

Evaluating Population Based Reinforcement Learning for Transfer Learning

Evaluierung von populationsbasiertem Reinforcement Learning für Transfer Learning

Master thesis by Jan Frederik Liebig (Student ID: 2325110)

Date of submission: November 29, 2021

1. Review: Prof. Dr. Kristian Kersting

2. Review: Johannes Czech

Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department

Artificial Intelligence and
Machine Learning Lab

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

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

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

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

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

Darmstadt, 29. November 2021

J. F. Liebig

Abstract

The areas of application of reinforcement learning continue to grow. Many approaches to automation rely on this type of machine learning. In the broad area of robotics in particular, transfer learning is a method that is widely used to learn in a virtual environment and then transfer what has been learned to real systems. A well-known example of this is automated driving. The efficient creation of these systems and the associated hyperparameter search is a challenge for the future development of reinforcement learning. Since the data generation often takes up a large part of the runtime, efficient ways must be found to utilize this data as well as possible. Frequent retraining to find better hyperparameters should be avoided as this can greatly prolong the training. Hyperparameter independent methods for creating reinforcement learning agents must be found and evaluated.

Population based training methods have shown success in training neural networks in the past. This type of training adjusts the hyperparameters again and again during training and optimizes them independently. Furthermore, models trained in this way are more robust. It is therefore a logical next step to combine these two methods of population based training and transfer learning. In this elaboration, we evaluate the influence of population based training on transfer learning in the context of reinforcement learning. It seems that population based training methods also have a positive influence in the area of transfer learning. In addition, it can be seen that both the agents' evaluation method and their initialization can greatly improve or decrease performance, especially in the case of inconsistent learning behavior.

Keywords: Deep Reinforcement Learning, Transfer Learning, Population Based Training, Preservative Rating, Gridworld

Implementation can be found at <https://git.io/JMBEu> .

Zusammenfassung

Die Anwendungsbereiche von Reinforcement Learning wachsen immer weiter. Viele Ansätze in der Automatisierung verlassen sich auf diese Art des maschinellen Lernens. Gerade im weit gefassten Bereich der Robotik ist auch das Transfer Learning eine viel genutzte Methode um in einer virtuellen Umgebung zu lernen und dann das Gelernte auf reale Systeme zu übertragen. Ein bekanntes Beispiel hierfür ist das automatisierte Fahren. Eine Herausforderung für die zukünftige Entwicklung des Reinforcement Learning ist das effiziente Erstellen dieser Systeme und die dazugehörige Hyperparametersuche. Da die Datengeneration oft einen großen Anteil der Laufzeit beansprucht müssen effiziente Wege gefunden werden diese Daten so gut wie möglich auszunutzen. Häufiges neu trainieren um bessere Hyperparameter zu finden sollte vermieden werden da dies das Training stark verlängern kann. Es müssen Hyperparameter unabhängige Methoden zum Erstellen von Reinforcement Learning Agenten gefunden und evaluiert werden.

Populationsbasierte Trainingsmethoden haben in der Vergangenheit Erfolge bei dem trainieren von Neuralen Netzwerken gezeigt. Diese Art des Trainings passt die Hyperparameter während des Trainings immer wieder neu an und optimiert diese so selbstständig. Des Weiteren sind so trainierte Modelle robuster. Es ist daher ein logischer nächster Schritt diese beiden Verfahren von populationsbasiertem Training und Transfer Learning zu kombinieren. In dieser Ausarbeitung evaluieren wir den Einfluss von populationsbasiertem Training auf Transfer Learning im Umfeld des Reinforcement Learnings. Es scheint, dass populaionsbasierte Trainingsmethoden auch im Bereich des Transfer Learnings einen positiven Einfluss haben. Zusätzlich zeigt sich, dass sowohl die Bewertungsmethode der Agenten, als auch deren Initialisierung vor allem bei inkonsistentem Lernverhalten die Leistung stark verbessern oder verschlechtern können.

Schlüsselworte: Deep Reinforcement Learning, Transfer Learning, Population Based Training, Preservative Rating, Gridworld

Implementation kann gefunden werden unter <https://git.io/JMBEu> .

Contents

1	Introduction	10
2	Foundations	11
2.1	Reinforcement Learning	11
2.1.1	Reinforcement Learning Problem	11
2.1.2	Environment	11
2.1.3	Reward	13
2.1.4	State	13
2.1.5	Markov Decision Processes	15
2.1.6	Exploration and Exploitation	15
2.2	Taxonomy of Reinforcement Learning	15
2.2.1	Model Based and Model Free	16
2.2.2	Policy and Value	16
2.2.3	Offline and Online Learning	18
2.2.4	Algorithms	19
2.3	Population Based Training	22
2.4	Transfer Learning	25
2.5	Related Work	25
3	Concept and Idea	27
3.1	Gridworld	27
3.1.1	Tiles and States	27
3.1.2	Reward	28
3.1.3	Modes	30
3.2	Computational Setup	31
3.2.1	Neural Network Architecture	31
3.2.2	Reinforcement Learning Setup	32
3.2.3	Container	33
3.3	Population Based Training	34
3.3.1	Rating	36
3.3.2	Mutation	36
3.4	Transfer Learning	37
4	Evaluation	39
4.1	Reinforce	39
4.2	Deep Q-Learning	40
4.3	Proximal Policy Optimization	41
4.4	Preservative Rating	42



4.5	Transfer Population	45
5	Discussion	47
6	Conclusion	49
7	Appendix	52
7.1	Reinforce	52
7.2	Deep Q-Network (DQN)	55
7.3	Proximal Policy Optimization (PPO)	58
7.4	Transfer Population	61

List of Figures

2.1	Goal of the reinforcement learning agent.	11
2.2	The interaction of an agent and an environment.	12
2.3	The agent state S_t at time step t	13
2.4	The complete interaction between an agent and an environment.	14
2.5	Example for a policy based approach.	16
2.6	Example for a value function.	17
2.7	Population based training concept.	22
2.8	A population with different models.	23
2.9	A population from one origin model.	24
3.1	Rendered images of the different tiles.	27
3.2	Rendered observation.	28
3.3	A plot for the reward function	29
3.4	Rendered maps of both gridworld modes.	30
3.5	The neural network architecture.	31
3.6	General Reinforcement learning training loop.	32
3.7	General training loop for population based training.	34
4.1	The mean reward for the Reinforce algorithm.	39
4.2	The best reward for the Reinforce algorithm	40
4.3	The mean reward for the DQN algorithm.	40
4.4	The best reward for the DQN algorithm	41
4.5	The mean reward for the PPO algorithm.	41
4.6	The best reward for the PPO algorithm	42
4.7	The mean reward without (left) and with (right) preservative rating.	42
4.8	The mean reward per iteration without (left) and with (right) Preservative Rating.	43
4.9	The mean accuracy without (left) and with (right) preservative rating.	43
4.10	The mean accuracy per iteration without (left) and with (right) preservative rating.	44
4.11	The accuracy per iteration without preservative rating.	44
4.12	The mean reward with the different population initialization methods.	45
4.13	The mean accuracy with the different population initialization methods.	45
4.14	The mean reward and the mean accuracy split into two cluster.	46
7.1	The mean accuracy for the Reinforce algorithm.	52
7.2	The best accuracy for the Reinforce algorithm	52
7.3	The mean accuracy per iteration for the Reinforce algorithm.	53
7.4	The best accuracy per iteration for the Reinforce algorithm	53
7.5	The mean reward per iteration for the Reinforce algorithm.	54
7.6	The best reward per iteration for the Reinforce algorithm	54

7.7	The mean accuracy for the DQN algorithm.	55
7.8	The best accuracy for the DQN algorithm	55
7.9	The mean accuracy per iteration for the DQN algorithm.	56
7.10	The best accuracy per iteration for the DQN algorithm	56
7.11	The mean reward per iteration for the DQN algorithm.	57
7.12	The best reward per iteration for the DQN algorithm	57
7.13	The mean accuracy for the PPO algorithm.	58
7.14	The best accuracy for the PPO algorithm	58
7.15	The mean accuracy per iteration for the PPO algorithm.	59
7.16	The best accuracy per iteration for the PPO algorithm	59
7.17	The mean reward per iteration for the PPO algorithm.	60
7.18	The best reward per iteration for the PPO algorithm	60
7.19	The mean steps per game with the different population initialization methods.	61
7.20	The mean reward per iteration with the different population initialization methods.	61
7.21	The mean accuracy with the different population initialization methods.	62
7.22	The steps per iteration reward with the different population initialization methods.	62

Listings

3.1	Calculating the reward of the gridworld.	29
3.2	Making a Gridworld in both used modes.	30
3.3	The loop of the PPO Actor-Critic.	33
3.4	The population based trainig loop for reinforce.	35
3.5	The rating function.	36
3.6	The mutation step.	37
3.7	The Transfer Learning for a single agent.	37
3.8	Population Based Transfer Learning.	38

1 Introduction

Machine learning already plays an important role in the modern world and the influence is expected to increase. Especially the usage of reinforcement learning in combination with transfer learning is likely to rise. It is this combination of techniques that allows to train an agent in a virtual environment and transfer it into the real world. There are several use cases for this interaction. One very important of these use cases is within the wide field of robotics. In addition to many other areas, this includes automated driving. One big problem of reinforcement learning is the data generation. Unlike other deep learning techniques reinforcement learning relies on a reply of the environment and in some cases this can take a while. It is important to use the generated data as efficiently as possible. Hyperparameter search can be a slow process, this and the data generation problem can slow the reinforcement learning process and the building of good performing agents down a lot. Building reliable models through reinforcement learning will be a big challenge in the future.

Population based training has been shown as efficient method to build robust models. This method is independent from the selected hyperparameters since it optimizes them itself.

Transfer learning suffers twice as much from the problem of data generation combined with hyperparameter search. For transfer learning not only one set of hyperparameters has to be found, but after training with the first environment the agent has to transfer in a second environment.

In theory transfer learning should profit from the extra stability population based training provides. Population based training can also help within transfer learning and let an agent better fit in the new environment.

In this elaboration we try to answer the question whether the combination of these two techniques, transfer learning and population based training, will result in the expected increase in stability and robustness. If that is the case we also aim to answer the follow-up question, does the stability of population based training justify the increased complexity for transfer learning?

2 Foundations

2.1 Reinforcement Learning

2.1.1 Reinforcement Learning Problem

Reinforcement learning is the *learning by doing* kind of machine learning. The basic concept is to let an machine learning agent interact with an environment or a model of an environment and learn by trial and error [12]. Reinforcement learning is a kind of unsupervised learning, there is no supervisor that tells the agent whether the action was right or wrong. The environment only sends a reward signal as feedback and the goal of the agent is to maximize the future reward R [25]. The reward R_t is a scalar value the agent receives each time step t . So the goal to maximize the future reward is to maximize the cumulative reward.

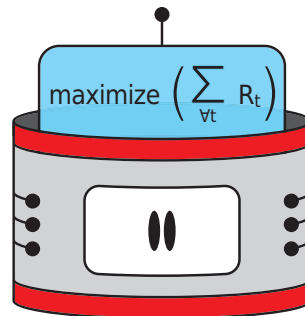


Figure 2.1: Maximizing the commutative future reward, called *return*, is the goal of the agent in reinforcement learning [1].

We call the cumulative discounted future reward the shown as in Figure 2.1 the *return*.

2.1.2 Environment

In reinforcement learning the agent learns by interacting with an environment. The agent learns to take actions in specific state to yield the most future reward. The environment changes with the actions taken from the agent, the only way for the agent to interact with the environment is through the actions, but the environment may also change itself or throughout the time.

Reinforcement learning can be applied to a wide variety of use cases [16]. To train a specific use case an agent just needs a suitable environment. An environment can be anything an agent can interact with.

Just to get a picture of what reinforcement learning can be capable of we will be looking at some examples. Some of the environments are games, because games are natural environments. Games have a clear action space and most games have a easy identifiable reward.

DeepMind trained an agent to play different old Atari games [19] and let an agent play Starcraft on Grand-master level [31]. OpenAI had a group of agents win in a Dota 2 [3] match versus humans. The classic games like chess [22], poker [14], go [23], and backgammon [28] are also environments for reinforcement learning. But there are more use cases for reinforcement learning than just games. Great successes in the development of self-driving cars can be traced back to reinforcement learning. An reinforcement learning agent can learn to manage in investment portfolio [6] and help in medical research [18].

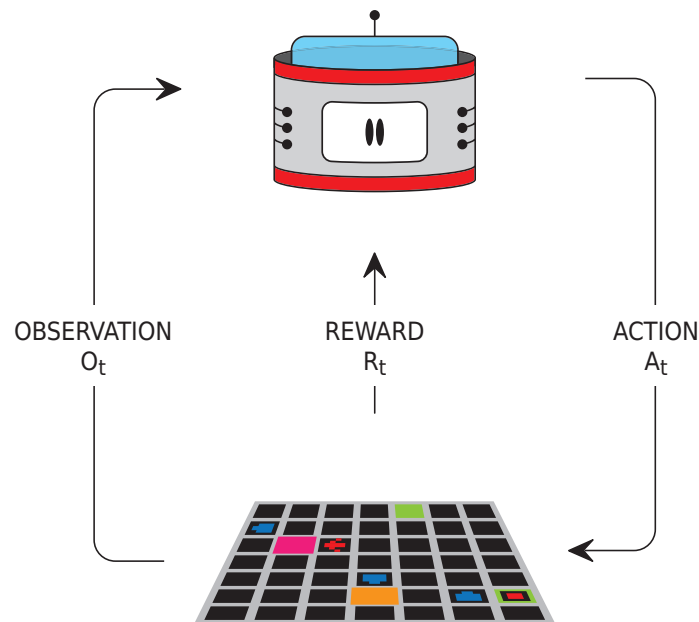


Figure 2.2: This is a simplified view of the agents interaction with the environment. Every time step t the agent can send an action A_t to the environment and receives a new observation O_t and R_t [1].

Figure 2.2 shows the simplified interaction of the agent with the environment. Through actions the agent can interact with the environment, it can be seen as the world in which the agent exists. The agent can observe the environment and only manipulate it through the action channel, but the agent cannot change the rules of the environment.

These different environments can allow different actions. Often we call the set of actions the *action space*. There are two main distinctions between action spaces, discrete action spaces and continuous action spaces. A discrete action space is a finite set of given actions, a number of different moves available for the agent. On the other hand a continuous action space consists of real-valued actions. This kind of action space is common for robotic systems.

Gridworld

We will work with a gridworld environment. A gridworld is basically a 2d grid. Each tile can have a specific effect on the game, the player can observe the world around him in a limited range. The goal of this environment is to reach a certain goal tile. To maximize the reward the player has to reach this tile as fast as possible, this gridworld has a discrete action space.

2.1.3 Reward

The reward as a scalar can model very different situations and goals. Often the reward itself indicates how good or bad an agent is doing in a particular time step, but this is not a necessary feature for a reward. Some environments have an easy to understand reward system. Managing an investment portfolio is pretty straight forward, making money refers to a positive reward and losing some to a negative, but the agent has to learn, that sometimes taking a negative reward, for example investing, can turn into a positive one later on. To train an agent to finish a maze or gridworld with a fewest amount of steps a common reward shaping is to give the agent a negative reward, for example -1 each time step until the agent finishes. And training for example chess or backgammon the agent receives a positive or negative reward at the end of the game, depending on the result.

In Section 2.1.1 the term return was used for the discounted reward. We calculate the return with

$$r_t = R_{t+1} + \gamma^1 \cdot R_{t+2} + \dots + \gamma^{T-t-1} \cdot R_T = \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \quad . \quad (2.1)$$

$\gamma \in [0, 1]$ is the discount factor. Discounting the future rewards means valuing a immediate reward more than a delayed reward.

2.1.4 State

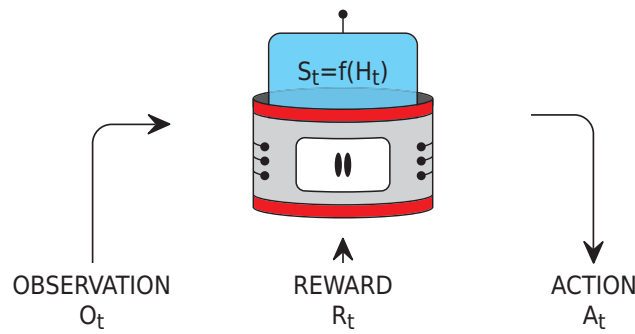


Figure 2.3: Every time step t the agent determines an agent state S_t considering the history H_t [1].

Additionally to the reward the environment outputs an observation state O_t each time step t . The observation state does not necessarily have all the information the environment has.

For example in a Poker environment the observation state does not know anything about the cards in the other players hands or the upcoming cards. So we differentiate the observation O_t in time step t from the environment state referred to as S_t^e . If the agent can observe the complete environment we say that the environment is *fully observable*.

The only observable variables for the agent are the observation state O_t , the reward R_t and the action leading to these A_{t-1} . The sequence of these variables is called the history as shown in

$$H_t = O_1, R_1, A_1, \dots, A_{t-1}, O_t, R_t \quad . \quad (2.2)$$

The agent state S_t is a function on the history H_t , Figure 2.3 shows the process in the agent to determine the next agent state S_t .

It is worth considering that the agent state can be any function over the history, S_t can only be the last observation or the whole history. The agent state is the agent's internal representation of the current environment state and is used to determine the next action.

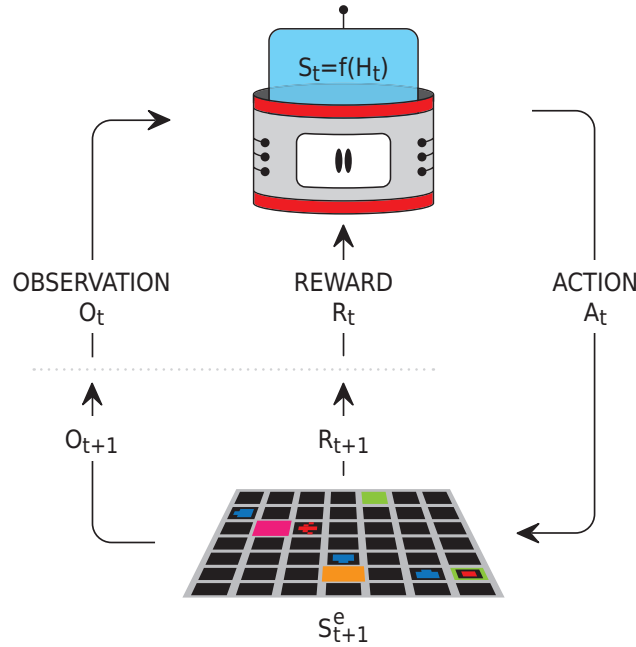


Figure 2.4: The complete interaction of the agent with the environment. The agent determines an agent state S_t considering the history H_t , using this state the agent computes an action A_t and sends this to the environment. The environment performs the action A_t , transitions to the next environment state S_{t+1}^e and transmits the next observation O_{t+1} and reward R_{t+1} [1].

We can see the complete interaction of the agent and the environment in Figure 2.4. Starting with an agent determine the next agent state S_t with the history H_t and decide the next action A_t . The environment processes this action and specify the next environment state S_t^e , the next reward R_{t+1} and the next observation O_{t+1} . The agent then inserts these values in the history h_{t+1} .

