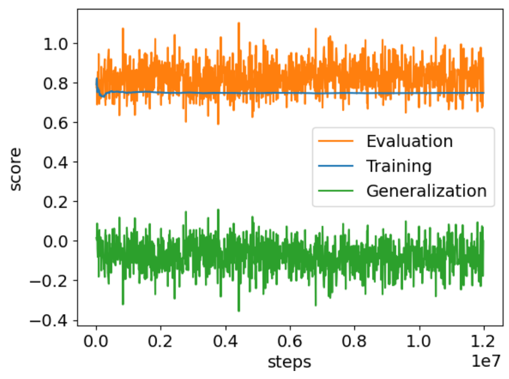
(m) miner



(n) ninja



(o) plunder

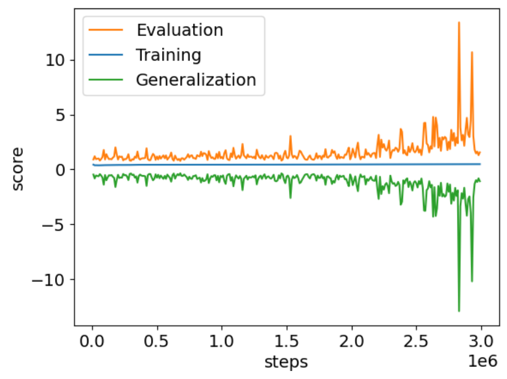


(p) starpilot

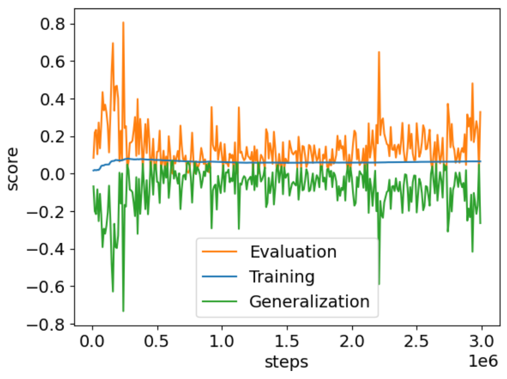
Figure A.4.: *REINFORCE* performance - hard setting

A.5. DQN^M - Easy

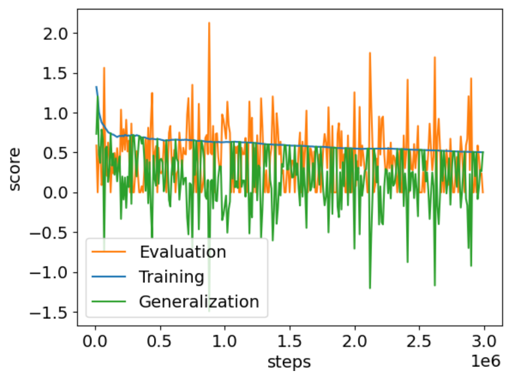
Figure A.5 shows the training, evaluation, and generalization performance of the DQN^M agent for all 16 environments in the easy setting.



