Growing with Experience: Growing Neural Networks in Deep Reinforcement Learning

---

Lukas Fehring

September 24, 2024
Version: Final Version

Institute of
Artificial Intelligence

Leibniz
Universität
Hannover

Electrical Engineering and Computer Science,
Institute of Artificial Intelligence (LUHAI)
Appelstraße 9a
30167 Hannover

Master's Thesis

Growing with Experience: Growing Neural Networks in Deep Reinforcement Learning

Lukas Fehring

1. Reviewer Prof. Dr. rer. nat. Marius Lindauer
Electrical Engineering and Computer Science
Leinbiz University Hannover

2. Reviewer Jun.-Prof. Dr.-Ing. Alexander Dockhorn
Electrical Engineering and Computer Science
Leinbiz University University

Supervisor M.Sc. Theresa Eimer

Date of Submission: September 24, 2024

Lukas Fehring
Growing with Experience:
Growing Neural Networks in Deep Reinforcement Learning
Master's Thesis, September 24, 2024
Reviewers: Prof. Dr. rer. nat. Marius Lindauer
and Jun.-Prof. Dr.-Ing. Alexander Dockhorn
Supervisors: M.Sc. Theresa Eimer
Automated Machine Learning (AutoML)
Institute of Artificial Intelligence (LUHAI)
Electrical Engineering and Computer Science
Welfengarten 1
30167 Hannover

Declaration

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie die aus fremden Quellen direkt oder indirekt übernommenen Stellen/Gedanken als solche kenntlich gemacht habe.

Hannover, September 24, 2024,

_____

Lukas Fehring

Abstract

The success of deep and complex artificial neural networks has transformed the field of Machine Learning. However, the reported difficulty in using highly-parameterized networks for Reinforcement Learning (RL) results in the community focusing on well-established, often lightweight networks. At the same time, promising approaches like Hyperparameter Optimization and other AutoML techniques, which could alleviate these challenges, often remain overlooked. In response, we propose using Multi-Fidelity Hyperparameter Optimization to train Reinforcement Learning agents, with the network's parameterization treated as fidelity increased over time. Instead of training networks from scratch, we employ Net2Net-style network morphisms to reduce both computational costs and resource consumption. Our experiments on the MiniHack Room and Ant environments demonstrate that this approach enhances the training process of deep networks and even aids in transferring performance to increasingly difficult tasks. Based on these results, we advocate for the adoption of this paradigm and recommend further research into growing neural networks and the related field of neural architecture search for RL.

Acknowledgement

In this work, I used *ChatGPT* as a tool for assistance with writing, i.e., to improve previously written content, for example, by fixing typographical errors. Additionally, I utilized it for general research, for example, to find alternative names for "Growing Neural Networks", as a coding assistant.
I also used *GitHub Copilot* as part of the code development process.
Furthermore, I used *Grammarly* to improve the clarity and correctness of the text.

This work was supported by the Federal Ministry of Education and Research (BMBF), Germany, under the AI service center KISSKI (grant no. 01IS22093C).

Contents

1 Introduction

1.1 Motivation and Problem Statement

In Reinforcement Learning (RL), the interaction between the agent and the environment is central: the agent selects actions based on the current states that the environment provides. Then, the environment analyzes these actions to generate the subsequent states and rewards. Through these experiences, the agent learns about the environment and how to maximize the accumulated rewards (Sutton et al., 2018).

Over the years, many RL algorithms have been proposed, often utilizing artificial neural networks, so-called policy networks, to parameterize the policy. However, while other areas of Machine Learning (ML) have moved beyond simple Multi-Layer Perceptrons and toward increasingly deep and complex neural architectures, this trend has only marginally transferred to the policy networks used in Reinforcement Learning (Parker-Holder et al., 2022), and to the best of our knowledge, there is still no survey paper focusing on architectures. Instead, architectures such as the Nature CNN, the original CNN used for Deep Reinforcement Learning (Mnih et al., 2013), and the Impala CNN (Espeholt et al., 2018) are reused in different experiments. This is likely partly rooted in the research finding that training deep networks for RL appears to be problematic (Sinha et al., 2020; Ota et al., 2021; Obando-Ceron et al., 2024).