2.1.5 Markov Decision Processes

A Markov decision process is a model to formally describe an decision problem. It is common to model a reinforcement problem as a Markov decision problem [30]. To start with this we define the Markov state (also known as information state), a state S_t is a Markov state if and only if it fulfills the Markov property

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \dots, S_t] \quad . \quad (2.3)$$

This property says that a state, once it is determined, contains all information of the history. By definition the environment state S_t^e and the history H_t is Markov, also the observation of a fully observable environment is Markov. In a Markov decision process all states fulfill the Markov property and the environment is assumed to be fully observable.

A Markov decision problem is a tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$ with

- \mathcal{S} a set of all possible states
- \mathcal{A} a set of all possible actions
- p a probability transition $p(S'|S, A)$, it denotes the probability of transitioning in state S' with a given state S and action A
- r a expected reward $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ for transitioning starting in (S, A)
 $r = \mathbb{E}[R|S, A]$.
- γ a discount factor $\gamma \in [0, 1]$ to discount the rewards.

2.1.6 Exploration and Exploitation

Exploration and exploitation is a concept that is mainly used in reinforcement learning. The first step of this concept is the exploration, the idea behind this is letting an agent explore the environment. At the start of training the reinforcement learning agent does not know anything about the environment, the agent does not know what is the goal, the agent does not know what to do and the agent does not know what the actions do. In the step of exploration the agent should have a chance to explore this. In exploration step the agent does not always pick the best known action in a given situation, but instead of this the agent picks a random action. A common way to construct this is called *epsilon greedy strategy*, this stand for having a value, usually called epsilon, as proportion of picking a random action and picking the best known action. This epsilon will be decreased while training.

Exploitation is the exact opposite of this. The phase of exploitation is usually the last phase when the epsilon is close to 0%. The goal of this phase is letting the agent train with all the in the exploration phase acquired information about the environment.

2.2 Taxonomy of Reinforcement Learning

Reinforcement learning algorithms can be divided in several subgroups. We will look at the most common distinctions.

2.2.1 Model Based and Model Free

The first keyword is model. There are model based and model free systems. To anticipate, we want to note that we only use model free algorithms.

A model based system is a system that tries to internally build a model of the environment. Then the system plan the interaction with the environment based on the learned model.

Exemplary we will look at a maze the agent is trying to solve as quickly as possible. In a model based system the agent will try to create a map of the maze. The agent then can plan the movement in the maze using the learned map.

In contrast to this a model free method will not try .

2.2.2 Policy and Value

Before we can subdivide algorithms into whether they are policy based or value based are we have to have a brief look what these therms mean.

Policy

A policy, commonly called π , is a mapping or a rule for the agent to determine in a state what action to pick

$$\begin{aligned} A_t &= \pi(S_t) \\ A_t &\sim \pi(\cdot|S_t) \end{aligned} \quad (2.4)$$

In contrast to a deterministic policy a stochastic policy, mostly denoted by $\pi(A|S)$ does not return one action, it rather returns the likelihood for an actions to be picked in a specific state

$$\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1] \quad (2.5)$$

The policy is the decision maker, more or less the brain, of an reinforcement learning agent.

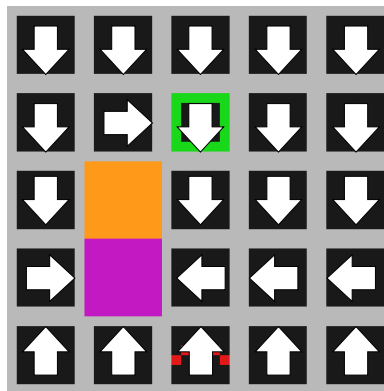


Figure 2.5: This is an exemplary policy π to solve this gridworld. Each arrow denotes the direction to move next, actions to turn in the right direction are implicit.

Figure 2.5 shows an exemplary policy for a gridworld. Each arrow shows the agent in which direction to move. This can mean several actions such as turning multiple times and a subsequent step. In deep reinforcement learning the neural network is the policy. So in this case we work with a parameterized policy

$$A_t = \pi_\theta(S_t) \quad \text{or} \quad A_t \sim \pi_\theta(\cdot|S_t) \quad (2.6)$$

where θ describes the parameter. The parameter in a neural network are the weights and the biases which will be optimized throughout the learning process.

Value

Value in this context, the context of reinforcement learning, denotes the expected return of something. A value function is consequently a function determines the expected future reward. The value can be determined for an state or an action. Figure 2.6 shows an exemplary value function, valuing each tile.

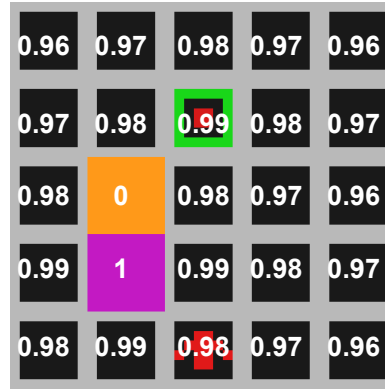


Figure 2.6: This example for a value function values the goal tile with a reward of 1 and each step with -0.01 .

There are common identifier for $Q(S, A)$ as a (state) action value function and $V(S)$ as a long term state value function of state S . These two functions can be connected as follows

$$\begin{aligned} V_\pi(S) &= \mathbb{E}[r_t | S_t = S, \pi] \\ Q_\pi(S, A) &= \mathbb{E}[r_t | S_t = S, A_t = A, \pi] \\ V_\pi(S) &= \sum_a \pi(a|s) \cdot Q_\pi(S, A) = \mathbb{E}[Q_\pi(S_t, A_t) | S_t = S, \pi] \quad , \forall s \end{aligned} \quad (2.7)$$

We can define optimal value functions $V^*(S)$ for the optimal state-value function and $Q^*(s, a)$ for the optimal state-action-value function

$$\begin{aligned} V^*(S) &= \max_\pi V_\pi(S) \\ Q^*(S, A) &= \max_\pi Q_\pi(S, A) \end{aligned} \quad (2.8)$$

An optimal value function solves any Markov decision problem since it specifies the best possible performance. Using a value function we can define a partial order for policies

$$\pi \geq \pi' \Leftrightarrow V_\pi(S) \geq V_{\pi'}(S) \quad , \forall s \quad (2.9)$$

Using this we can also define an optimal policy π^* with

$$\begin{aligned}\pi^* &\geq \pi \quad \forall \pi \\ V_{\pi^*}(S) &= V^*(S) \\ Q_{\pi^*}(S, A) &= Q^*(S, A) \quad .\end{aligned}\tag{2.10}$$

Bellman Equation

The value functions V for any policy π in an given Markov decision problem obey the Bellman expectation equation [2]

$$\begin{aligned}V_{\pi}(S) &= \sum_A \pi(S, A) \left[r(S, A) + \gamma \sum_{S'} p(S'|A, S) \cdot V_{\pi}(S') \right] \\ Q_{\pi}(S, A) &= r(S, A) + \gamma \sum_{S'} p(S'|A, S) \sum_{a' \in \mathcal{A}} \pi(a'|S') Q(S', a') \quad .\end{aligned}\tag{2.11}$$

Additionally they obey the Bellman optimally equations

$$\begin{aligned}V^*(S) &= \max_a \left[r(S, a) + \gamma \sum_{S'} p(S'|AS) \cdot V^*(S') \right] \\ Q^*(S, A) &= r(S, A) + \gamma \sum_{S'} p(S'|A, S) \max_{a' \in \mathcal{A}} Q^*(S', a') \quad .\end{aligned}\tag{2.12}$$

When transforming a reinforcement learning problem into a Markov decision process the Bellman equations can be used to solve it.

Taxonomy

Knowing this, we can now divide algorithms into three different categories. The first category is value based. Value based algorithms do not have an explicit policy, they use a value function as an implicit policy. Next are the policy based algorithms, these algorithms do not use a value function, They only work with a policy function. Actor-Critic methods use both of these, a policy as an actor for the action selection and an value function as critic for the optimization process.

2.2.3 Offline and Online Learning

This is not a very common distinction in reinforcement learning, but in this special case it is necessary to know the differences. Algorithms can be differentiated in online learning and offline learning, or a combination of these two.

Offline learning is far more common in the classical deep learning, it is batch learning. This means the algorithm gets a batch of data and starts the learning process on this data batch.

An online learning algorithm learns while the data generation process. The data is processed through the neural network when it is generated. This is a very common feature for reinforcement learning algorithms, due to the fact, that the data generation is always a part of reinforcement learning.

There are several algorithms that combine these two techniques, to use the data generation for the learning process and beyond that also learn on batches of the generated data.

The method of population based training, further explained in Section 2.3, restricts us to that we cannot use online learning. So the used algorithms are all offline learning or hybrid algorithms in which we exclusively use the offline learning.

2.2.4 Algorithms

There are several different algorithms for Reinforcement Learning, the algorithms we use here are based on this book [15]. In order to provide a brief overview, at least one algorithm for each of the three fields Value Based, Policy Optimisation and Actor-Critic is used in this work

Reinforce

The Reinforce method is a basic policy gradient algorithm. At first a new agent is initialized then the Reinforcement Loop starts.

1. Using the agent to play N episodes and saving (s_t, a, r, s_{t+1}) .
2. Next up for every step t in every episode k the discounted reward is calculated using

$$Q_{k,t} = \sum_{i=0} \gamma^i r_i \quad (2.13)$$

with the discount factor γ

3. The loss for all the saved transitions is calculated with

$$\mathcal{L} = - \sum_{k,t} Q_{k,t} \log(\pi(s_{k,t}, a_{k,t})) \quad (2.14)$$

4. At the end is an optimizer step to adjust the weights minimizing the loss.

These steps will be repeated. N is a previous set parameter and denotes the number of played episodes, the values s_t, a, r, s_{t+1} are the state, the performed action, the received reward and the next state.

Deep Q-Network (DQN)

Deep Q-Learning is the deep learning version of the table based Q-Learning [32], it is a value iteration algorithm.

In the initialisation step two models, a value model $Q(s, a)$ and a target model $\hat{Q}(s, a)$ as value functions, also set $\epsilon = 1.0$ for the epsilon greedy strategy. Next up is the training loop.

1. Determine action a using the epsilon greedy strategy, further explained in Section 2.1.6, and the target model $\hat{Q}(s, a)$.
2. Save the transition (s_t, a, r, s_{t+1}) into the memory.

3. Take a random sample mini-batch from memory and calculate the target

$$\begin{aligned} t &= r \quad \text{if } s \text{ was a terminal state} \\ y &= r + \gamma \cdot \max_i (\hat{Q}(s_{t+i}, a_{t+i})) \quad \text{else} \end{aligned} \quad (2.15)$$

again with the discount factor γ .

4. The next step is calculating the loss with

$$\mathcal{L} = (Q(s, a) - y)^2 \quad (2.16)$$

and making an optimisation step for $Q(s, a)$ to minimize the loss.

5. As final step every N steps the weights of the neural network Q will be copied to \hat{Q}

Proximal Policy Optimization (PPO)

As mentioned in Section 2.2.2 actor critic methods combine the advantages of Value Iteration and Policy Gradient methods. PPO is an Actor-Critic algorithm and PPO stands for Proximal Policy Optimization and was proposed by OpenAI [21] in 2017.

In Actor-Critic models the policy π acts, it is the actor, and chooses an action. The critic, the value function $V(s)$ is used to update the parameters.

Initialize a value network $V(s)$ and a policy π_θ copy this policy to $\pi_{\theta_{old}}$.

1. Generate data using the old policy $\pi_{\theta_{old}}$ for T steps in the environment.
2. Compute the advantage estimates $\hat{A}_1, \dots, \hat{A}_T$ using

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \dots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \quad (2.17)$$

3. Optimize the parameter using

$$\begin{aligned} r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \\ \mathcal{L}^{CLIP}(\theta) &= \hat{E}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right] \\ \mathcal{L}^{VF}(\theta) &= (V_\theta - V^{target})^2 \\ \mathcal{L}(\theta) &= \mathcal{L}^{CLIP}(\theta) - c_1 \mathcal{L}^{VF}(\theta) + c_2 S[\pi_\theta(s)] \end{aligned} \quad (2.18)$$

4. Every N steps replace the weight parameter θ_{old} with θ

Equation 2.18 uses:

- θ the policy parameter ,
- \hat{E}_t empirical expectation over time,
- r_t probability ratio of the new and old policy,
- advantage estimates \hat{A}_t ,

-
- ϵ an hyperparameter normally in between 0.1 and 0.2,
 - c_1 and c_2 are coefficients,
 - S is the entropy bonus.

2.3 Population Based Training

Population based training for deep learning is a rather new concept. It first was presented by DeepMind in 2017 [11]. But the idea of imitating the nature is not new to computer science, a better known example for this is simulated annealing [13], but there are quite a few more, for example evolutionary algorithms [33]. Population based learning is an evolutionary algorithm itself, again this is not the first time using evolutionary algorithms in deep learning [34].

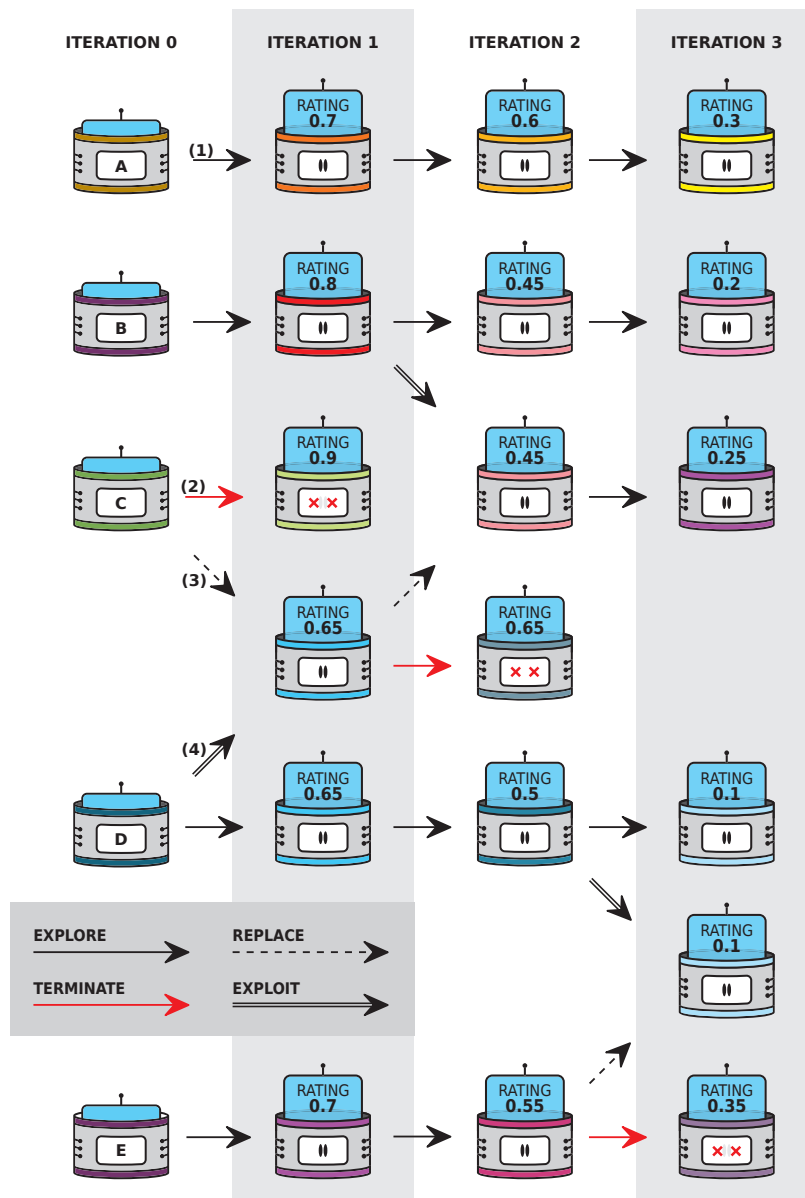


Figure 2.7: This graphic shows an abstract view of the population based training concept. An agent either mutates, what can be equated with exploring, or gets replaced by a better performing agent, this is an exploiting step. The goal is to minimize the rating, so the best agent replaces the worst [1].

The method of population based training was originally designed for automated hyperparameter optimisation. Models resulting from population based training were better and more robust.

Figure 2.7 shows the abstract concept of population based training. Instead of training a single model or agent in population based training a whole population of models or agents get trained, in this picture the five agents A through E. The agents train for several steps, in deep reinforcement learning the generated data can be shared and reduce the computing time this way significantly.

After one iteration of training all agents within the population get rated, in this and the following figures a lower value is better. The evaluated models then enter the mutation phase. At this stage, one of two things will happen to a model, either the hyperparameters will be mutated or it will be replaced. A red arrow (2) indicates a replacement process, the worst performing models in a population are replaced by copies of the best performing ones (4). This way the size of the population stays the same while getting rid of the worst performing agents. Arrow type (3) illustrates how the population continues after the agent has been replaced. The other agents mutate (1) in this step the hyperparameters of the agent will be modified, this step produces new hyperparameters each iteration. Population based training can also be seen as a kind of explore and exploit, mutating the hyperparameter is the exploring step and replacing the worst performing ones with the best performing one is exploiting this model.

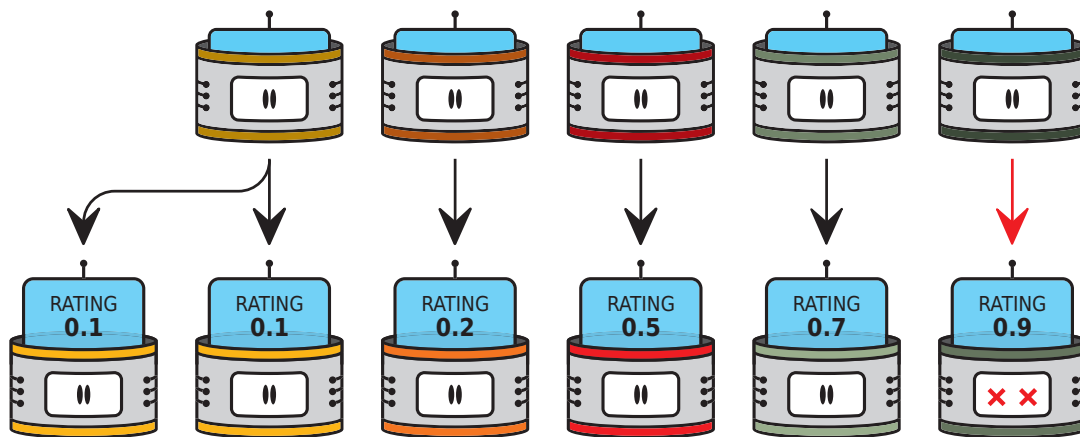


Figure 2.8: This figure shows one step of the population based training method with differently initialized agents. Each agent is created randomly, not only with an own set of hyperparameters, but also with a different model [1].

A population can be created in a number of ways, we want to limit ourselves to two of them here. Figure 2.8 pictures one way of doing this, each agent in the population is created independently from all the others. This way the population based training can profit from the different random initialisation and the overall better models will replace the worse models sooner or later. On the other hand this way the aspect of hyperparameter optimisation not the main focus, of course the better performing models have better fitting hyperparameters, but this way the better agents are the ones with right hyperparameters for their own model.