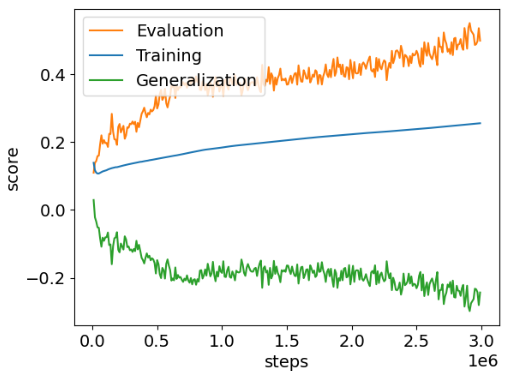
(a) bigfish



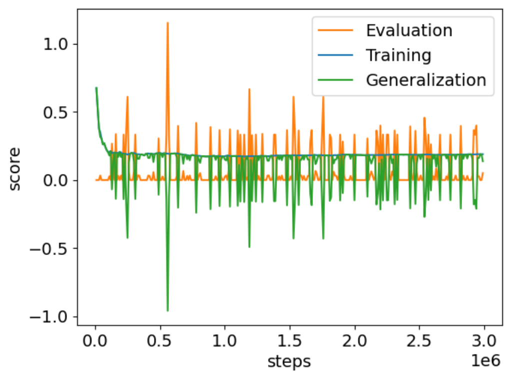
(b) bossfight



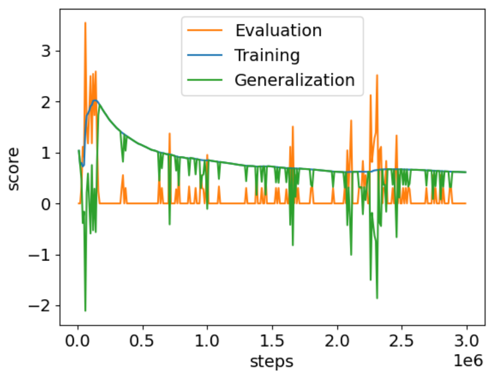
(c) cavefyler



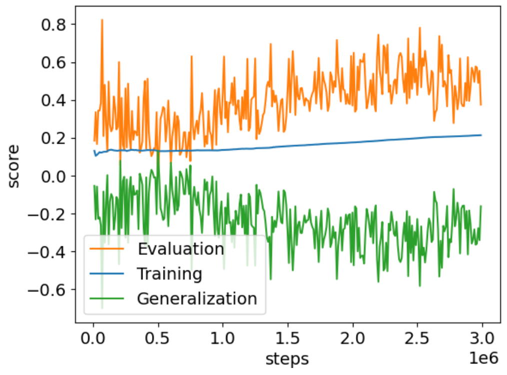
(d) chaser



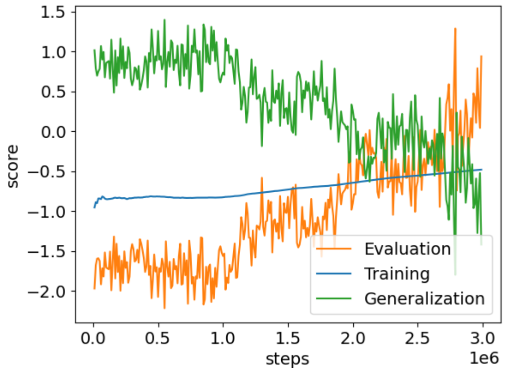
(e) climber



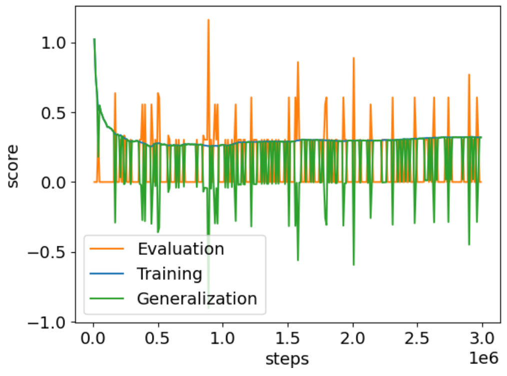
(f) coinrun



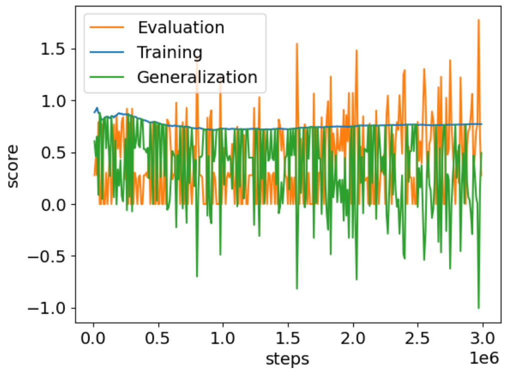
(g) dodgeball



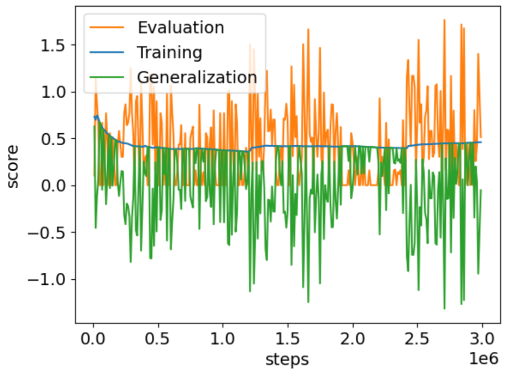
(h) fruitbot



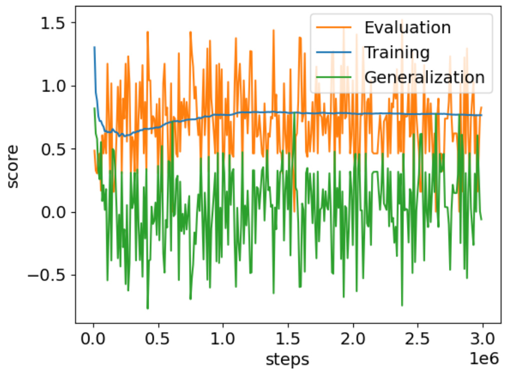
(i) heist



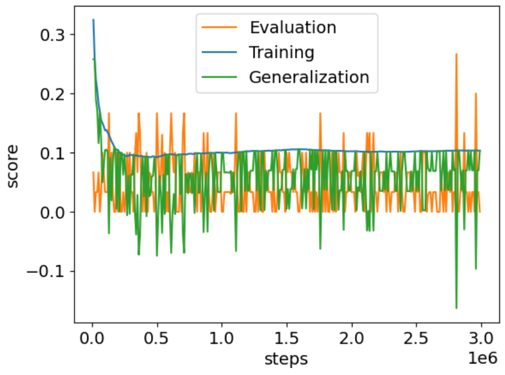
(j) jumper



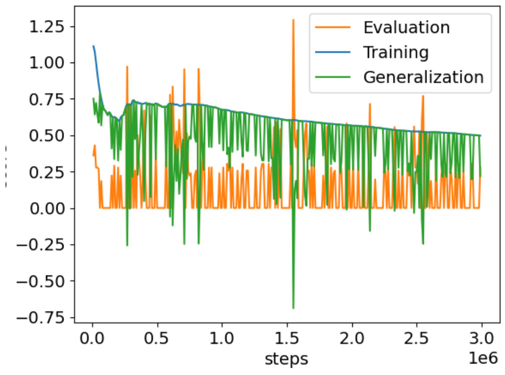
(k) leaper



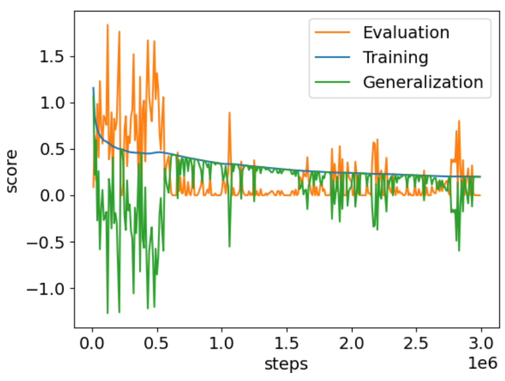
(l) maze



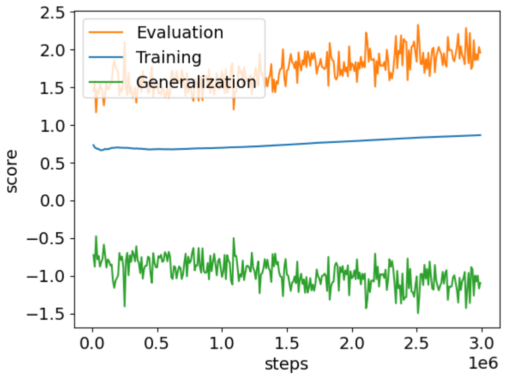
(m) miner



(n) ninja



(o) plunder

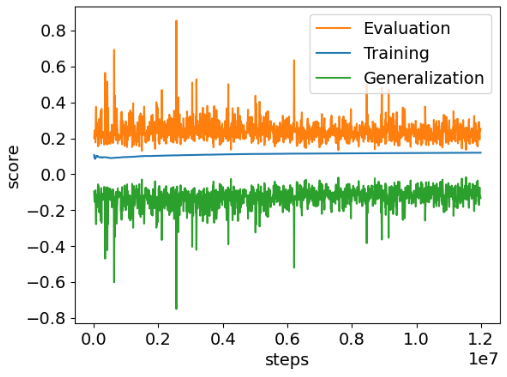


(p) starpilot

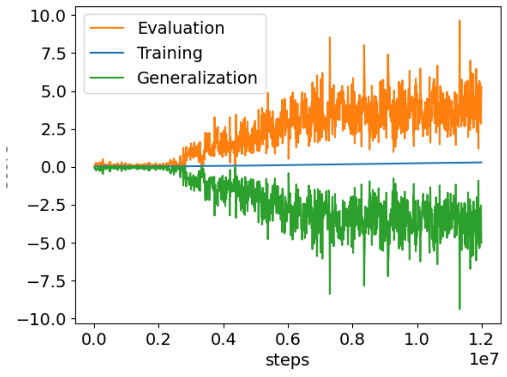
Figure A.5.: DQN^M performance - easy setting

A.6. DQN^M - Hard

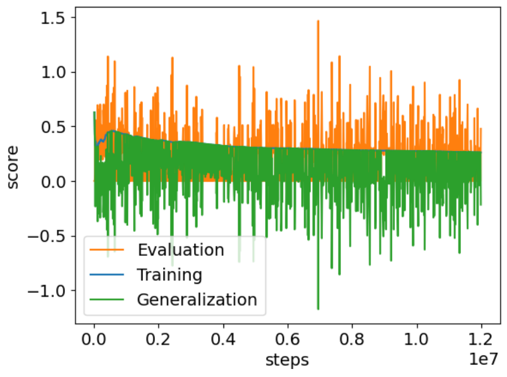
Figure A.6 shows the training, evaluation, and generalization performance of the DQN^M agent for all 16 environments in the hard setting.