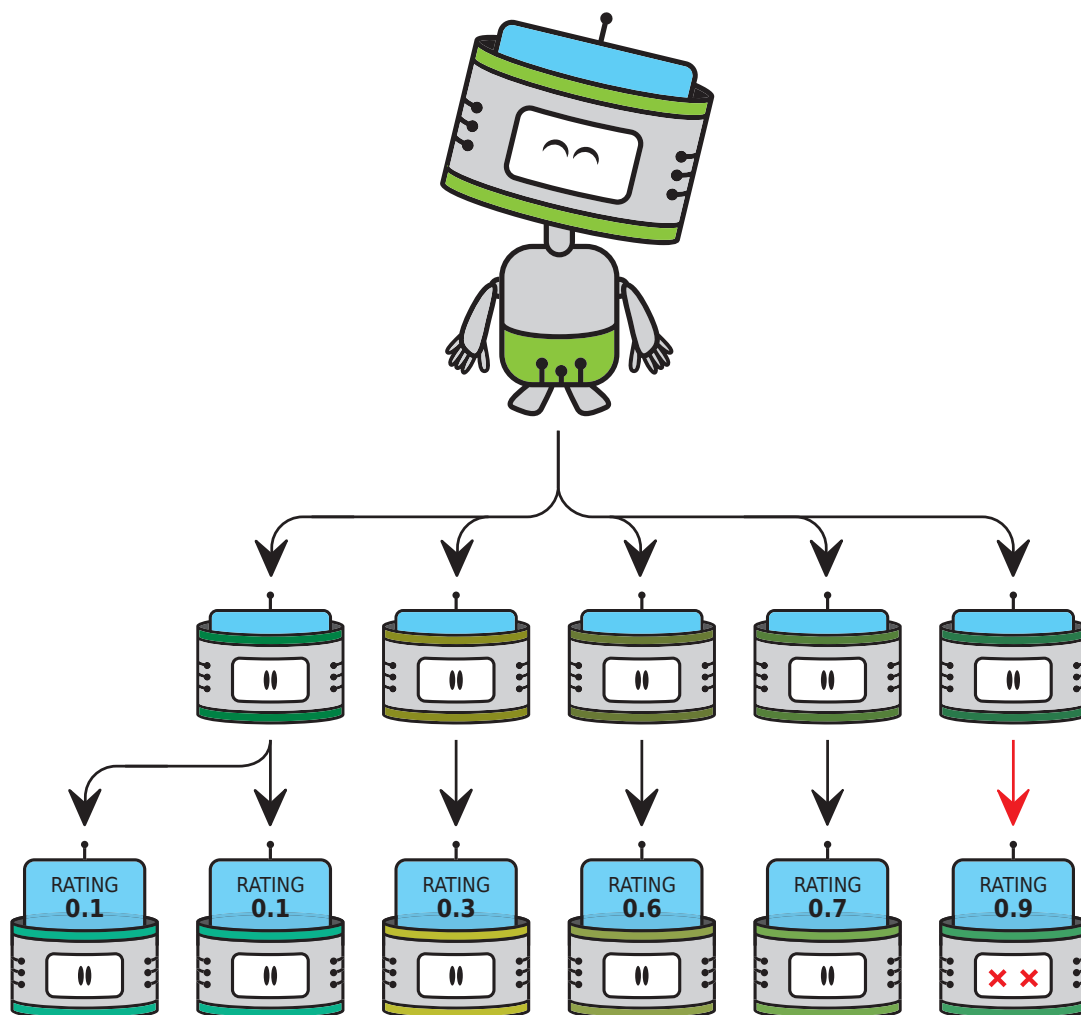


Figure 2.9: This is another initialization method for the population based training. In this case all agents have an own set of hyperparameters, but the models are all descents from one single model [1].

Figure 2.9 shows a different approach, here all agents are descendants from one model. This brings the optimization more into focus, since all these models are initialized with the same randomness the hyperparameters are more important to get a better performance.

2.4 Transfer Learning

Transfer learning is about applying learned knowledge to a different but related problem [29]. There are many different possible applications for this, for example image classification [8], in the medical field [7], and also in deep reinforcement learning [27].

There are several different ways to use transfer learning [26]. In this elaboration we will focus on retraining a whole model, there are possibilities of just retraining single layer. A common concept in the image processing is training on a big data set like image net and just retrain the output layer with a smaller specific data set. This means we have two different environments and train a new agent on the first one. After training is done, the environment is changed and training is carried out on the new environment. Both of these environments are similar reinforcement learning problems. The training on the second environment should deliver the desired performance much faster.


2.5 Related Work

The first work we take a look at is "Rethinking the usage of batch normalization and dropout in the training of deep neural networks" proposed by Guanyong Chen et al. [5] this work is about the usage batch normalization [9] and dropout [24]. This paper introduces a new Independent-Component (IC) layer that combines the techniques of dropout and batch normalization. Guanyong Chen et al. achieved a more stable learning process and faster convergence by placing these IC layer before the weight layer. A basic topic of this work is stable training, we decided to adapt this technique and place one dropout layer in front of each layer of the neural network instead of the standard way of placing one dropout layer at the start of the network. The work of Guanyong Chen et al. is not directly related to the topic of this work, population based training and transfer learning, but it is an interesting way to receive more stable models and therefore useful in this work about stable and robust reinforcement learning agents.

Max Jaderberg et al. [10] showed with "Human-level performance in first-person multiplayer games with population-based deep reinforcement learning" that deep reinforcement learning in combination with population based training in a multi-agent environment is not only capable of training one agent, but training agents in a way that they can work in a team. While every agent had to maximize the own reward, agents adapted human like strategies and worked together to achieve better results for all of them. The agents were capable of working together with different other players, other agents as well as humans.

In "A population-based learning algorithm which learns both architectures and weights of neural networks" Yong Liu, Xin Yao, et al. [17] showed the adaptability of population based algorithms for neural networks. In 1996 they proposed an population based algorithm, different to the one we use in this work, for population based training. This algorithm is not only capable to train a network, it also is used to create the architecture of the network. The concept of partial training and rating a population of neural networks is similar to the concept we use in this work, it also shows, that population based algorithms can do more than just train the neural network, it can design the neural network. This work is especially interesting for future works and automatic creation of neural networks.

The paper "Kickstarting Deep Reinforcement Learning" of Simon Schmitt et al. [20] is especially interesting for this work because it also combines the tow techniques of population based training and a kind of transfer learning. They use a so called 'teacher' agent to train a new 'student' agent. The 'teacher' agent is an already for this environment trained agent.



This idea of transfer learning is not about transferring knowledge from one environment to another, but transferring knowledge from one agent to another agent. Their experiments show, that kickstarting a reinforcement learning setup can help the new agent to achieve even better performance than the 'teacher'. This shows that the transfer of knowledge can be a key part in developing high performing reinforcement learning agents.

3 Concept and Idea

3.1 Gridworld

We work with a gridworld specifically created for this project, because we need some features that are not given in the most commonly used gridworld variants. The agent we want to train should not learn to memorize one specific way, the agent should learn to search the grid most efficiently. Our gridworld randomly generates a new map every reset. This way the agent not only has to learn to search to maximize the reward, this also helps to prevent overfitting. Additionally the agent learns on image data so the observation state will be rendered tile by tile and returned as a 3 channel RGB-image. This gridworld also supports different modes, explained in more detail in Section 3.1.3, this opens the possibility for transfer learning.

3.1.1 Tiles and States

In the gridworld we implemented has 8 different types of tiles. Each tile represented by an $8 \times 8 \times 3$ list for an 8×8 pixel RGB image. Figure 3.1 shows the rendered tiles.

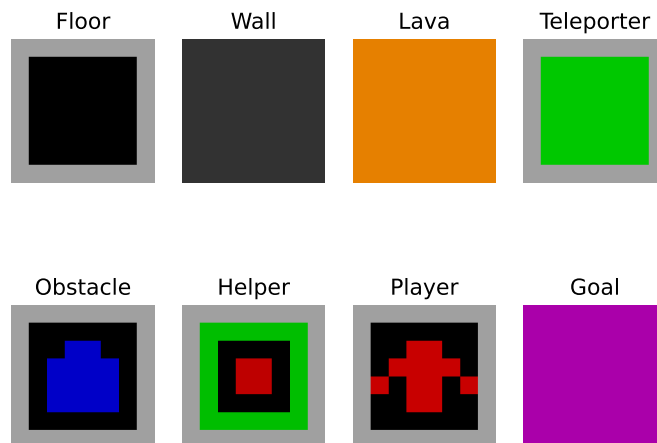


Figure 3.1: This are the 8×8 pixel representations for each tile. The most important ones are the player tile and the goal tile.

Player and the obstacles can just move in the forward direction, in these examples, as shown above, the direction would be up. The shape of these two tiles is important and these tiles are turned if needed, so the agent can learn to predict in which direction an obstacle can move.

Each step the obstacles and the player can either turn 90 degree left or right or make a step forward to the next tile. If the player wants to make a step forward onto a wall tile the player does not move, but the step is done. The obstacles make a random move and reduce the reward if they hit the player, the obstacle will be destroyed afterwards. The map contains two teleporter tiles if the player steps onto one of these tiles the player will be moved to the other, both tiles will be replaced by normal tiles. This also happens to the helper tile if the player steps on it.

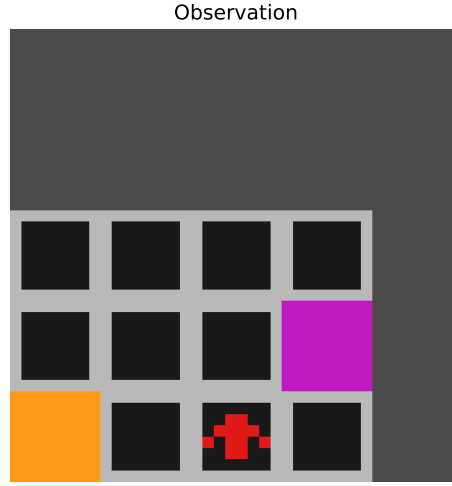


Figure 3.2: This is a fully 5×5 tiles rendered observation from the environment. The player always is in the bottom center.

The environment state S_t^e contains the full grid, surrounded by wall tiles. Additionally it contains the current reward penalty and the current step count t in this episode. Each action of the player influences the environment state, the step count is increased regardless of the action.

We work with a discrete action space of size 3 in the gridworld. One action is to move forward one tile and the other two are turning the player by 90 degrees to the left or to the right. There is no other trigger for the gridworld to change the environment state.

The observation state O_t is a 5×5 sector. It is rotated in a way that the player tile is always on the bottom middle tile as we can see in Figure 3.2.

3.1.2 Reward

For this gridworld the reward is always 0 except in the terminal state. If the reward is in a terminal state the reward is calculated as shown in Equation 3.1

$$R = 1 - \frac{t^{1.5}}{t_{max}^{1.5}} \quad . \quad (3.1)$$

Since the gridworld is completely random a perfect run for one instance can be more steps than a good run for another. This reward calculation takes into consideration that the optimal number of steps can vary and does not punish a few more steps too hard, but the more steps a client takes the lower is the reward. Figure 3.3 shows how the reward is scaled between 1 and 0.

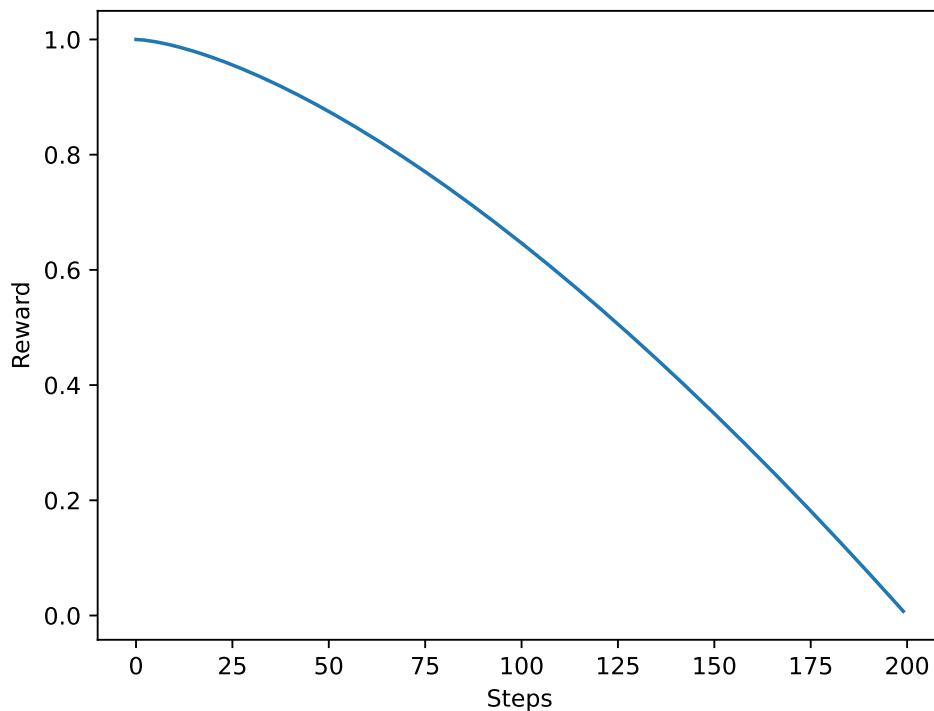


Figure 3.3: A plot for the reward function with 200 maximum steps. Each step t the possible reward is reduced. While the first steps do not matter a lot the later steps reduce the reward even more.

There are four types of terminal states in the environment. The first type is when the player steps on the goal tile, the reward then will most likely be strict positive. Second type is when the maximum number of steps is reached. And the third type is when the player steps on lava. Last the fourth type is when the reward is less or equal to zero, this is possible because there is a feature called *reward penalty*. Under certain conditions the gridworld has a feature called reward penalty, this lowers the possible reward. If the player makes a step into the wall the reward penalty will increase by 0.05, if the player hits an dangerous obstacle the reward penalty will increase by 0.2. This should teach the agent to not run into a wall and dodge the dangerous obstacles. The helper tile also works with the reward penalty value, but instead of increasing the penalty it decreases it by 0.1.

Listing 3.1: Calculating the reward of the gridworld.

```
def get_reward(self):
    reward = 1 - ((current_steps ** 1.5) / (max_steps ** 1.5))
    reward = reward - current_reward_penalties
    if reward < 0:
        self.done = True
        reward = 0
    return reward
```

Listing 3.1 is the complete function for the reward calculation. The concept of the reward penalty theoretically allows to get a negative reward, but the `if`-clause permits this. So if the player gathers up to much reward penalty instead of getting a negative reward the episode ends and the reward will be zero.

3.1.3 Modes

To enable transfer learning the gridworld has different modes, we use two of them. In both modes all objects are completely random placed on a 10×10 grid with a 5×5 observation range, also for both modes the maximum number of steps is 200.

The first mode is an empty grid without any dangerous obstacles or lava. The grid contains one helper and one teleporter pair.

The second mode is for the transfer learning, this mode contains all tiles the first mode contains, but also contains three dangerous obstacles and one tile of lava.

Listing 3.2: Making a Gridworld in both used modes.

```
gw = Gridworld.make("empty-10x10-random")
gw_hc = Gridworld.make("hardcore-10x10-random")
```

In Listing 3.2 we can see how to generate a gridworld. The usability of this gridworld is inspired by the gym framework from OpenAI [4]. Making one gridworld like this returns an object with the appropriate parameters. Figure 3.4 shows a full rendered map of the different modes, to be noted is that the slightly lighter area marks the observation.

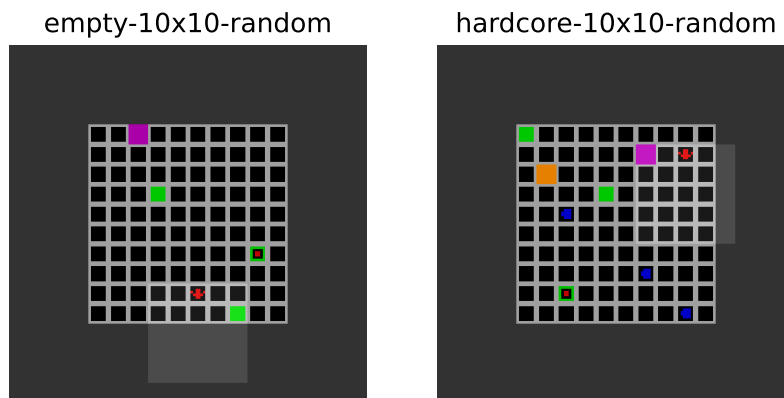


Figure 3.4: This are rendered maps of both gridworld modes. On the left is the empty version with just a helper, the goal and a teleporter. On the right is the hard version with additionally a lava tile and three obstacles. In both maps the current observation is slightly lighter than the rest.

Implementation can be found at <https://git.io/JMBRH>

3.2 Computational Setup

3.2.1 Neural Network Architecture

To enable the comparability of these algorithms we use all of them work with one basic setup for the neural network. Figure 3.5 is a simplified diagram of this network.

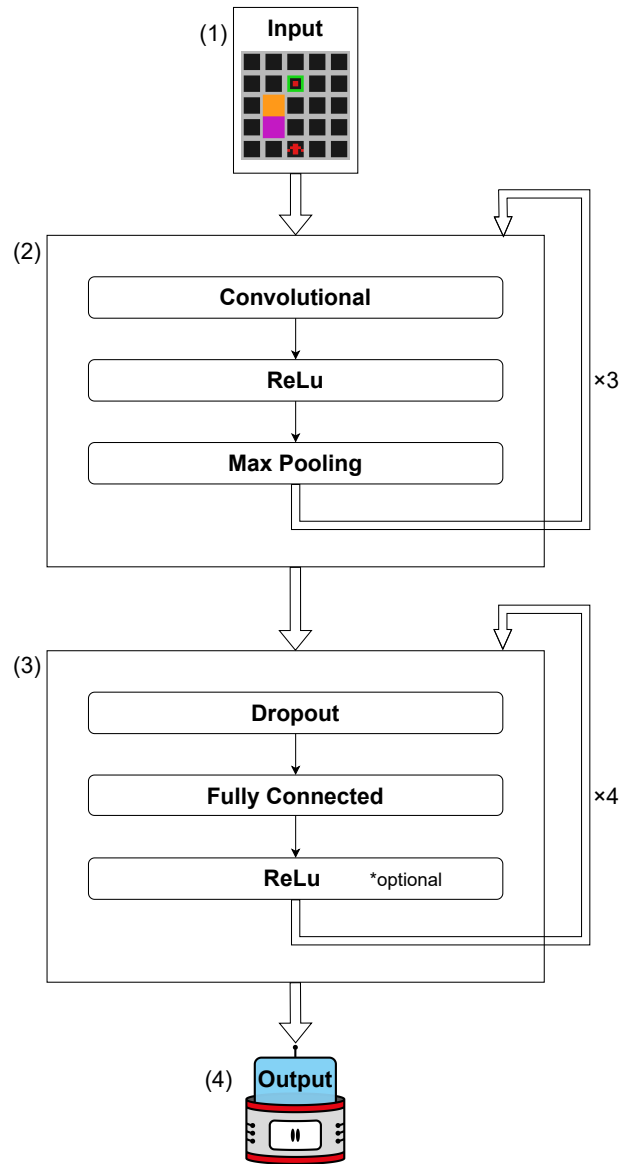


Figure 3.5: The neural network architecture. The input is one or more $3 \times 40 \times 40$ pixel image data. Next up the convolutional block is build up with thee layers each one convolutional layer followed up by a ReLu and a map pooling. The final block consists out of four fully connected layers, each in between a dropout and an activation function, mostly ReLu.

At first we start with the input (1). We know from Section 3.1.1 that each tile is a $3 \times 8 \times 8$ list and from Section 3.1.3 that the observation is 5×5 tiles. This makes each observation a $3 \times 40 \times 40$ RGB-image. The agent we build has three different types for the agent state S_t . The first and easiest mode is to just use the last observation frame as state, making this $3 \times 40 \times 40$ vector the input for the network. The other two types use stacked frames. While one uses three stacked frames as an $9 \times 40 \times 40$ input vector, the other uses three images as an $3 \times 3 \times 40 \times 40$ 3d input.