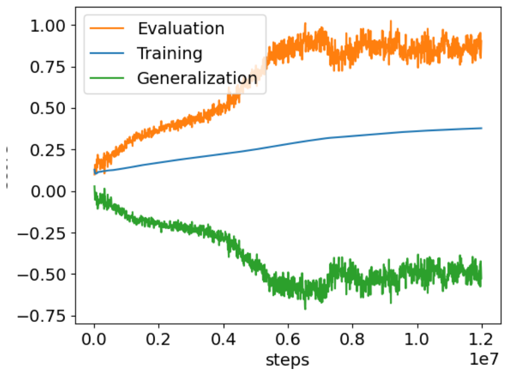
(a) bigfish



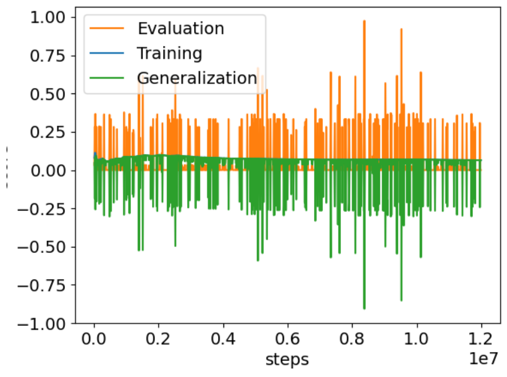
(b) bossfight



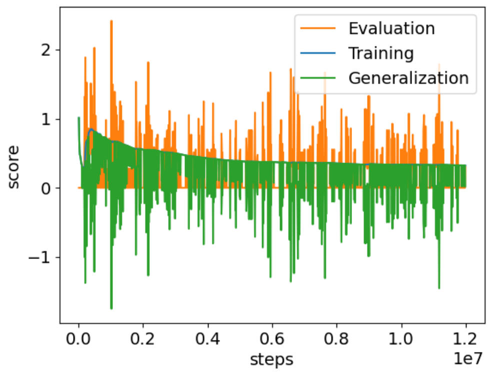
(c) cavefyler



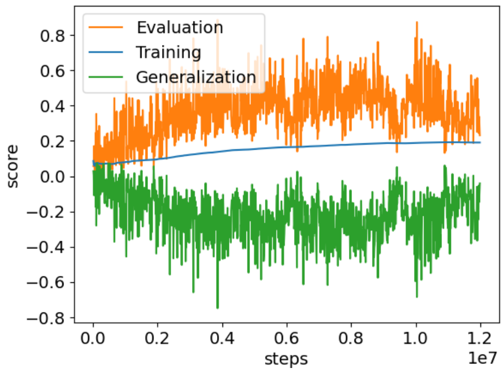
(d) chaser



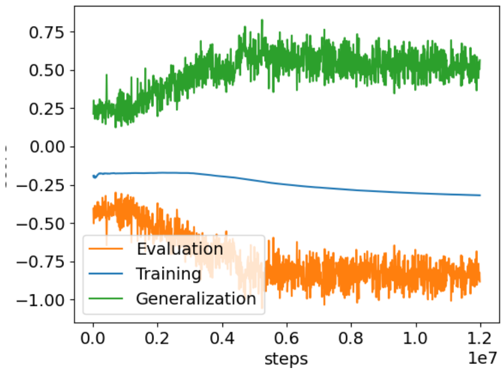
(e) climber



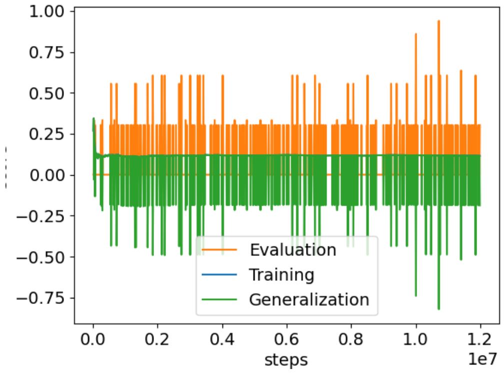
(f) coinrun



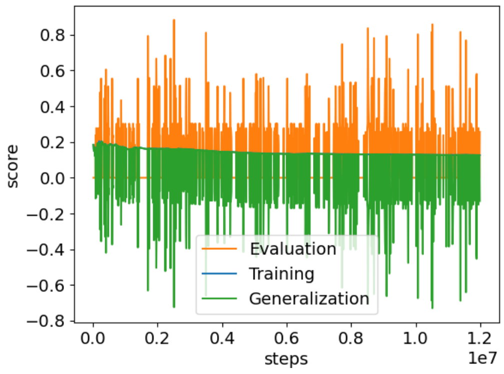
(g) dodgeball



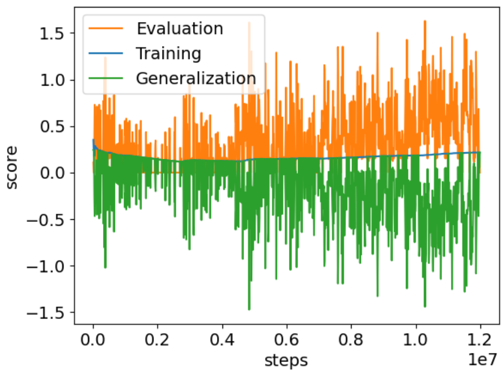
(h) fruitbot



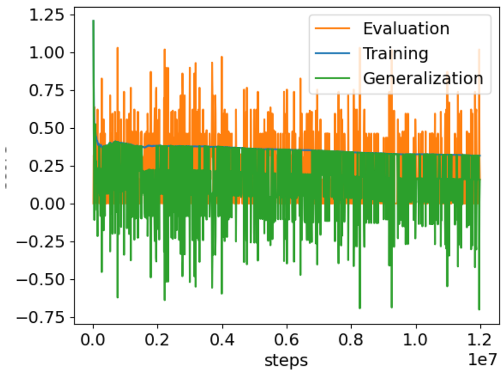
(i) heist



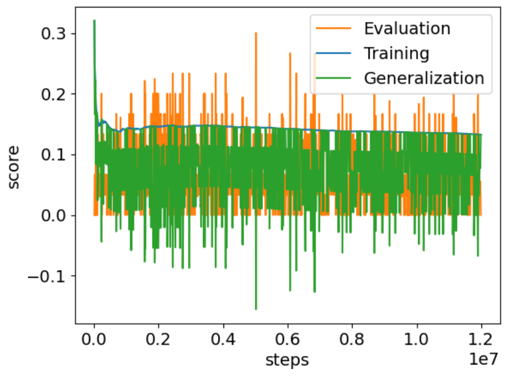
(j) jumper



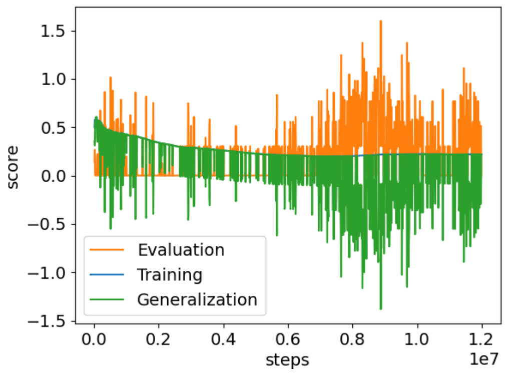
(k) leaper



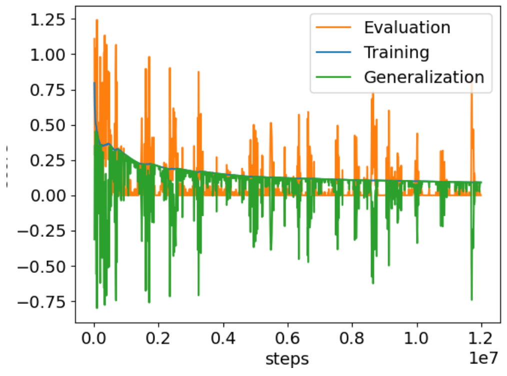
(l) maze



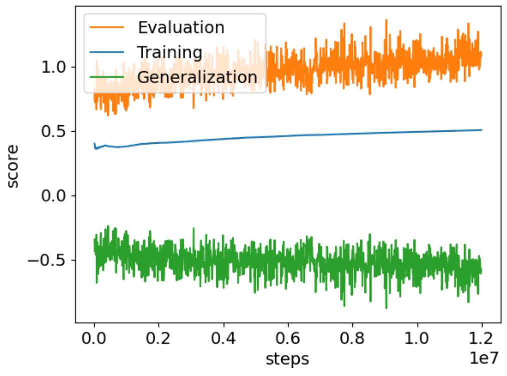
(m) miner



(n) ninja



(o) plunder

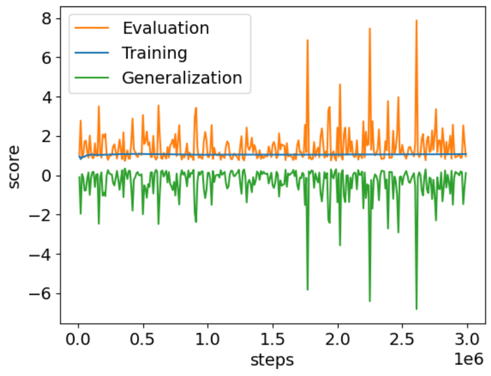


(p) starpilot

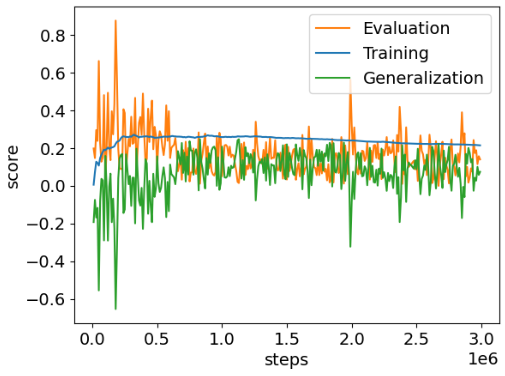
Figure A.6.: DQN^M performance - hard setting

A.7. DQN^P - Easy

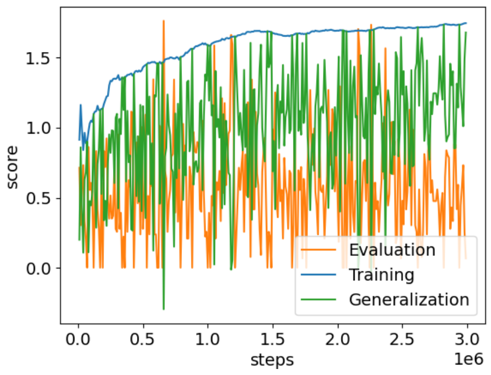
Figure A.7 shows the training, evaluation, and generalization performance of the DQN^P agent for all 16 environments in the easy setting.



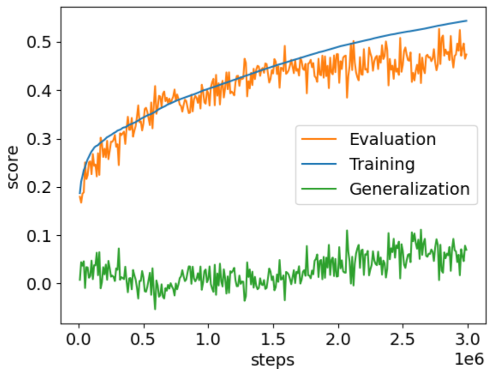
(a) bigfish



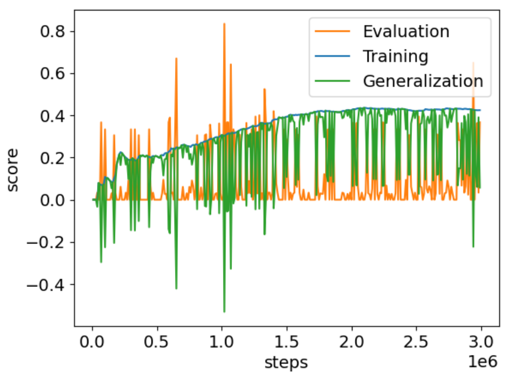
(b) bossfight



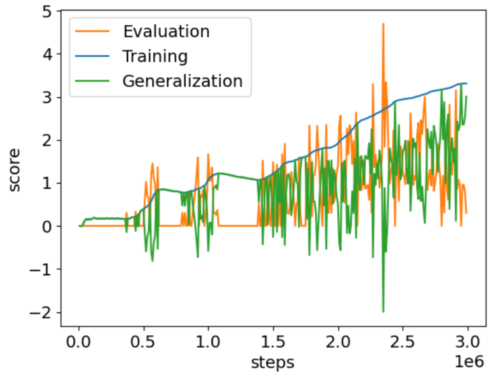
(c) cavefyler



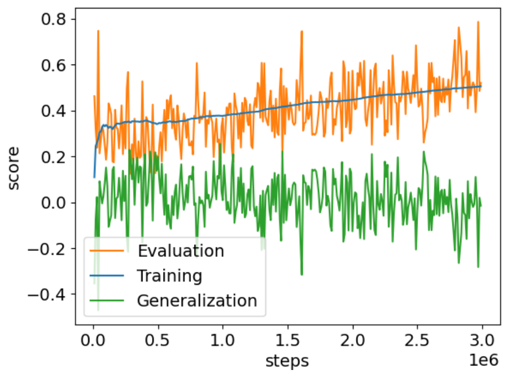
(d) chaser



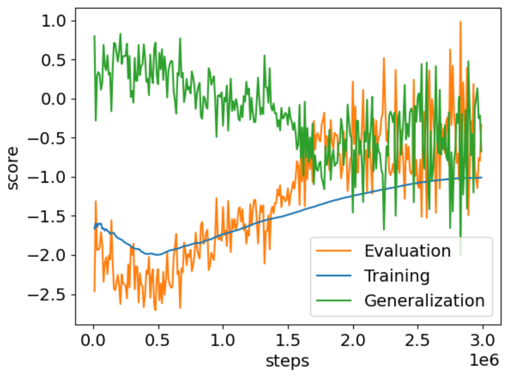
(e) climber



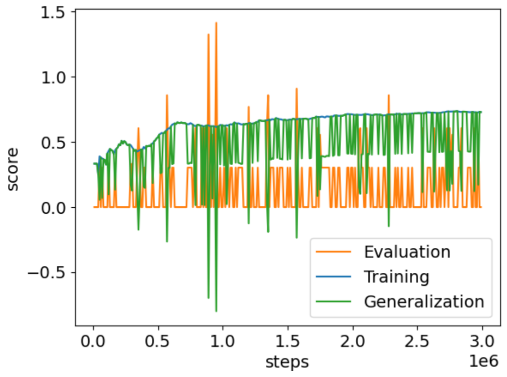
(f) coinrun



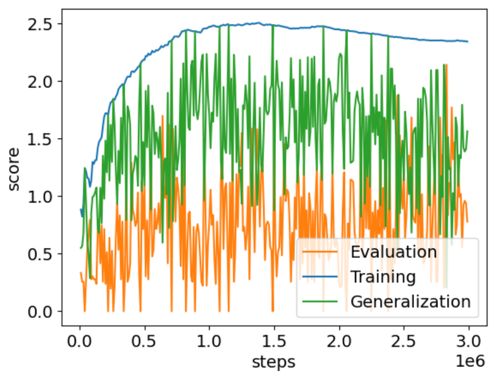
(g) dodgeball



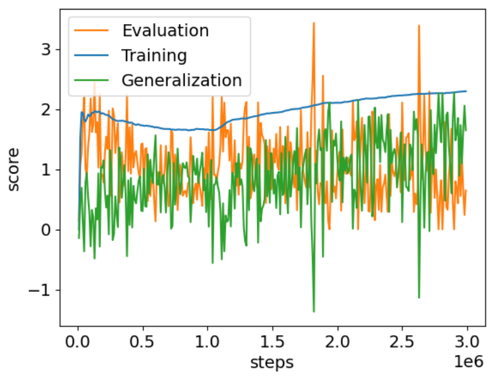
(h) fruitbot



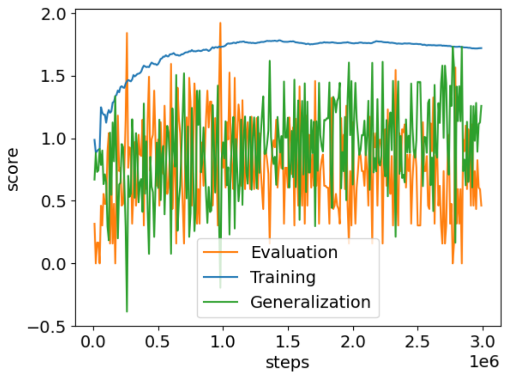
(i) heist



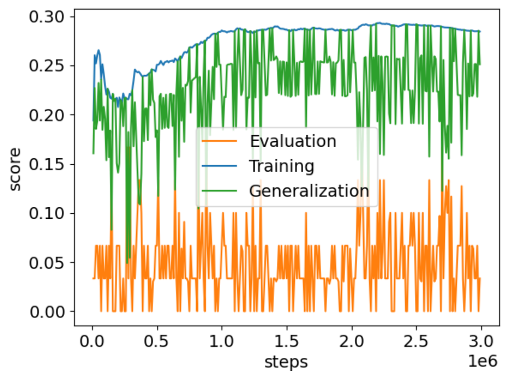
(j) jumper



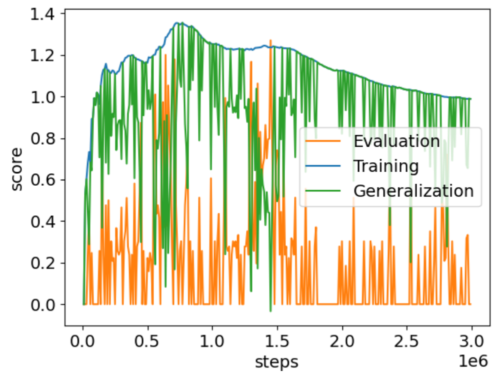
(k) leaper



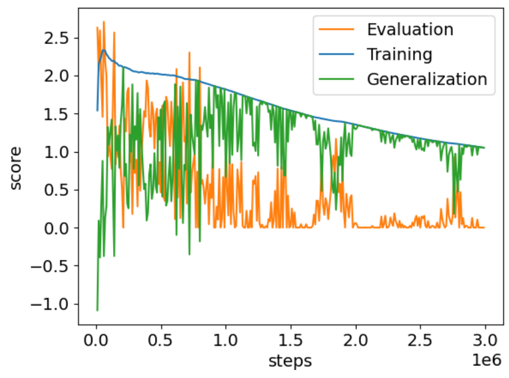
(l) maze



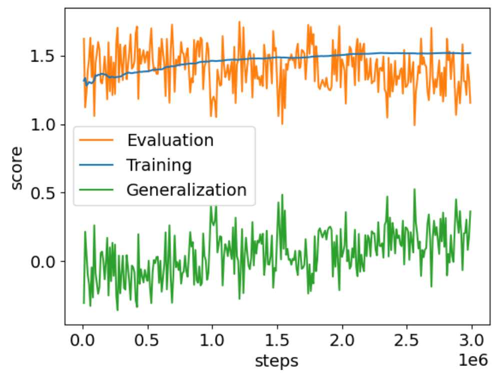
(m) miner



(n) ninja



(o) plunder

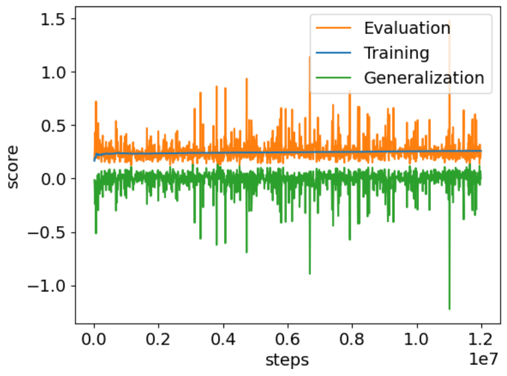


(p) starpilot

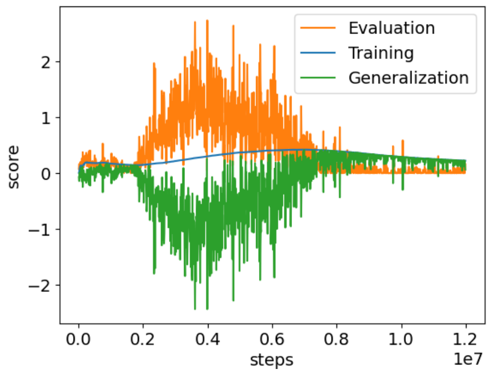
Figure A.7.: DQN^P performance - easy setting

A.8. DQN^P - Hard

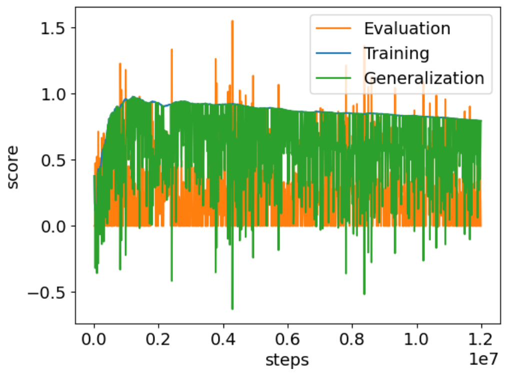
Figure A.8 shows the training, evaluation, and generalization performance of the DQN^P agent for all 16 environments in the hard setting.