With the different input the convolutional layer (2) also differ. While the first two just differ in number of the input channels, for the 3d-stacked frames we also use 3d-convolutional layer. Because we work with raw image data we start with convolutional layer with an *ReLU* activation function and max pooling. The network has three of these convolutional layer with ascending output channel sizes of 32, 64 and 128 and a kernel size of 5, the 3d-convolutional layer have an $1 \times 5 \times 5$ kernel size.

Next up is the block of fully connected layers (3). Before each of the four fully connected layers the network has a dropout layer with 1% dropout, so we also have 4 of these dropout layers. This is due to the work of Chen, Guangyong, et al.[5] addressed in Section 2.5. After each layer, except the very last one, we again have an *ReLU* activation function. To note is that the PPO has an shared convolutional block (2) but two separate blocks of fully connected layers (3). The output sizes of the fully connected layer are descending 128, 64, 32 and the last depends on the algorithm. The critic part of the PPO have an output size of 1. The actor part, as well as the neural networks of the other algorithm have outputs (4) of the number of actions in this case 3. Furthermore they do not have an *ReLU* as an activation function. The DQN has a *tanh*, the Reinforce model and the critic part of the PPO does not have an additional activation function and the actor parts of both these algorithms use a *softmax* as activation.

3.2.2 Reinforcement Learning Setup

The setup for each algorithm is identical. We start out with the neural network itself which is almost identical with slight deviations for all algorithms, as shown in Section 3.2.1.

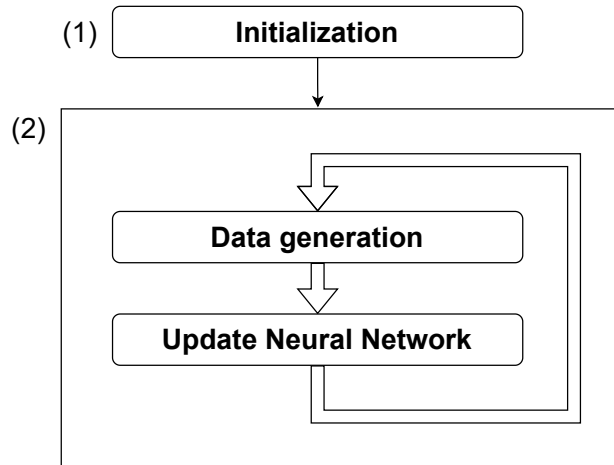


Figure 3.6: The general reinforcement training loop starts with data generation, the agent interacts with the environment. The following is the update step, in this step the loss is calculated and the parameters are updated.

Figure 3.6 is the basic training loop used in all algorithms. We start with the initialization step (1). In this step the environment and the neural network is made. The type of input is selected with the building of the neural network. The agent receives the neural network as well as the hyperparameters for the training and a memory.

Next up is the training loop (2), the loop starts with the data generation in this phase the agent plays the game, the states, the actions and the rewards are saved into the memory of the agent. The number of steps the agent plays depends on the algorithm.

Following the data generation step in the training loop is the update function. In this function the neural network is trained. The neural network is trained following the specific algorithm.

Each of the for used algorithms, explained in detail in Section 2.2.4, has its own Reinforcement Training loop.

Listing 3.3: The loop of the PPO Actor-Critic.

```
for episode in tqdm(range(parameter.environment_steps())):
    state, reward, done, info, memory, images = self.make_step(model,
        state, images, True, memory)
    if done:
        state, images = self.reset_env()
        done = False
    if not episode % parameter.episode_update() and episode >= parameter.
        episode_update():
        self.update(memory, model, optimizer, parameter)
        memory.clear()

    return agent
```

Apart from logging the mostly generic loop, exemplary shown in Listing 3.3 for the actor critic method, is adapted to the particular algorithm. All agents train with roughly 100,000 steps within the environment, but the way an agent determines the next action and when the next update is depends on the algorithm.

3.2.3 Container

We use several different container classes for uncomplicated management. One of the most important of these classes is the `MinMax` class. This class is a container for an integer or a floating point value, not only containing the actual value but also a minimum and a maximum value for this. The class assures that the value is always in between these limits. This class is mostly used for the hyperparameters, this is crucial for the population based training so the hyperparameters stay within a certain range and do not scale unlimited. There are several other container we use. One type of container is a specific container for hyperparameters for each of the used algorithms. These container include all hyperparameters for the algorithm, not only the ones that are modified later in the population based training.

There are two more relevant container classes, both of them are for the population based training. One of them contains all the parameter for the population based training. This container contains the size of the population, we always use a size of 10 in the experiments. It also contains the cut of the agents that are replaced in the population based training and the range of the allowed mutation.

The last container contains all information about an agent, the model, the statistics, the hyperparameter container and the history.

Additionally to the container classes the agent has its own memory class, this memory stores up the state, the action the reward and the information whether it is a terminal state, or not. This memory is necessary, since we need the algorithms to train offline.

3.3 Population Based Training

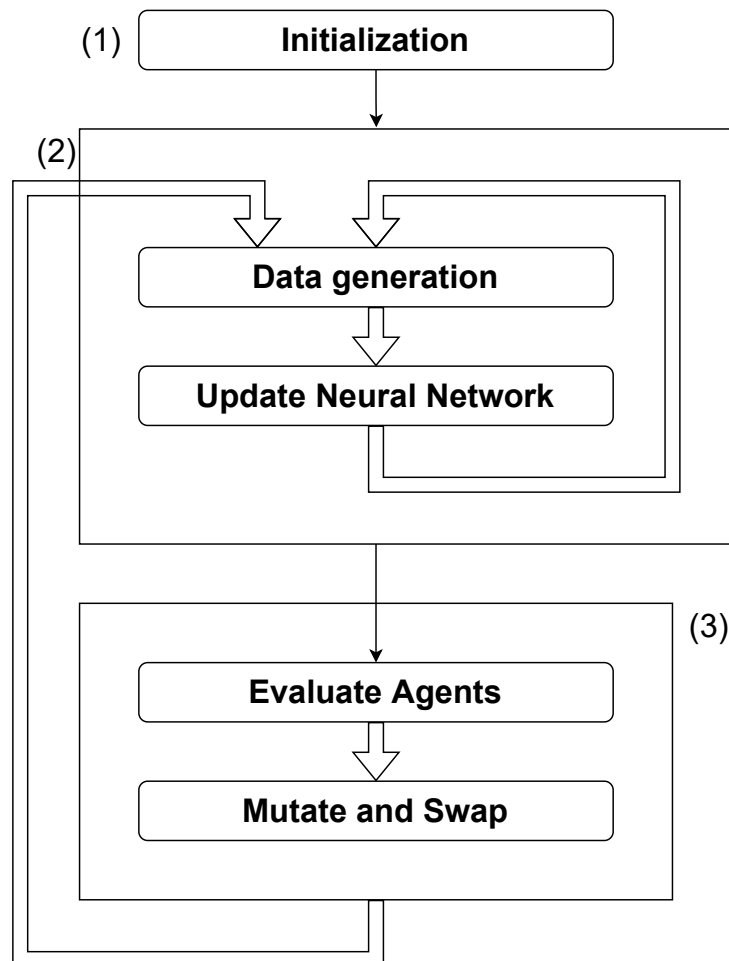


Figure 3.7: The population based training loop consist out of two main blocks and the initialization step. The first block is a slightly modified reinforcement training loop. The second block is the key part of population based training, mutating hyperparameters and swapping out bad performing agents.

The general algorithm for population based training, as we can see in Figure 3.7, does not differ a lot from the previous reinforcement training loop. The initialisation step (1) mostly corresponds to the initialisation step of the general reinforcement algorithm in the previous Section 3.2.2, but instead of creating one agent we have to create the whole population.

Each agent is created individually and not a copied version of an different agent, similar to the agents in Figure 2.8. While initializing each agent samples the own starting hyperparameters.

Since all algorithms we use are offline learning algorithms the data creation step of the general loop (2) completely equals the previous one. The data is created by just one of the agents, we always use the agent with currently the best rating to create the data. With this data the models of all agents will be updated according to the algorithm. As already mentioned we use a population with 10 agents.

Next we can see the population based training step (3), here is an additional branching from the training loop to the evaluation. The evaluation and mutation will not be triggered each iteration. In this case we evaluate the agents approximately every 20,000 steps in the environment.

Listing 3.4: The population based trainig loop for reinforce.

```
def train_population(self, population):
    episodes = 0
    done_training = False
    while not done_training:
        next_evaluation = self.eval_episodes(episodes)
        for eps in range(next_evaluation):
            memory = self.generate_memory(population[0].model,
                                          population[0].hyper_container[-1])
            for t in population:
                t_memory = copy.copy(memory)
                t_memory.compute_returns(
                    t.hyper_container[-1].discount())
                self.update(t_memory, t.model, t.optimizer,
                           t.hyper_container[-1])
                t_memory.reset()
            memory.reset()

            if episodes == self.pbt_container.evaluation_steps:
                done_training = True
                episodes += 1

        self.evaluate_all(population)
        self.mutate_hyper(population, episodes)

    return population
```

Listing 3.4 is the population based training loop exemplary for the reinforce algorithm. It should be noted that `population[0]` always contains the currently best performing agent. The middle `for`-loop is the data generation and model update loop and the outer `while`-loop is the population based training loop, in this loop we start with evaluating the population. Evaluating the agents current performance is a key step for the population based training. Each agent plays 10 complete games and will be evaluated. The exact rating we use is explained in the following Subsection 3.3.1. After the evaluation step is the mutation step. In the mutation step the 20% worst performing agents, in this case the two agents with the highest rating, will be removed and replaced by copies of the best performing agent. The hyperparameters for all other agents will be modified. In Subsection 3.3.2 we go into more detail about the mutation process.

3.3.1 Rating

To evaluate and rate the population all models play 10 episodes and save the metrics. To get the best rating we take several values into account. The accuracy is the percentage of the agent finding the goal and successfully winning an episode. Additionally the average number of steps, the average reward and the average reward penalty are counted in. Using the reward and the reward penalty weights the fact that an agent efficiently dodges the obstacles and does not run into walls higher. All the metrics we use are from the current evaluation episodes and not the all time performance since the performance of a model can drastically change each evaluation.

Preservative Rating

For the effectiveness of transfer learning the current performance of a model should not always be crucial. Due to the nature of population based training the performance of one agent can vary each iteration a lot. In Transfer Learning the models are trained to be trained again, so the future performance is critical. This makes a consistent model more valuable than an inconsistent one with slightly better metrics. To consider the consistency of an agent we also use the previous Rating.

Listing 3.5: The rating function.

```
accuracy_value = 1 - (success / episodes)
step_value = average_steps / 200
reward_value = 1 - average_reward
summed_values = accuracy_value + step_value + reward_value +
    average_reward_penalty
new_rating = summed_values / (4 + history * (1 - rating[-1]))
```

Listing 3.5 shows exactly how the rating is calculated, the goal is to minimize the rating. The `history` is a value for each model starting at 0, each iteration a model is replaced this value will be reset to 0 again, every other time this value will be increased, in this case by 0.05. The previous rating will be multiplied by this value, this way the influence of the rating of consistently good performing agents increases.

3.3.2 Mutation

In the mutation step differentiates the agents. The agents with the worst rating will be removed and replaced by copies of the best performing agent. Not only the neural network will be replaced, the hyperparameters will also be replaced. This way the currently best agent will be obtained.

For all other agents, including the best one, the hyperparameters will be modified.

Listing 3.6: The mutation step.

```
for i in reversed(range(len(population))):
    if i >= (len(population) - len(population) * self.pbt_container.
        cut):
        population[i].history = 0
        population[i].hyper_container.append(copy.copy(population[0].
            hyper_container[-1]))
        population[i].model.load_state_dict(population[0].model.
            state_dict())
        population[i].parent.append(population[0].id)
    else:
        population[i].history += self.pbt_container.
            history_rating_cut
        population[i].hyper_container.append(copy.copy(population[i].
            hyper_container[-1]))
        population[i].hyper_container.append(
            self.mutator(step, population[i].hyper_container[-1]))
```

Listing 3.6 is the mutation method, the parameter `population` is an already rated and sorted list containing the agents, `population[0]` is the currently best rated agent. The `mutator` modifies the existing hyperparameters, the modification is done by sampling a random value in the range of $1 - \text{mutation_cut}$ and $1 + \text{mutation_cut}$ and multiplying this with the old parameter, if this value is within the permitted range of the hyperparameter this will be the next value. The `mutation_cut` value is the permissible deviation each iteration. In this case we use a `mutation_cut` of 0.2, this means we allow an adjustment of 20% per iteration.

3.4 Transfer Learning

Due to the fact that the implementation of the algorithms was kept very generic and was not specially adapted to the environment, the transfer learning was not complex. As already mentioned in Section 2.4 we retrain the already trained model on the new environment.

Listing 3.7: The Transfer Learning for a single agent.

```
agent = self.make_model_container(0)
self.env = Gridworld.make("empty-10x10-random")
transfer_agent = self.train_single(agent)
self.env = Gridworld.make("hardcore-10x10-random")
transfer_agent.rating = [0.0]
transfer_agent.history = 0
transfer_agent = self.train_single(transfer_agent)
```

As we can see in Listing 3.7 we train a model on the empty gridworld and after resetting the statistics and the history we retrain the model on the more complex environment, the gridworld in the hardcore mode. There were several options for the population based transfer learning approach.

The first decision to be made is whether to only initially train with Population Based Training and just retrain the best performing model or not, but since especially the Transfer Learning should profit from Population Based Training it was obvious to continue Population Based Training for the Transfer Learning. Another occurrence, further treated in Section 4.2, supports this decision.

Next, the same question arises as with population based training, characterized in Section 2.3, whether all agents descended from one model or not. In the first case this one model would obviously be the best performing, in the other the whole trained population will be retrained.

Listing 3.8: Population Based Transfer Learning.

```
population = self.init_population_training()
self.evaluate_all(population)
self.env = Gridworld.make("hardcore-10x10-random")
transfer = []:
for i in range(self.pbt_container.population_size):
    if best_agent:
        transfer.append(copy.copy(population[0]))
    else:
        transfer.append(copy.copy(population[i]))
        transfer[i].optimizer = self.fit_optimizer(transfer[i])
        transfer[i].rating = [0.0]
        transfer[i].history = 0
        transfer[i].parent = [i]
self.train_population(transfer)
```

Listing 3.8 shows the population based transfer learning. The parameter `best_pop` indicates whether the whole population is retrained or a population of descendent of the best performing model, it should be mentioned again that the models are sorted in the `evaluate_all` method and `population[0]` contains the best performing model.

4 Evaluation

To evaluate the different algorithms we added a baseline of single training agents. The baseline agent uses manually optimized hyperparameters. Each agent in the population uses randomly selected hyperparameters. Each population is trained with a population size of 10. Due to the fact that only the best performing agent is involved in the data generation process the the agents within the population only create logging data while the evaluation step of the population based training. This causes the baseline data to be more fine-grained than the logging data from the population based training.

4.1 Reinforce

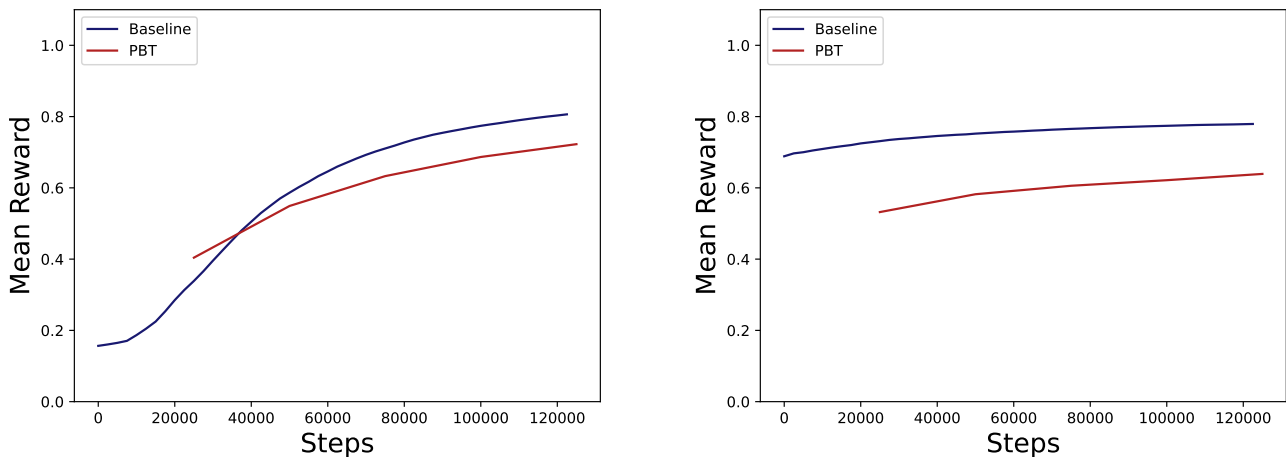


Figure 4.1: The mean reward of the initialisation training (left) and the transfer training (right) using the Reinforce algorithm.

In Figure 4.1 we can see that the average values in the population based training do not surpass the values of the baseline. But looking at this we have to keep two things in mind, the process to manually optimize the hyperparameters of the baseline took a long time while the hyperparameters of the agents is randomly sampled. Additionally the population based training is an exploit and explore process, most of the agents within a population are exploring new hyperparameter combinations. So if we just look at the best performing agents and compare them to the best agents in the baseline the Figure 4.2 looks quite a bit different. For comparability we compared the 20% best performing agents per population with the overall 20% best performing baselines.

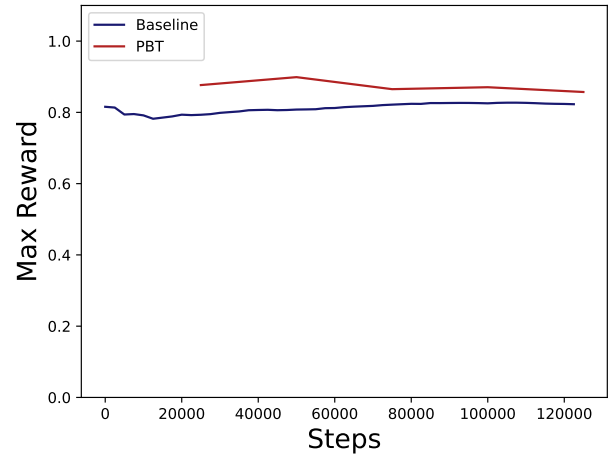
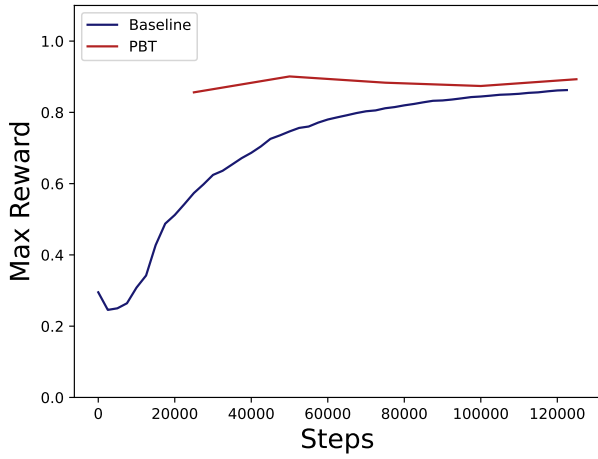


Figure 4.2: The best reward of the initialisation training (left) and the transfer training (right) for the Reinforce algorithm.