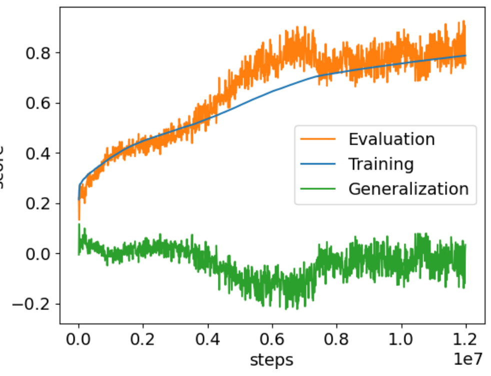
(a) bigfish



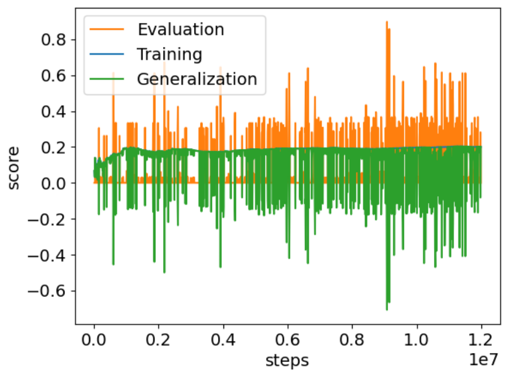
(b) bossfight



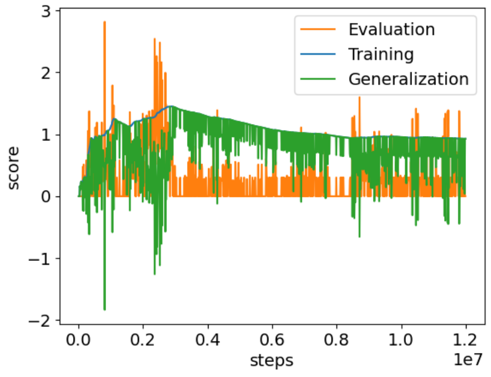
(c) cavefyler



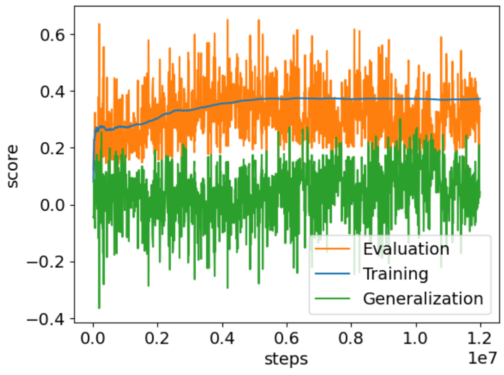
(d) chaser



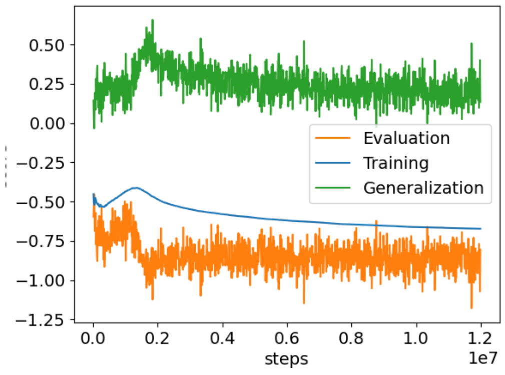
(e) climber



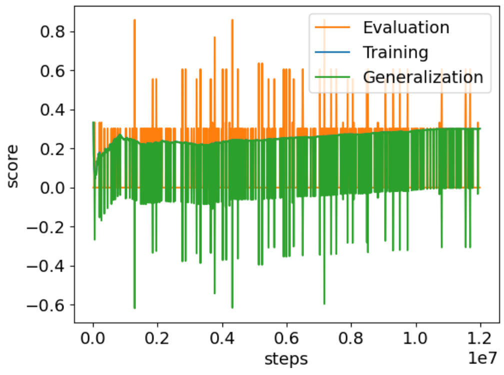
(f) coinrun



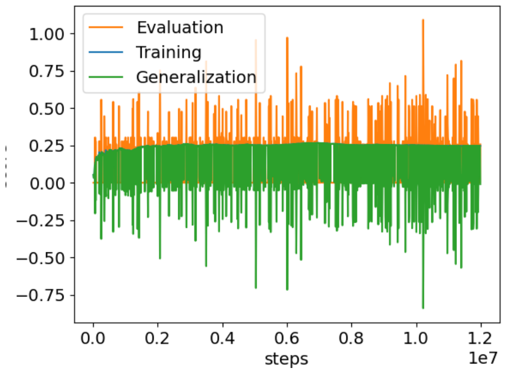
(g) dodgeball



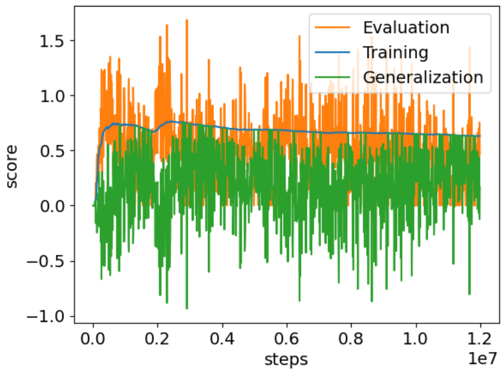
(h) fruitbot



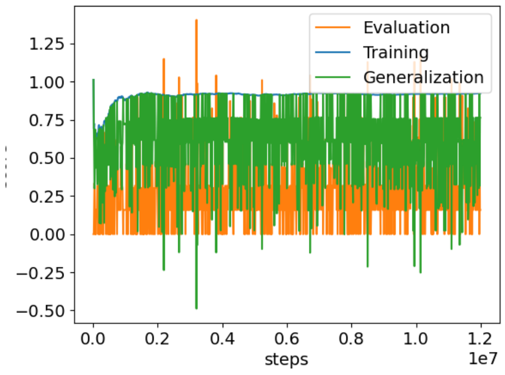
(i) heist



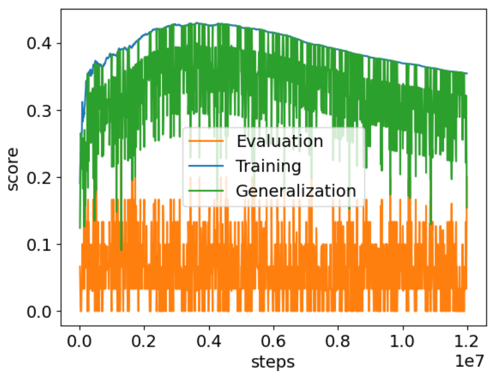
(j) jumper



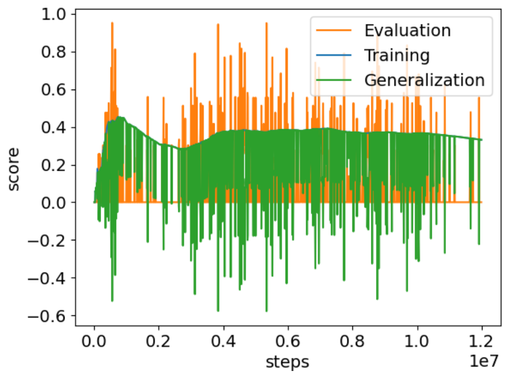
(k) leaper



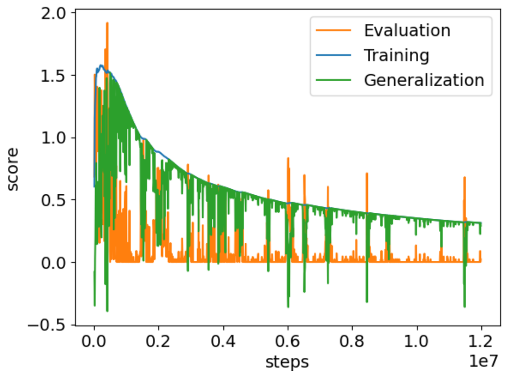
(l) maze



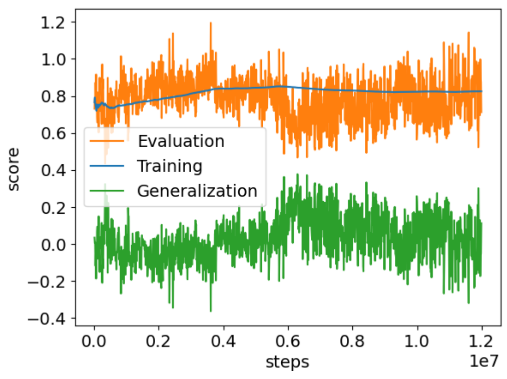
(m) miner



(n) ninja



(o) plunder



(p) starpilot

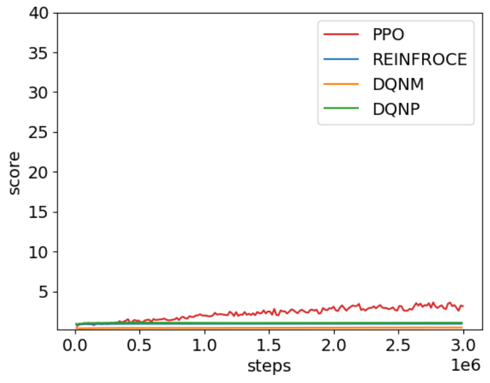
Figure A.8.: DQN^P performance - hard setting

B. Training Performance Including Maximal Score

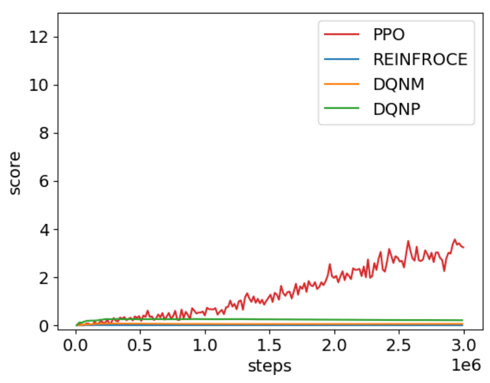
We additionally provide the visualized training progresses according to the maximal score of each environment. While the actual training progress is hard to see in most cases, it helps us assess the performance compared to the achievable performance.

B.1. Easy Setting

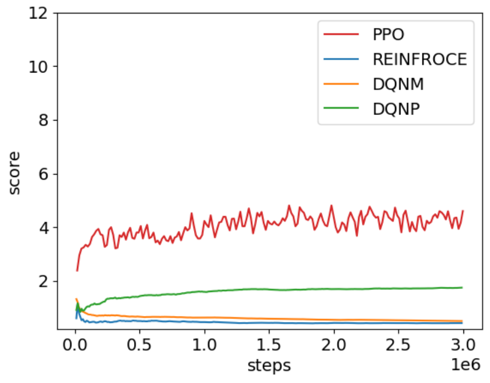
Figure B.1 shows the training performance compared to the maximal score of all agents for all 16 environments in the easy setting.



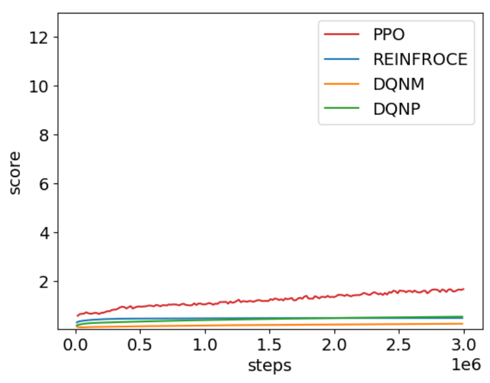
(a) bigfish



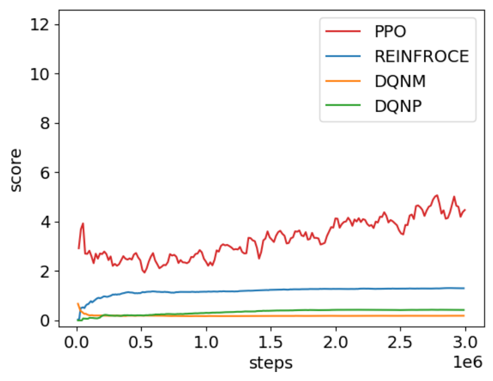
(b) bossfight



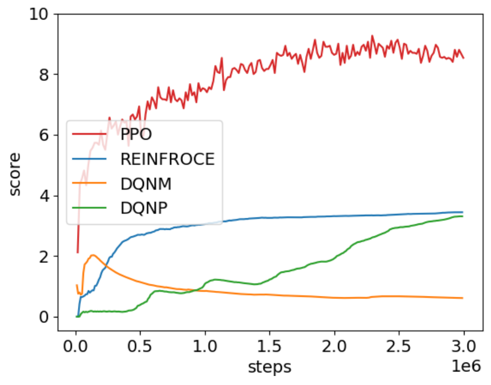
(c) cavefyler



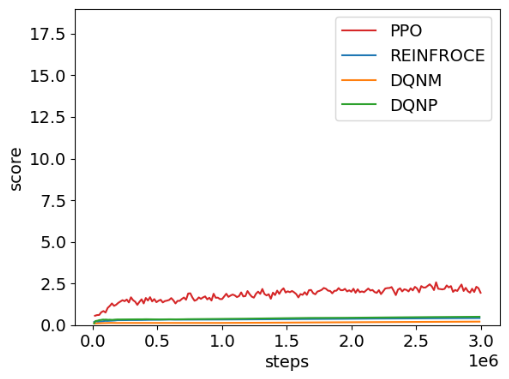
(d) chaser



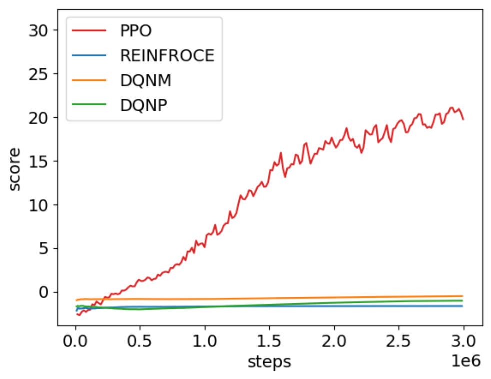
(e) climber



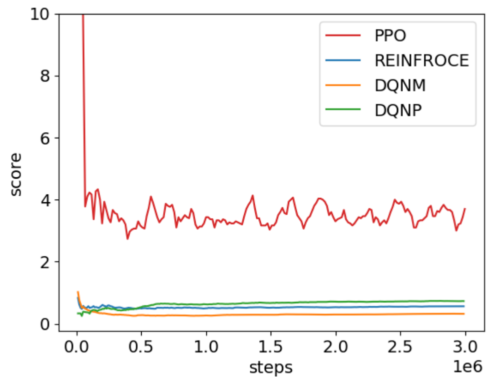
(f) coinrun



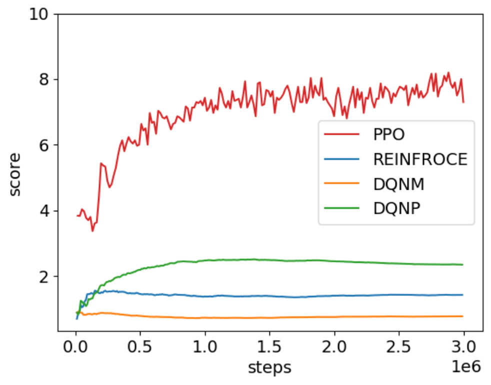
(g) dodgeball



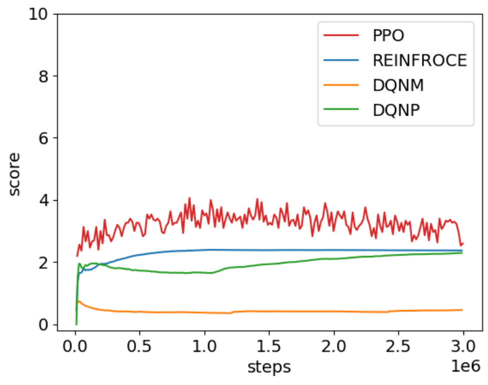
(h) fruitbot



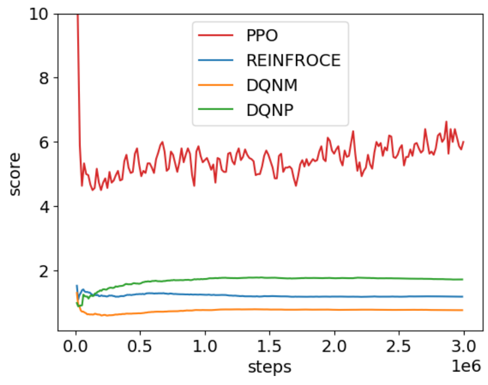
(i) heist



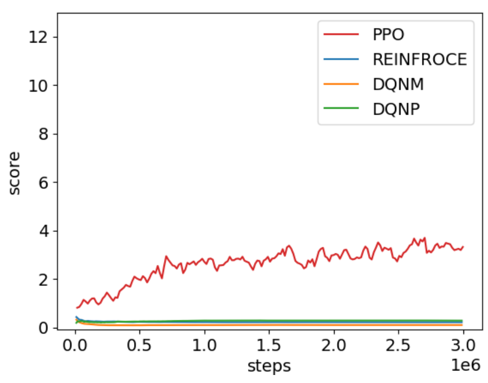
(j) jumper



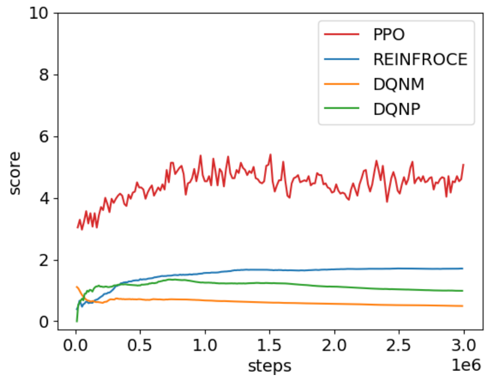
(k) leaper



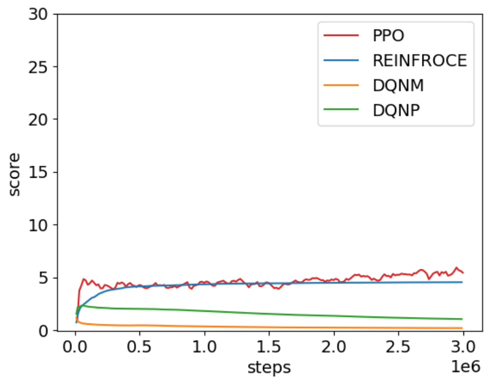
(l) maze



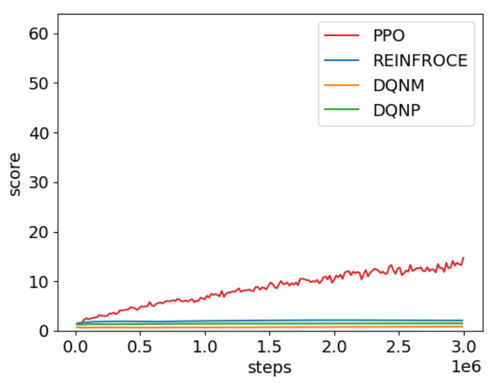
(m) miner



(n) ninja



(o) plunder

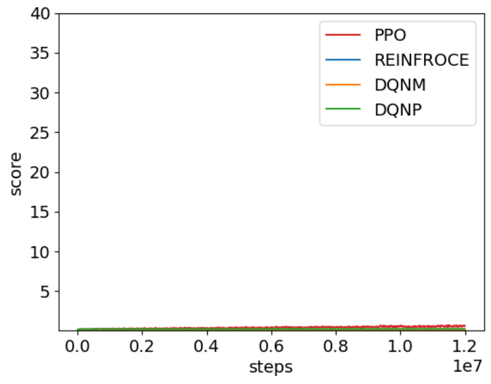


(p) starpilot

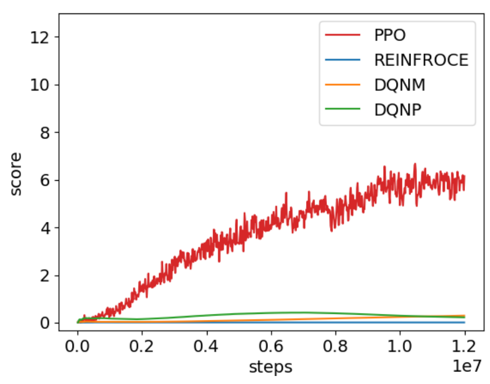
Figure B.1.: Training performance - easy setting

B.2. Hard Setting

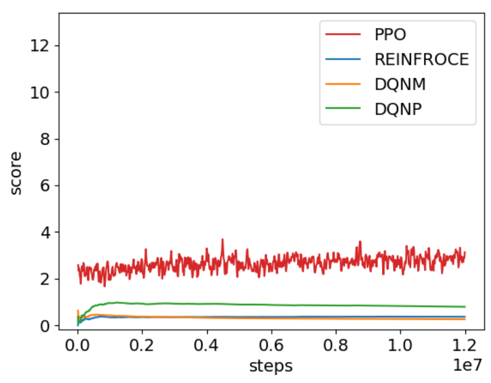
Figure B.2 shows the training performance compared to the maximal score of all agents for all 16 environments in the easy setting.