When comparing the best agents the population based training performed overall better, without any hyperparameter search. Additional graphs can be found in Section 7.1.

4.2 Deep Q-Learning

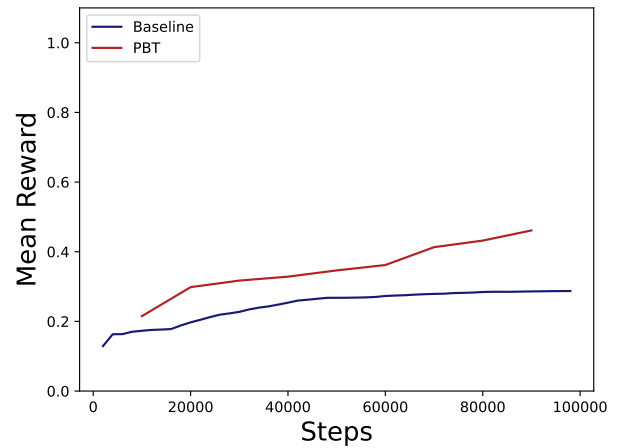
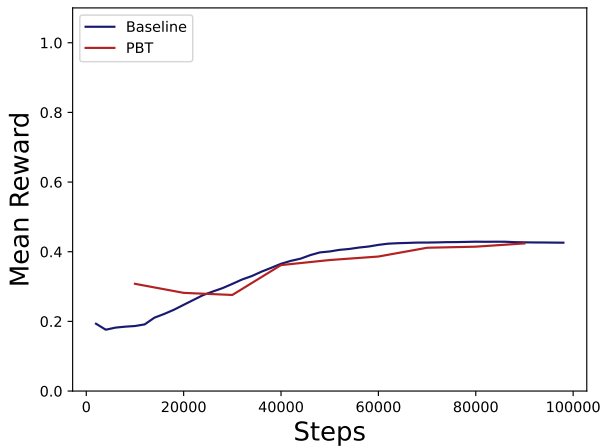


Figure 4.3: The mean reward of the initialisation training (left) and the transfer training (right) using the DQN algorithm.

The DQN had a lot of not good performing agents, an observation which is examined in more detail in Section 4.5. The performance from both, the baseline and the population based training is rather mediocre for the DQN as we can see in Figure 4.3.

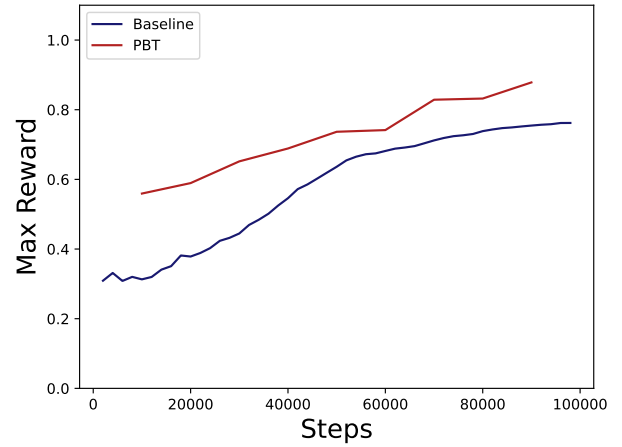
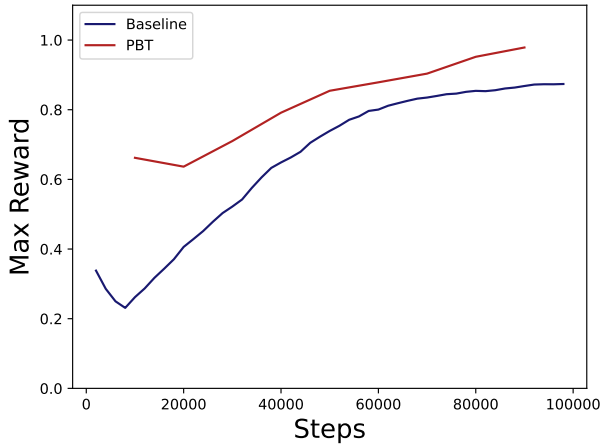


Figure 4.4: The best reward of the initialisation training (left) and the transfer training (right) for the DQN algorithm.

Again the performance of the best performing ones is acceptable. The population based training, again without any hyperparameter search, is slightly ahead in performance as we can see in Figure 4.4. More figures in Section 7.2

4.3 Proximal Policy Optimization

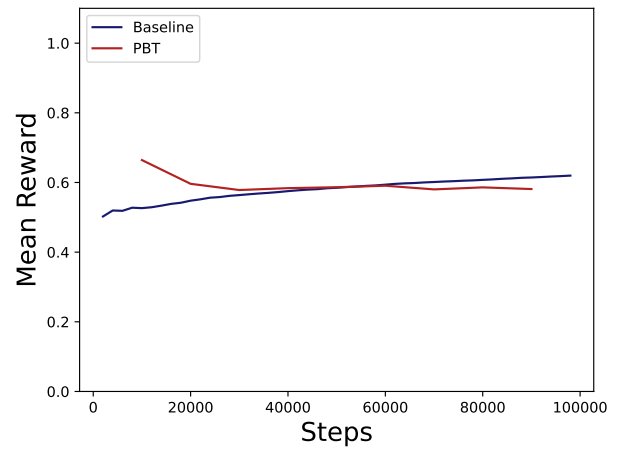
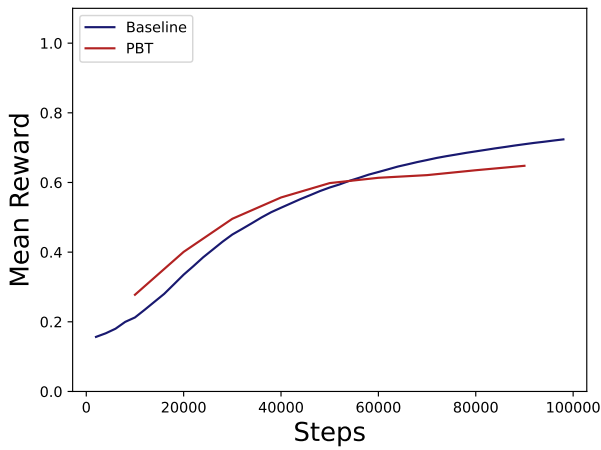


Figure 4.5: The mean reward of the initialisation training (left) and the transfer training (right) using the PPO algorithm.

The PPO algorithm was quite closer as we can see in Figure 4.5. When we look at the best 20%, in Figure 4.6 this time the baseline is slightly better than the population based training, but on the right side, in the transfer learning, the population based training has a slightly better performance.

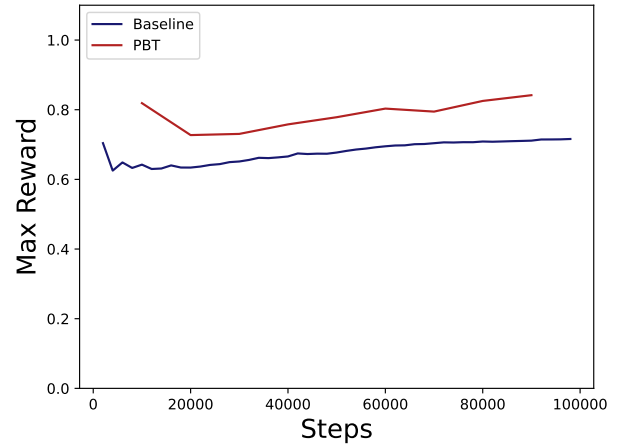
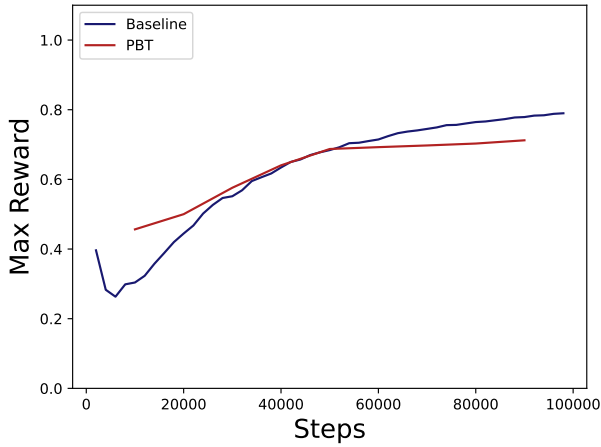


Figure 4.6: The best reward of the initialisation training (left) and the transfer training (right) for the PPO algorithm.

4.4 Preservative Rating

In Section 3.3.1 we described the the rating function used in the population based training and we introduced the preservative rating concept. As a reminder, this means that the previous rating will be included in the new one.

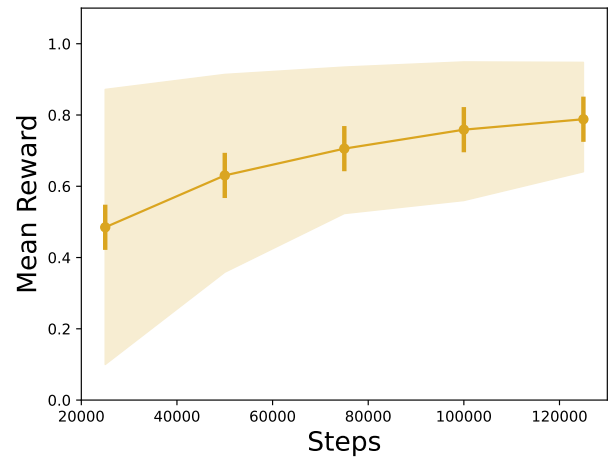
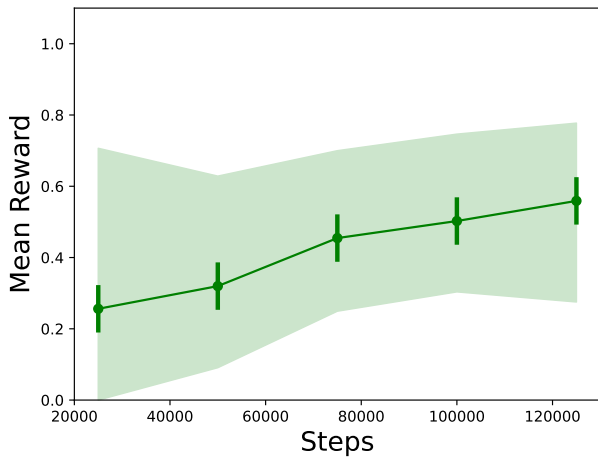


Figure 4.7: The mean reward without (left) and with (right) preservative rating. The line represents the average value, the bars are the variance and the lighter area are the minimum and maximum values.

We'll be taking a look at two different metrics. All populations trained with a population size of 10 and were created under otherwise identical conditions. The only change is that the history value does not increase and permits using the previous score this way. At first the *mean reward*, this is for one agent the averaged reward throughout the whole training.

On the left side of Figure 4.7 we can see the mean reward of several populations trained without using preservative rating. The line is the average value and the lighter area denotes the minimal and the maximal value, this also applies to the following figures. On the right side we can see the associated graph with preservative rating.

It is noticeable, although the average values are not significantly better, that the maximum and minimum value are significantly closer together at the end of the training.

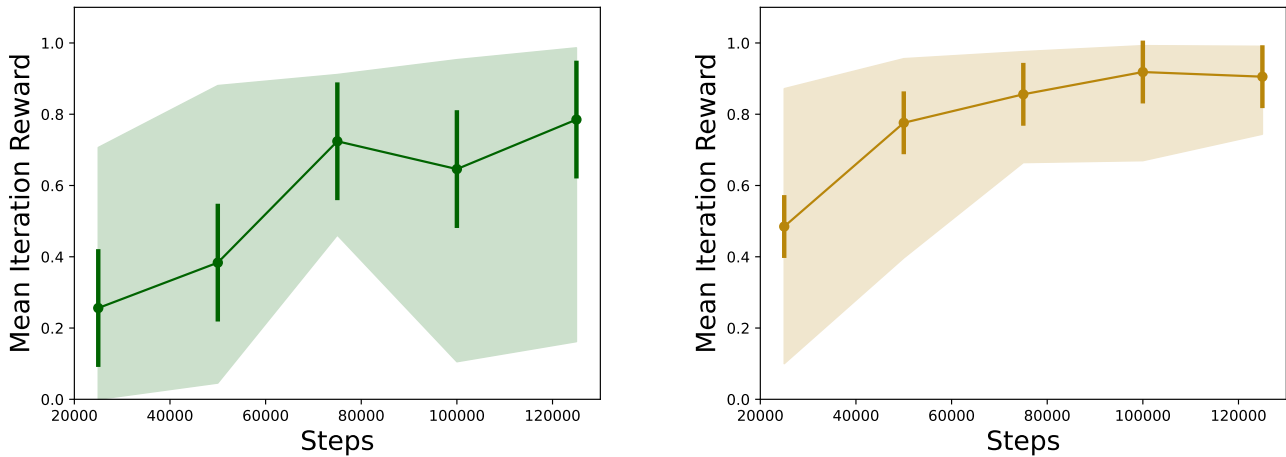


Figure 4.8: The mean reward per iteration without (left) and with (right) Preservative Rating. The line represents the average value, the bars are the variance and the lighter area are the minimum and maximum values.

To stick with this metric we look in Figure 4.8 at the mean reward, but this time only for the last iteration. The observation from just now repeats itself, the range of the extreme points is again significantly smaller towards the end of the training.

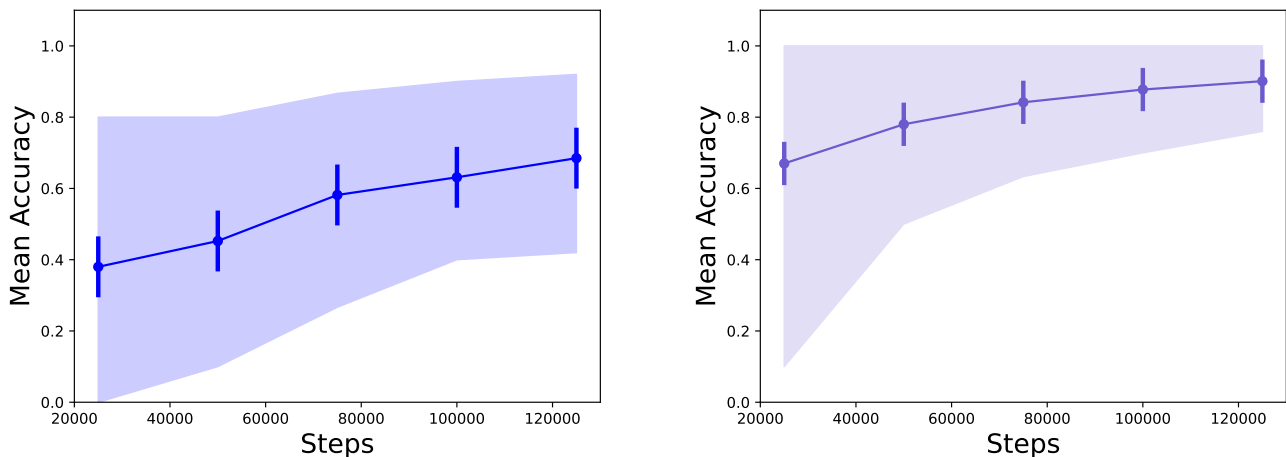


Figure 4.9: The mean accuracy without (left) and with (right) preservative rating. The line represents the average value, the bars are the variance and the lighter area are the minimum and maximum values.

The next metric we take a look at is the *accuracy*, the accuracy for one agent is the number of won games divided by the number of played games, it is essentially the win rate of the agent. In Figure 4.9, showing the mean accuracy throughout the whole training, the observed trend continues. Additionally we can see that the average accuracy is noticeably better. The accuracy per iteration, shown in Figure 4.10, shows this very clearly.

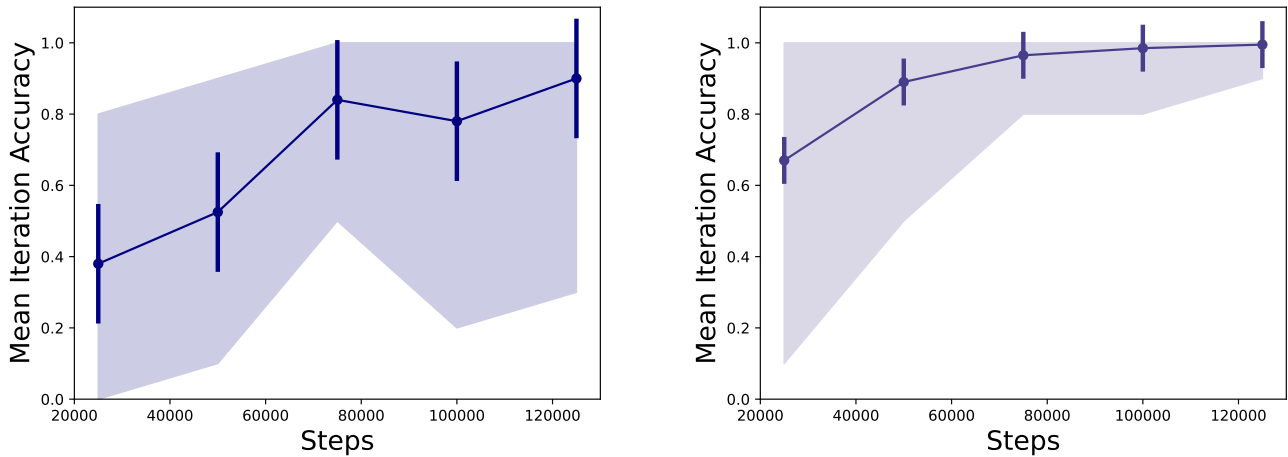


Figure 4.10: The mean accuracy per iteration without (left) and with (right) preservative rating. The line represents the average value, the bars are the variance and the lighter area are the minimum and maximum values.

To explain this observation we have to take a look into the data, Figure 4.11 shows a selected part of the raw data. We see, that the performance of a single agent varies a lot during the training. This is due to the mutation step of the population based training.

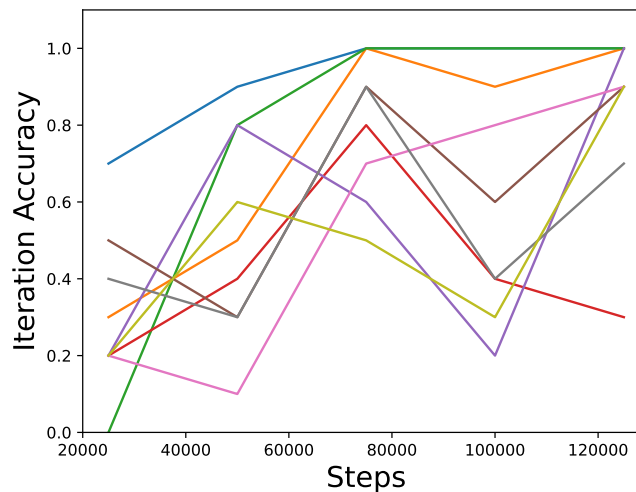


Figure 4.11: The accuracy per iteration without preservative rating. Each line represents the performance of a single agent throughout the population based training.

Despite the best agents performing on a very high level, there are several agents with strongly fluctuating performance. These unstable agents drag the overall performance of a population noticeably downwards. Preservative rating rewards stable behavior, this way the overall performance is increased.