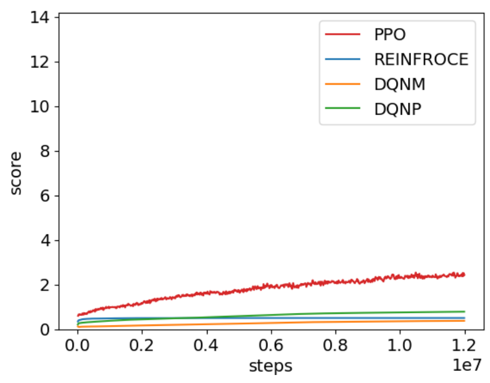
(a) bigfish



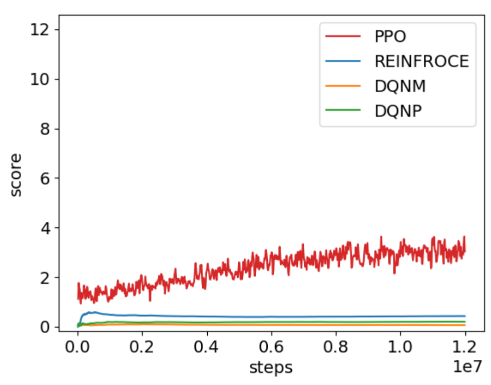
(b) bossfight



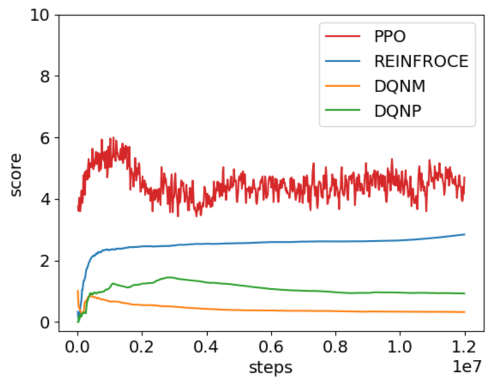
(c) cavefyler



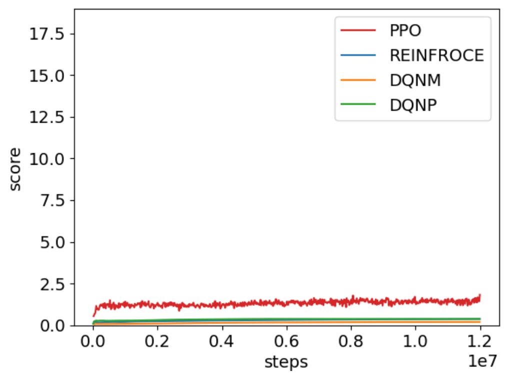
(d) chaser



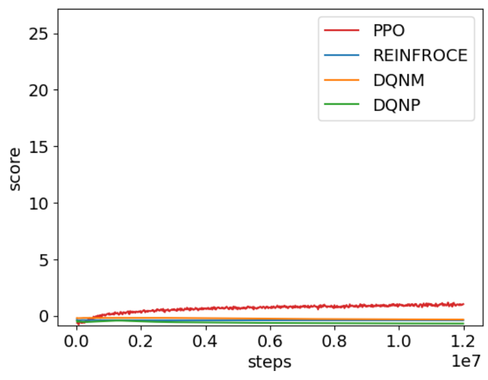
(e) climber



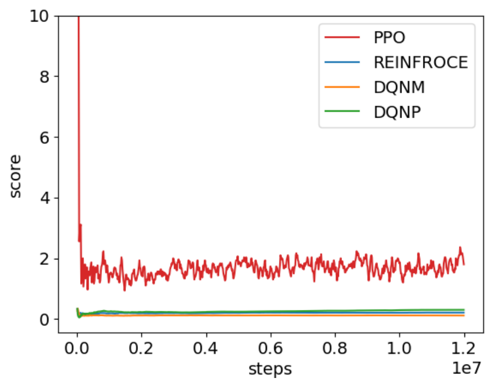
(f) coinrun



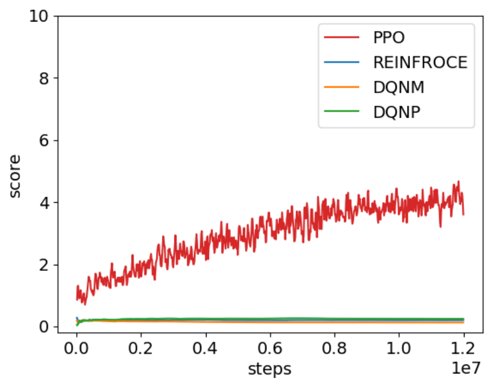
(g) dodgeball



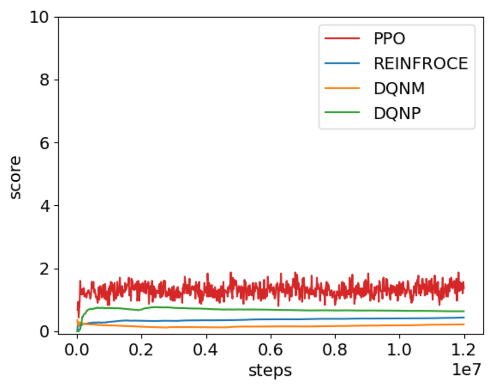
(h) fruitbot



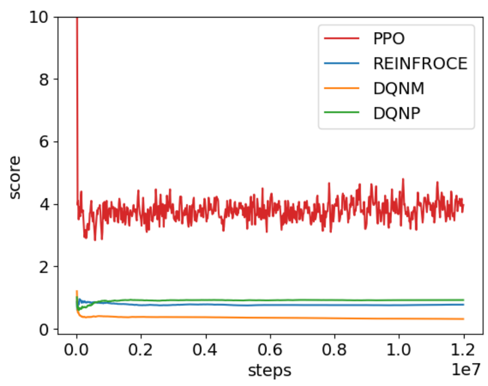
(i) heist



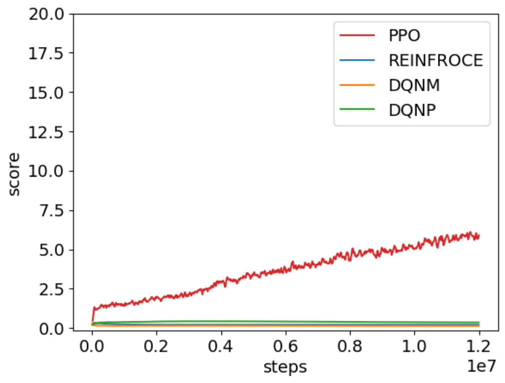
(j) jumper



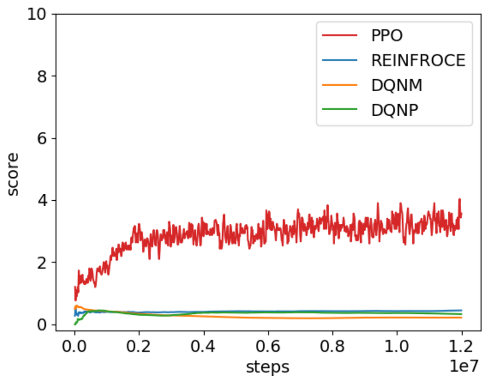
(k) leaper



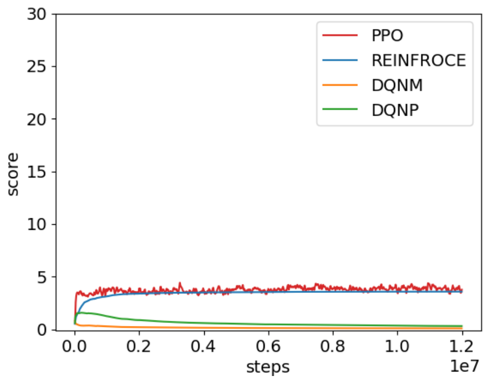
(l) maze



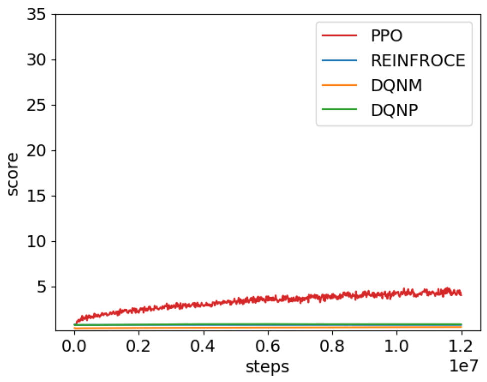
(m) miner



(n) ninja



(o) plunder



(p) starpilot

Figure B.2.: Training performance - hard setting