4.5 Transfer Population

As already mentioned in the Section 3.4, there are different methods by which the population can be initialized. The first method is retraining the whole population and the second is to create the entire population as a descendant of the best agent.

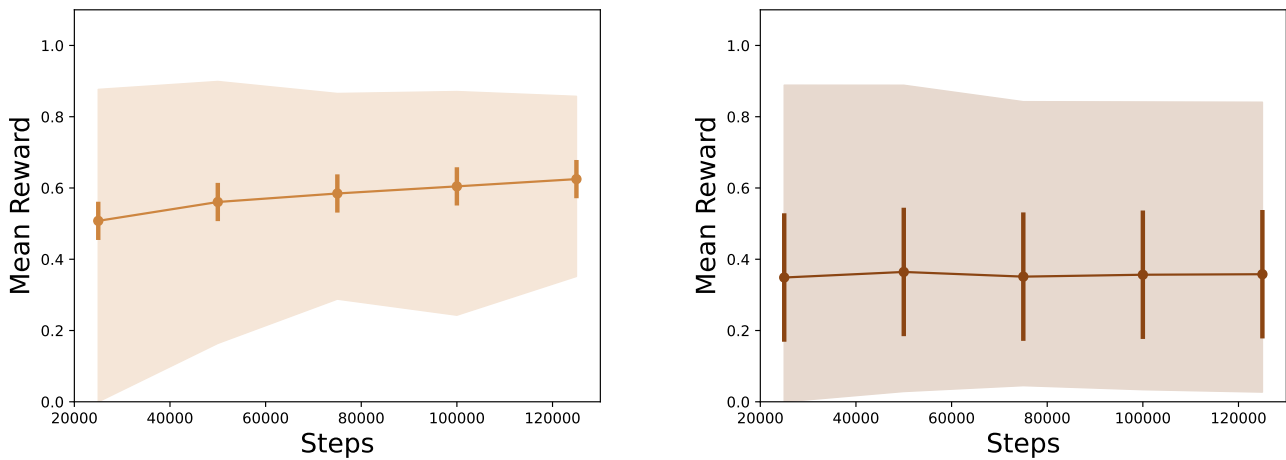


Figure 4.12: The mean reward with the different population initialization methods. On the left using the whole population and on the right using the best agent. The line represents the average value, the bars show the variance and the lighter area is the minimum and maximum span.

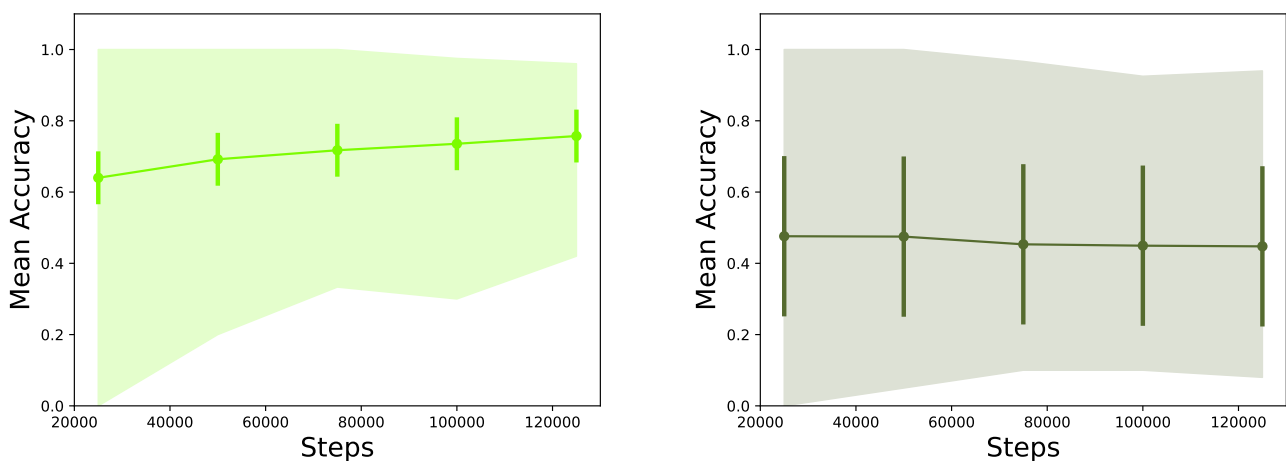


Figure 4.13: The mean accuracy with the different population initialization methods. On the left using the whole population and on the right using the best agent. The line represents the average value, the bars show the variance and the lighter area is the minimum and maximum span.

The Figures 4.12 and 4.13 show the mean rewards and the mean accuracies of the two Methods. On the left the whole retrained population and on the right a population based on the best performing agent in the easy environment. This data is combined from several different populations while training on the more difficult environment. As before the line represents the average value and the slightly lighter area shows minimum and maximum values.

As we can see, the average on the left is better in both metrics, and it actually improves slightly. We can also see the minimum value improve visibly.

Initializing a population with just one agent does not seem to work as good. It indicates no noticeable improvement and the average is no better than mediocre.

One closer look into the data reveals the problem.

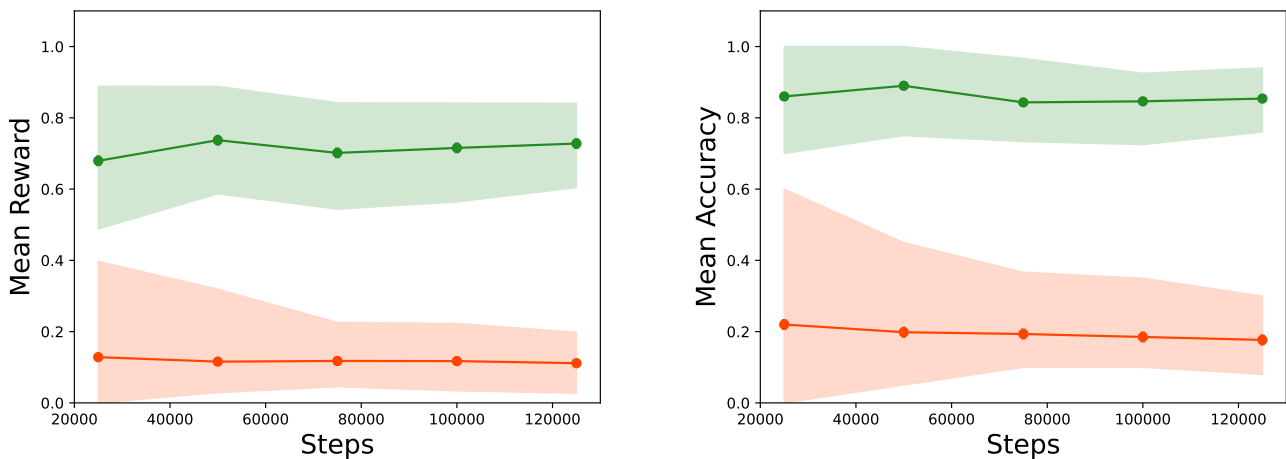


Figure 4.14: The mean reward (left) and the mean accuracy (right) using the best agent as initialization for the transfer learning. The line represents the average value, the small bars show the variance and the lighter area is the minimum and maximum span. The populations are split up in good performing ones (green) and bad performing ones (orange). A whole population is always in one of these and never split into both.

In Figure 4.14 the populations within the data are split into the good performing ones in green and the bad performing ones in orange. This is a problem we not only saw in the transfer learning, but in reinforcement learning itself, some agents just don't perform. This happens in the easy environment and in the harder environment.

It should be noted that the population comes from the best performing agent in the simpler environment. These agents performed very well on the empty environment, but they could not adapt well to the new environment. Because of this circumstance, entire populations initialized with this method can be pointless after the Transfer Learning.

More metrics for comparison are shown in the appendix in Section 7.4.

5 Discussion

When we look at the results of the population based training in Chapter 4 they seem very promising. The simple and non-optimized population based training was able to keep up with the baseline and in some cases even slightly surpass it. But we can also see that population based training cannot work miracles, if an algorithm is unstable for an environment, as the DQN for our gridworld, the results of the population based training can also be unstable. This unstable behavior can take on extreme behavior as we see when we look at the last part of the Section 4.5. Depending on the initialization the population based training can run without resulting in one useful agent, but especially in this case, the case of an algorithm training unstable in the environment, population based training can speed up the training. Instead of training several single agents and hoping one results in a good performing model, training a whole population with just shared data is more likely to result in a working agent.

The initialization of the population is very important, for unstable environments we recommend using different agents and benefit both in terms of stability from the diversity of the agents and in terms of computation time from the joint generation of data. This also applies to the generation of the population in transfer learning. We see that the population based training can bring robustness and stability to the learning process. One of the opening questions was whether the combination transfer learning and population based training, will result in the expected increase in stability and robustness. We can definitely say that population based training for initial learning increases stability.

Looking at the results of the DQN, in Section 4.2, we can affirm this question. The DQN was very unstable in our experiments, and every agent, whether within a population or as a baseline, that did not perform in the initialisation also did not perform in the transfer learning. The population based training helps to eliminate not performing agents in the initial learning phase.

Let us now take on the second question, "does the stability of population based training justify the increased complexity for transfer learning?"

There are two complexities to there are two types of complexity one can consider, implementation complexity and time complexity. There are a few small things to consider when implementing population based training. Especially the rating function is important. The perservative rating achieved a surprising performance, as we can see in Section 4.4. Using this to rate the population increased the stability noticeable.

Looking at the time complexity we can say of cause population based training is slower than just training a single agent, but it is faster than training the same number of agents.

Not only the increased stability does justify the increased complexity, but also the automated hyperparameter optimization does justify it. The population based training used randomly selected hyperparameters without spending any time on optimizing them.

In this work, population based training performed better than expected, but there is still a lot that can be tried out.

In this elaboration we worked with three algorithms, two of them very basic, the combination of population based training with transfer learning on more complex algorithms can be worth looking at. Also the selection of hyperparameters can be improved, depending on the algorithm several different hyperparameters can be selected, this work mainly focused on a limited set of them. In conclusion, it can be said that the combination of population based training and transfer learning has great potential, especially with perservative rating, and should be explored further.

6 Conclusion

Population based training can be very effective. A big advantage is the automated hyperparameter tuning. In this elaboration we showed that population based training can increase the performance of reinforcement learning also in combination with transfer learning. Population based training can be self correcting, but we also showed that population based training itself needs to be set up right for the environment.

The choice of the rating function for population based training can influence the results of a population based training a lot. Also the initialization of the population should be adapted to the problem and the environment. Training with a population based on a single model in a very error-prone environment can backfire and result in not learning agents.

Bibliography

- [1] Jasmin Ahmadi. *Design and drawing of several graphics in this elaboration*. 2021.
- [2] Richard Bellman. “Dynamic programming”. In: *Science* 153.3731 (1966), pp. 34–37.
- [3] Christopher Berner et al. “Dota 2 with large scale deep reinforcement learning”. In: *arXiv preprint arXiv:1912.06680* (2019).
- [4] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).
- [5] Guangyong Chen et al. “Rethinking the usage of batch normalization and dropout in the training of deep neural networks”. In: *arXiv preprint arXiv:1905.05928* (2019).
- [6] Yue Deng et al. “Deep direct reinforcement learning for financial signal representation and trading”. In: *IEEE transactions on neural networks and learning systems* 28.3 (2016), pp. 653–664.
- [7] Khalid M Hosny, Mohamed A Kassem, and Mohamed M Foad. “Skin cancer classification using deep learning and transfer learning”. In: *2018 9th Cairo international biomedical engineering conference (CIBEC)*. IEEE. 2018, pp. 90–93.
- [8] Minyoung Huh, Pulkit Agrawal, and Alexei A Efros. “What makes ImageNet good for transfer learning?” In: *arXiv preprint arXiv:1608.08614* (2016).
- [9] Sergey Ioffe. “Batch renormalization: Towards reducing minibatch dependence in batch-normalized models”. In: *arXiv preprint arXiv:1702.03275* (2017).
- [10] Max Jaderberg et al. “Human-level performance in 3D multiplayer games with population-based reinforcement learning”. In: *Science* 364.6443 (2019), pp. 859–865.
- [11] Max Jaderberg et al. “Population based training of neural networks”. In: *arXiv preprint arXiv:1711.09846* (2017).
- [12] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.
- [13] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. “Optimization by simulated annealing”. In: *science* 220.4598 (1983), pp. 671–680.
- [14] Marc Lanctot et al. “A unified game-theoretic approach to multiagent reinforcement learning”. In: *arXiv preprint arXiv:1711.00832* (2017).
- [15] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.
- [16] Yuxi Li. “Deep reinforcement learning: An overview”. In: *arXiv preprint arXiv:1701.07274* (2017).
- [17] Yong Liu, Xin Yao, et al. “A population-based learning algorithm which learns both architectures and weights of neural networks”. In: *Chinese Journal of Advanced Software Research* 3 (1996), pp. 54–65.
- [18] Riccardo Miotto et al. “Deep learning for healthcare: review, opportunities and challenges”. In: *Briefings in bioinformatics* 19.6 (2018), pp. 1236–1246.

-
- [19] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).
- [20] Simon Schmitt et al. "Kickstarting deep reinforcement learning". In: *arXiv preprint arXiv:1803.03835* (2018).
- [21] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).
- [22] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.
- [23] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.
- [24] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.
- [25] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [26] Chuanqi Tan et al. "A survey on deep transfer learning". In: *International conference on artificial neural networks*. Springer, 2018, pp. 270–279.
- [27] Matthew E Taylor and Peter Stone. "Transfer learning for reinforcement learning domains: A survey." In: *Journal of Machine Learning Research* 10.7 (2009).
- [28] Gerald Tesauro. "TD-Gammon, a self-teaching backgammon program, achieves master-level play". In: *Neural computation* 6.2 (1994), pp. 215–219.
- [29] Lisa Torrey and Jude Shavlik. "Transfer learning". In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264.
- [30] Martijn Van Otterlo and Marco Wiering. "Reinforcement learning and markov decision processes". In: *Reinforcement learning*. Springer, 2012, pp. 3–42.
- [31] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354.
- [32] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.
- [33] Bing Xue et al. "A survey on evolutionary computation approaches to feature selection". In: *IEEE Transactions on Evolutionary Computation* 20.4 (2015), pp. 606–626.
- [34] Steven R Young et al. "Optimizing deep learning hyper-parameters through an evolutionary algorithm". In: *Proceedings of the workshop on machine learning in high-performance computing environments*. 2015, pp. 1–5.

7 Appendix

7.1 Reinforce

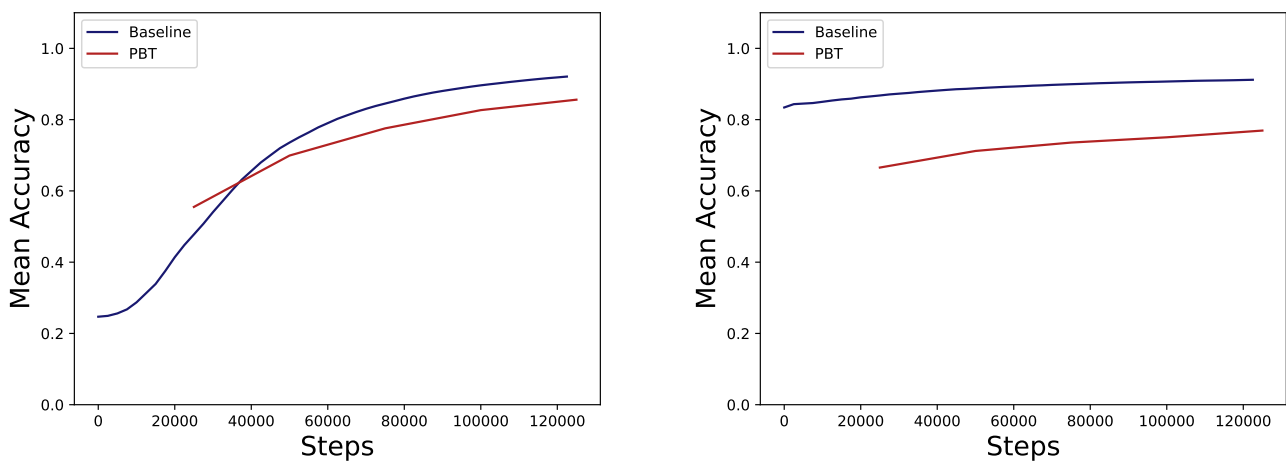


Figure 7.1: The mean accuracy of the initialisation training (left) and the transfer training (right) using the Reinforce algorithm.

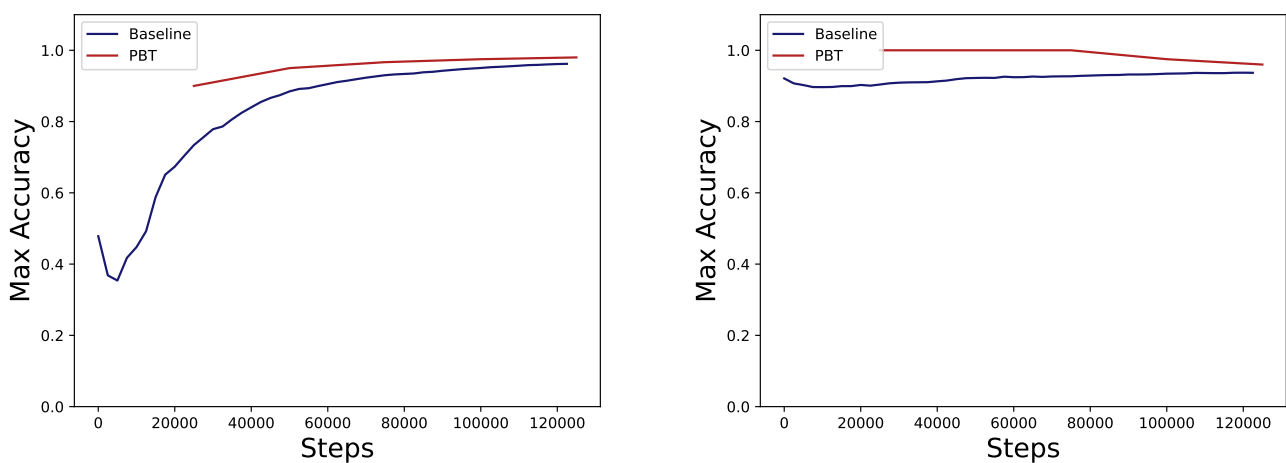


Figure 7.2: The best accuracy of the initialisation training (left) and the transfer training (right) for the Reinforce algorithm.

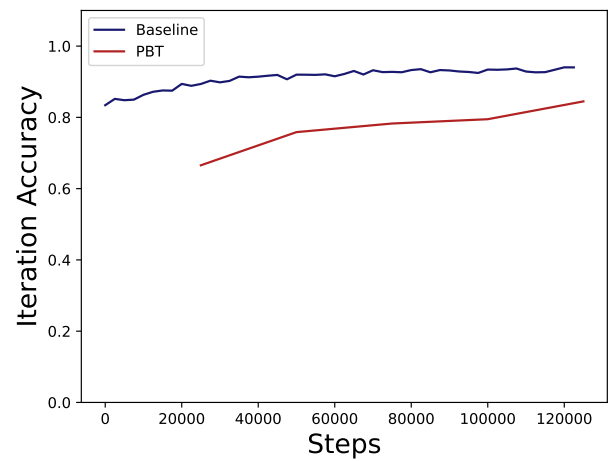
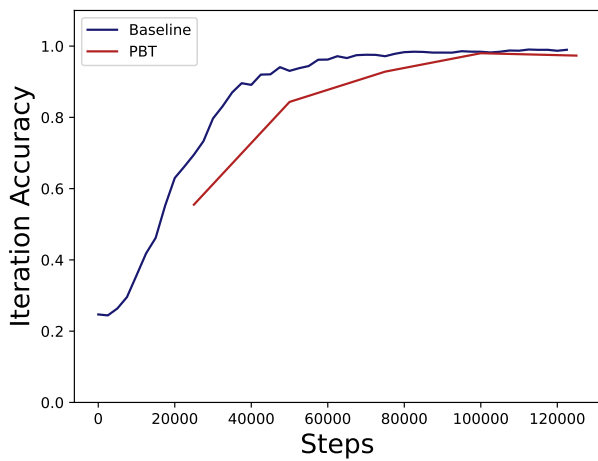


Figure 7.3: The mean accuracy per iteration of the initialisation training (left) and the transfer training (right) using the Reinforce algorithm.

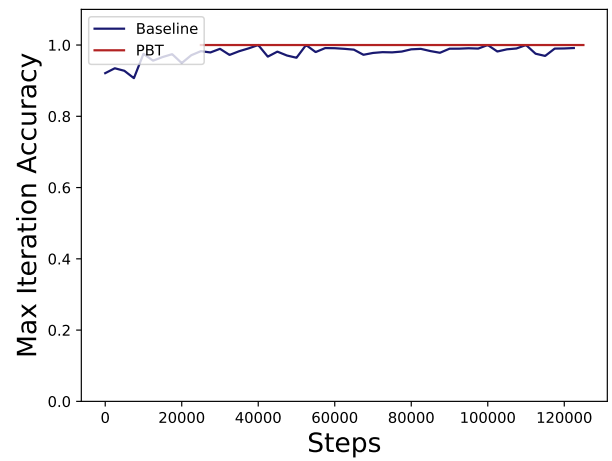
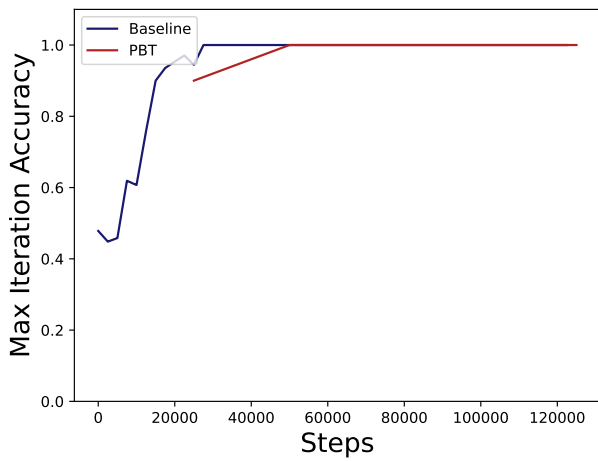


Figure 7.4: The best accuracy per iteration of the initialisation training (left) and the transfer training (right) for the Reinforce algorithm.

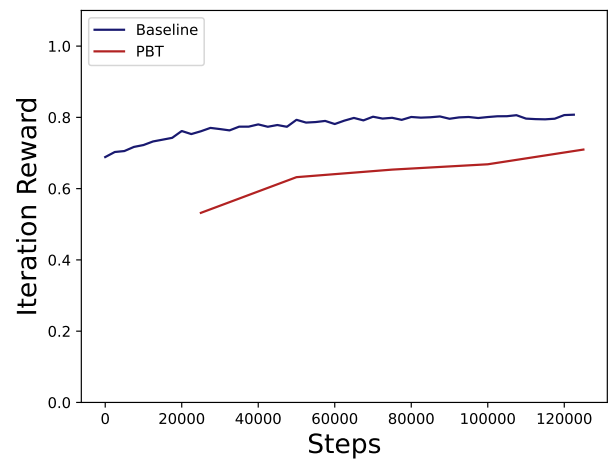
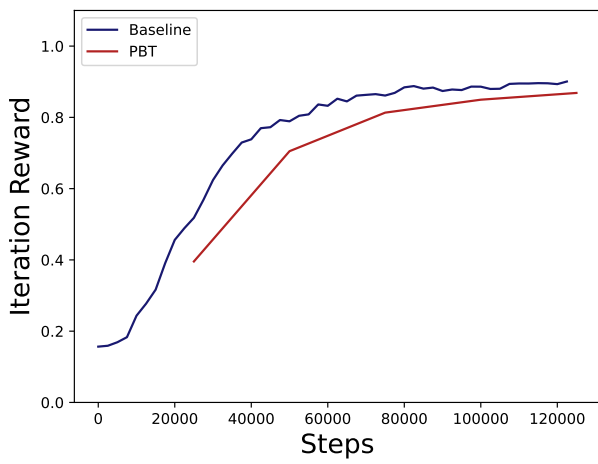


Figure 7.5: The mean reward per iteration of the initialisation training (left) and the transfer training (right) using the Reinforce algorithm.

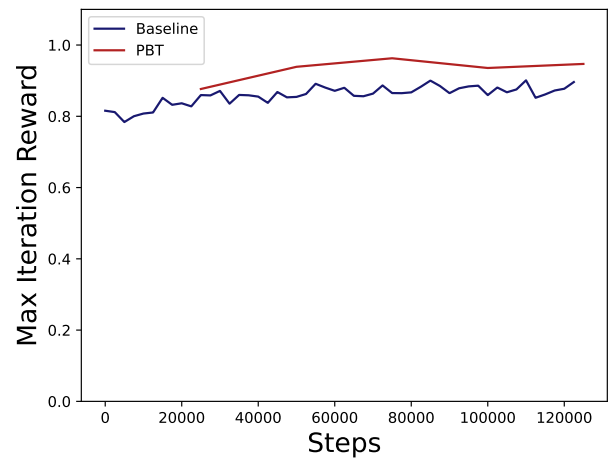
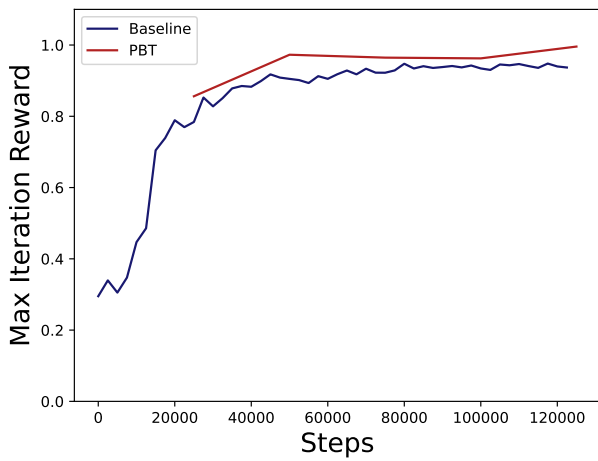


Figure 7.6: The best reward per iteration of the initialisation training (left) and the transfer training (right) for the Reinforce algorithm.

7.2 Deep Q-Network (DQN)

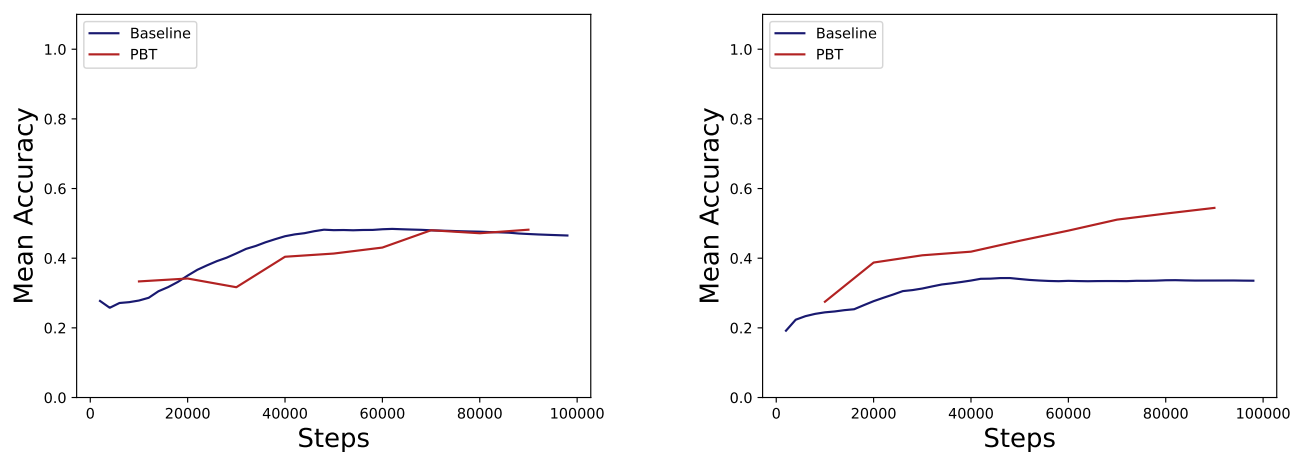


Figure 7.7: The mean accuracy of the initialisation training (left) and the transfer training (right) using the DQN algorithm.

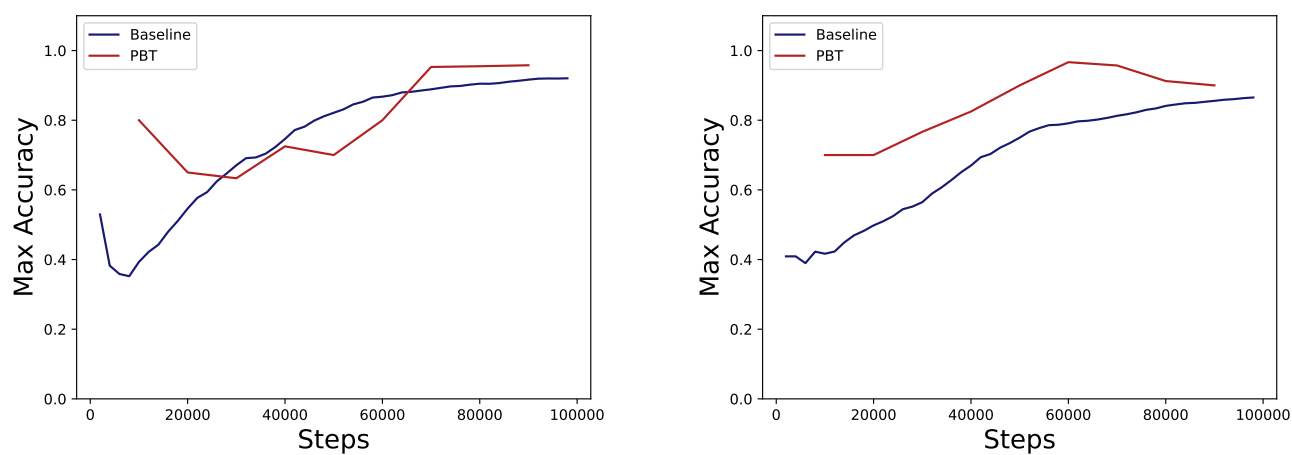


Figure 7.8: The best accuracy of the initialisation training (left) and the transfer training (right) for the DQN algorithm.

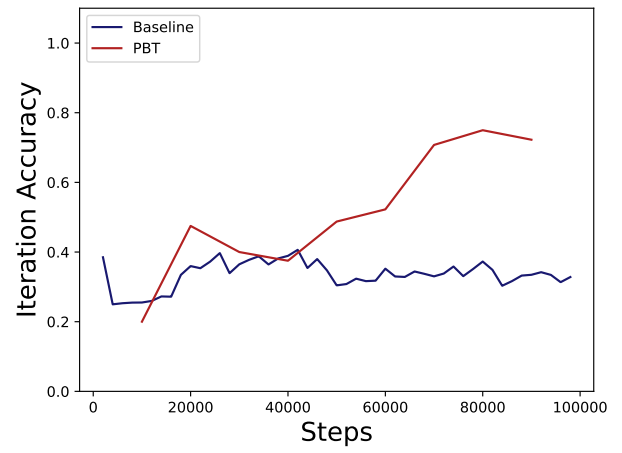
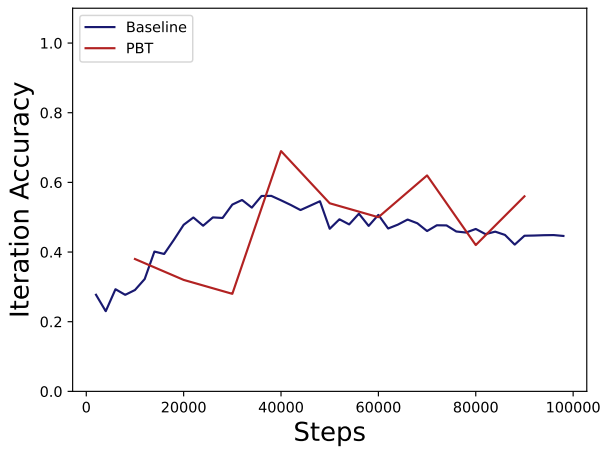


Figure 7.9: The mean accuracy per iteration of the initialisation training (left) and the transfer training (right) using the DQN algorithm.

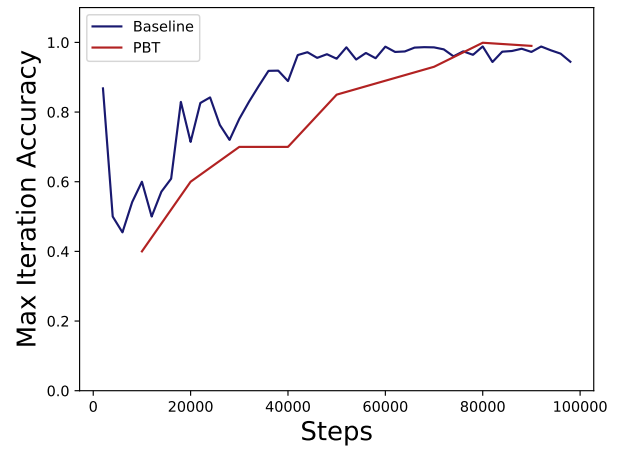
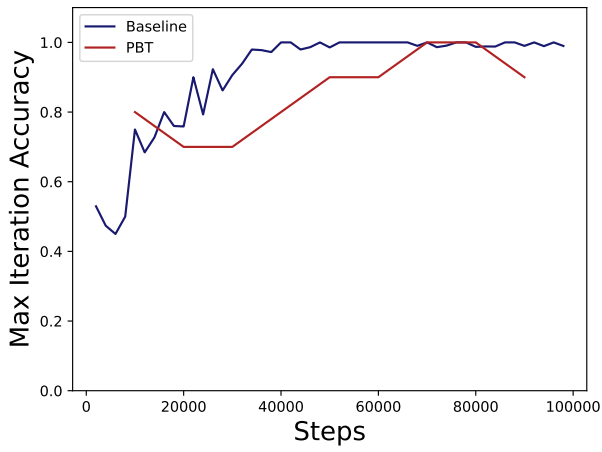


Figure 7.10: The best accuracy of the initialisation training (left) and the transfer training (right) for the DQN algorithm.

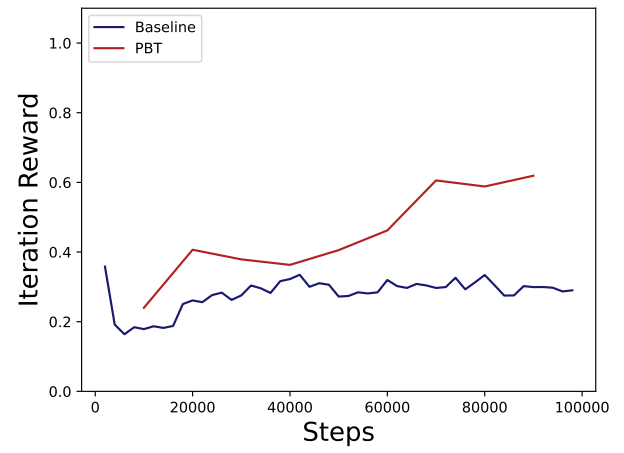
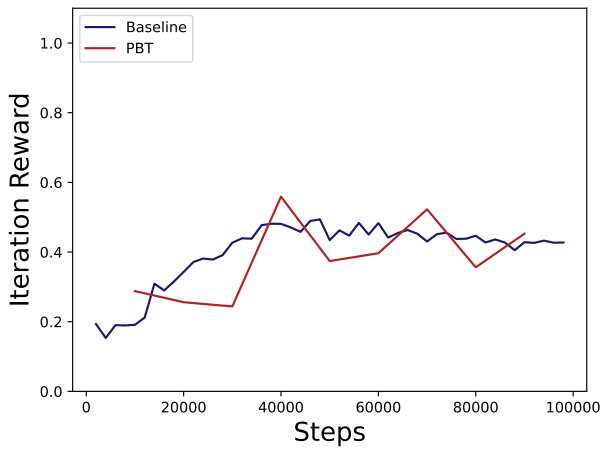


Figure 7.11: The mean reward per iteration of the initialisation training (left) and the transfer training (right) using the DQN algorithm.

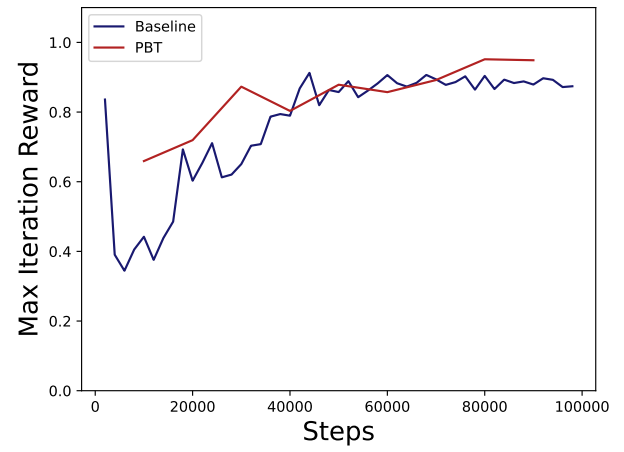
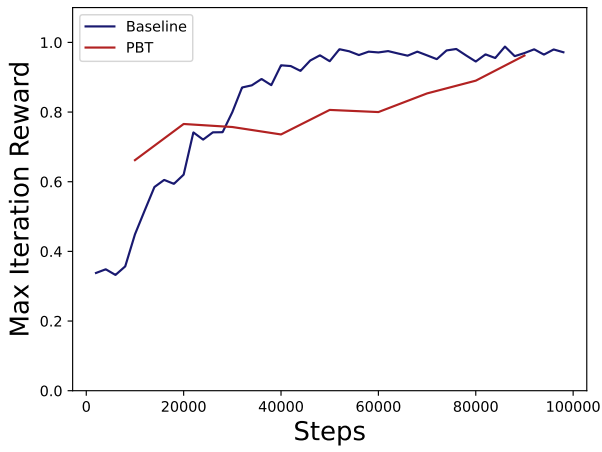


Figure 7.12: The best reward of the initialisation training (left) and the transfer training (right) for the DQN algorithm.

7.3 Proximal Policy Optimization (PPO)

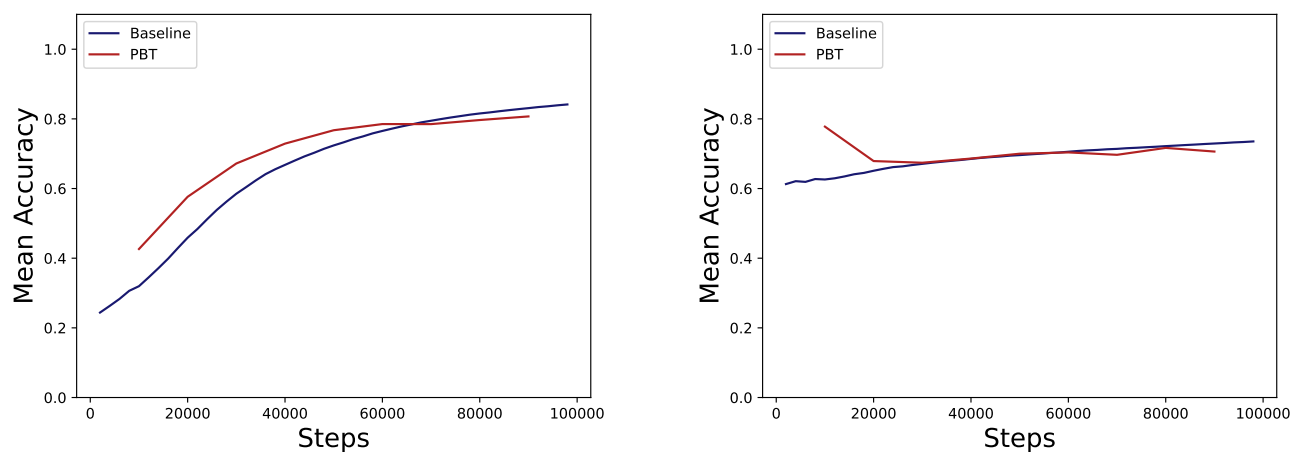


Figure 7.13: The mean accuracy of the initialisation training (left) and the transfer training (right) using the PPO algorithm.

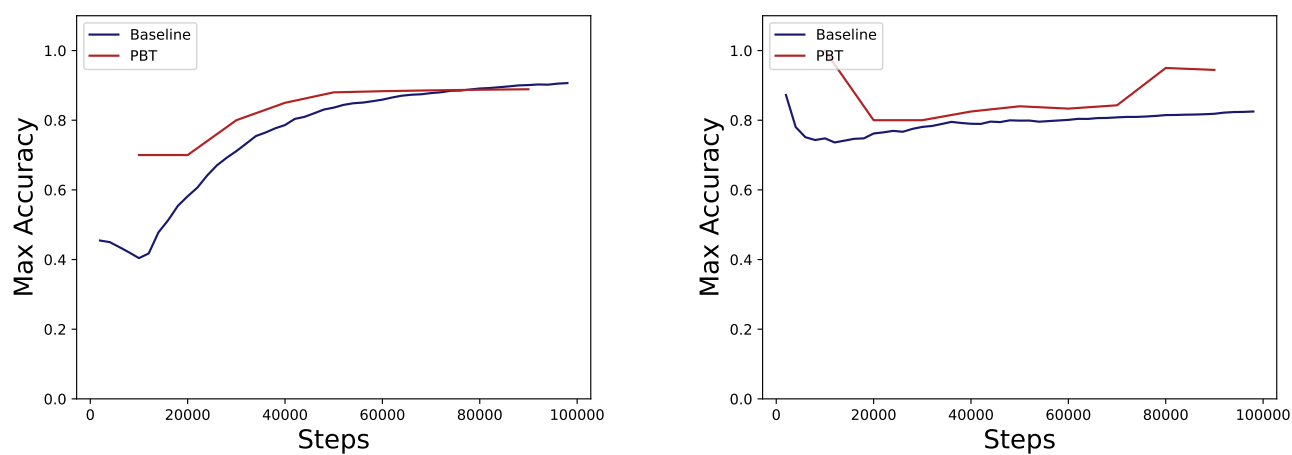


Figure 7.14: The best accuracy of the initialisation training (left) and the transfer training (right) for the PPO algorithm.

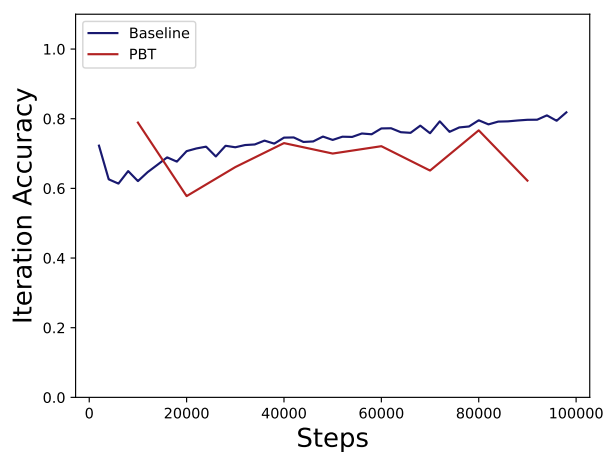
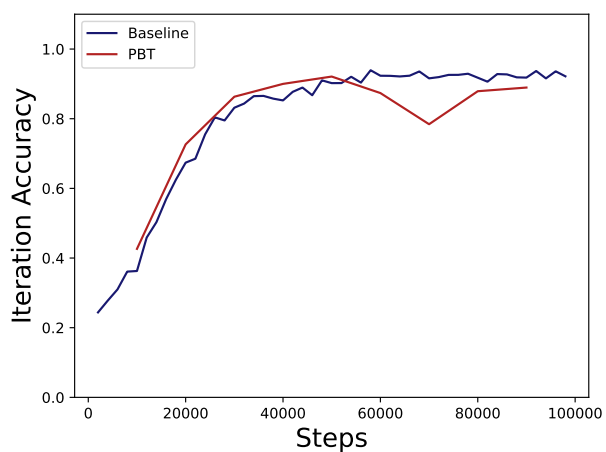


Figure 7.15: The mean accuracy of the initialisation training (left) and the transfer training (right) using the PPO algorithm.

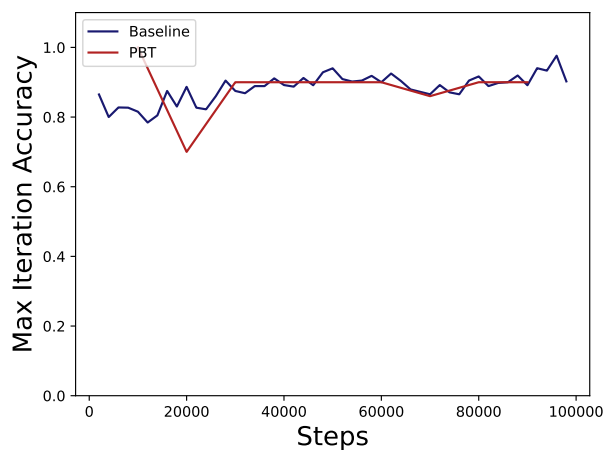
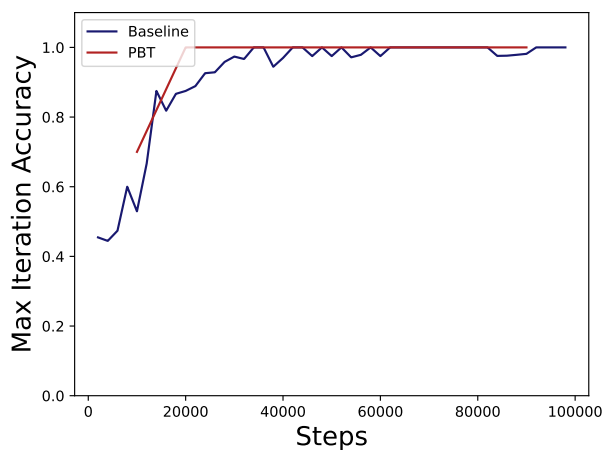


Figure 7.16: The best accuracy of the initialisation training (left) and the transfer training (right) for the PPO algorithm.

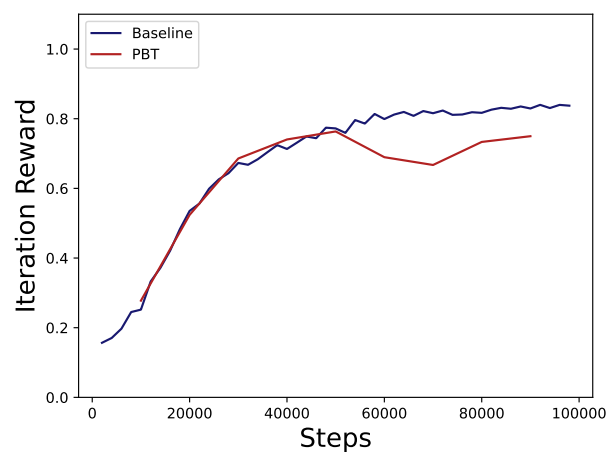
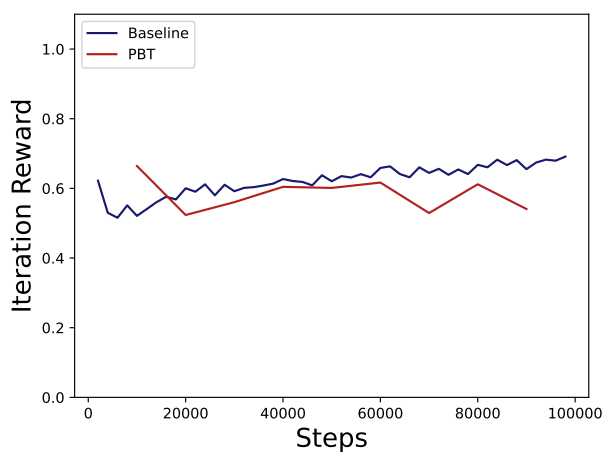


Figure 7.17: The mean reward of the initialisation training (left) and the transfer training (right) using the PPO algorithm.

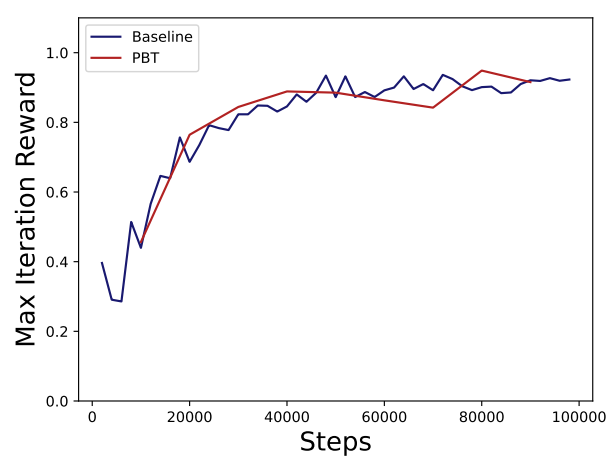
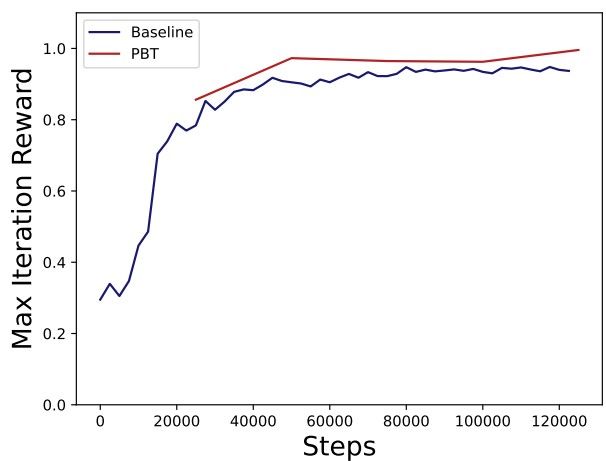


Figure 7.18: The best reward of the initialisation training (left) and the transfer training (right) for the PPO algorithm.

7.4 Transfer Population

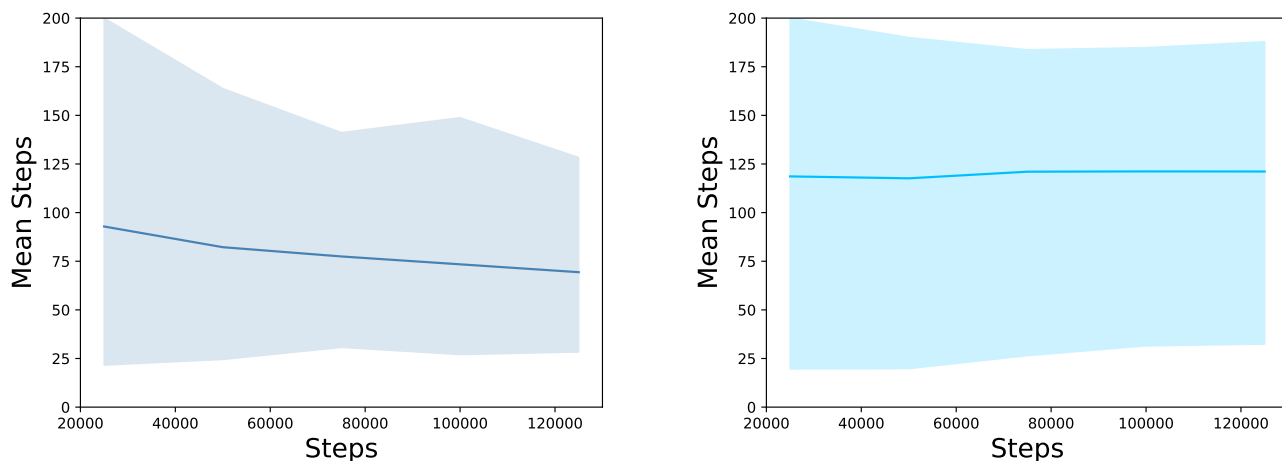


Figure 7.19: The mean reward with the different population initialization methods. On the left using the whole population and on the right using the best agent. The line represents the average value and the lighter area is the minimum and maximum span.

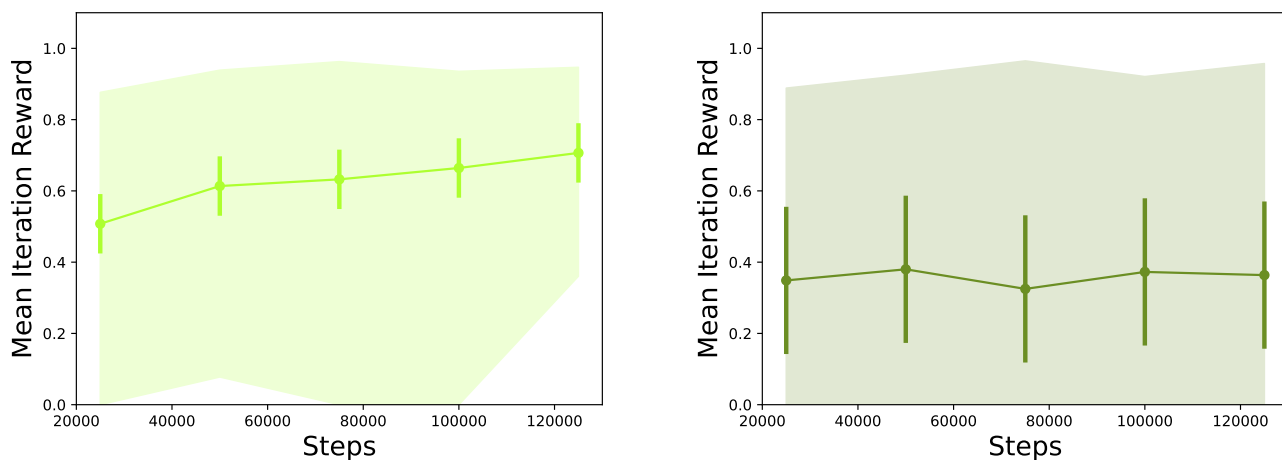


Figure 7.20: The mean reward per iteration with the different population initialization methods. On the left using the whole population and on the right using the best agent. The line represents the average value, the bars show the variance and the lighter area is the minimum and maximum span.

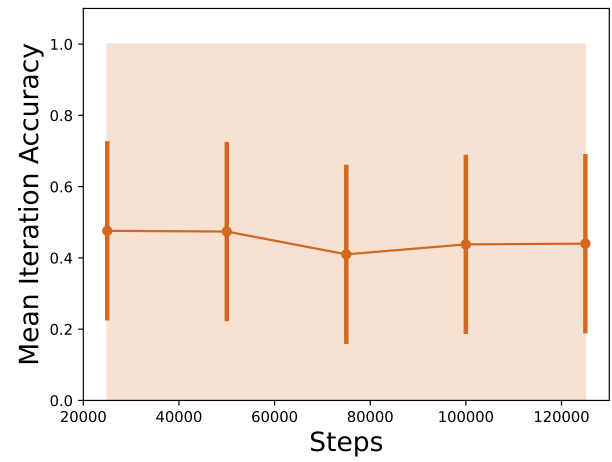
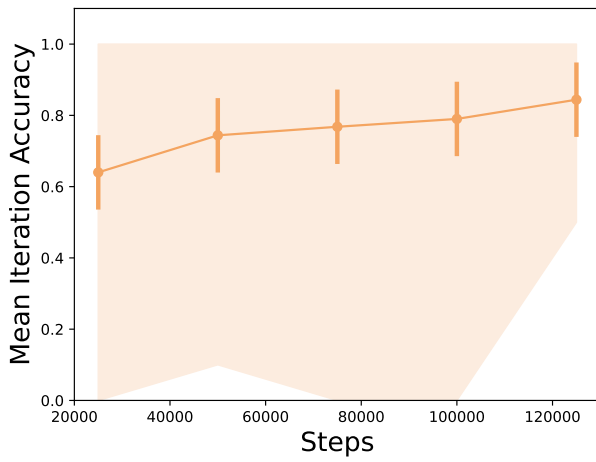


Figure 7.21: The mean accuracy with the different population initialization methods. On the left using the whole population and on the right using the best agent. The line represents the average value, the bars show the variance and the lighter area is the minimum and maximum span.

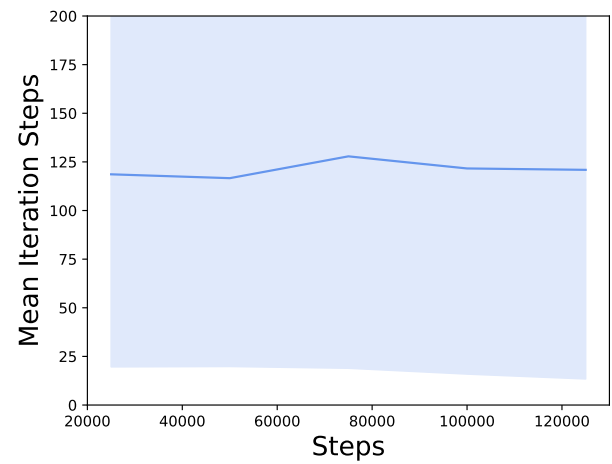
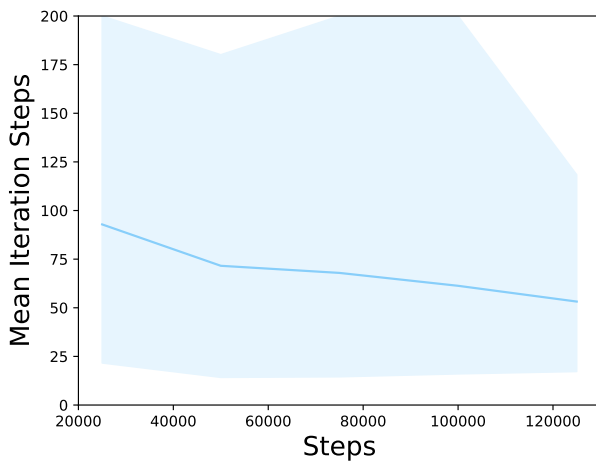


Figure 7.22: The mean steps per iteration with the different population initialization methods. On the left using the whole population and on the right using the best agent. The line represents the average value and the lighter area is the minimum and maximum span.