In addition to slow adoption, the RL community has largely disregarded AutoML approaches, even though they could help with these issues. AutoML deals with automating decisions on algorithm selection (Rice, 1976), finding network architectures (Zoph et al., 2017), and configuring their training algorithms with an appropriate set of hyperparameter configurations (Kohavi et al., 1995; Hutter et al., 2019). Importantly, all of the above, especially poorly chosen hyperparameters, can have a major impact on training success. However, in addition to copying architectures between different scenarios, the RL community also often continues

using previously reported hyperparameters, hindering reproducibility and resulting in the reporting of subpar algorithm performance (Parker-Holder et al., 2022).

In this thesis, we aim to tackle both issues simultaneously by including the network parametrization in the AutoML loop as a fidelity. In the process of evaluating a Hyperparameter Configuration, we, therefore, start with a lightweight neural network combined with a sampled hyperparameter configuration and, if the agent performs well, transition to a neural network of increased complexity without costly retraining. To iteratively increase the number of parameters, we utilize the network morphisms Net2WiderNet and Net2DeeperNet proposed by (T. Chen et al., 2016). Essentially, they transfer the knowledge from a pre-existing smaller network to a new one containing either increased width or depth. This allows us to always evaluate a hyperparameter configuration on a network well equipped to learn while not suffering from over-parameterization, which could result in:

1. slower and more resource-intensive training due to increased computational expenses and hardware requirements

2. poorer policies, since increasing the number of neurons can decrease the quality of the learned policy (Sinha et al., 2020; Ota et al., 2021; Obando-Ceron et al., 2024).

Our approach, therefore, combines Bayesian Optimization (BO) (Bergstra et al., 2013) with Multi-Fidelity Optimization, as in (Jamieson et al., 2016), resulting in an approach very similar to Bayesian Optimization Hyperband (BOHB) (Falkner et al., 2018). While our experiments utilize relatively simple Multi-Layer Perceptrons, our successful results motivate conducting similar experiments on more complex architectures.

The approach is developed and evaluated using the MiniHack (Samvelyan et al., 2021) and Mujoco's Ant (Schulman et al., 2016) Reinforcement Learning environments. In MiniHack, an agent tries to navigate inside a grid, possibly encountering adversaries, and in Ant, the agent is tasked with controlling the movements of an ant robot simulation. In our evaluations, we also explore what happens if an agent trained on a simple MiniHack environment continues training on a more complex version with a policy network of increased complexity.

1.2 Research Questions

As mentioned, we focus on exploring the utilization of growing neural networks in Reinforcement Learning. To that end, we iteratively grow the neural network, building on multi-fidelity and hyperparameter optimization concepts.

To do so, we focus on the following research questions:

1. How can network growing approaches (prominent in other Machine Learning areas) be employed in RL?

2. In what way does the learning process change upon network adaptation?

3. When moving from simpler to more difficult problems, does iteratively increasing the depth/width of a pre-trained neural network yield improved results, compared to starting with the final/initial network size?

To that end, we:

- Develop an approach to combine multi-fidelity optimization with incremental network growth.

- Combine SMAC's multi-fidelity implementation with network growing operations.

- Analyze the observations given by MiniHack (Samvelyan et al., 2021) and explore how we can utilize them with a Multi-Layer Perceptron.

- Adapt MiniHack (Samvelyan et al., 2021), and NetHack (Küttler et al., 2020) to allow for reproducable experiments and include 10×10 grids.

- Implement the growth of a neural network using the operations proposed in Net2Net (T. Chen et al., 2016) and evaluate their impact on the training process on the training process of Reinforcement Learning Agents.

- Train and evaluate agents with growing architectures using the MiniHack and Ant environments.

1.3 Thesis Structure

To provide a comprehensive overview of the considerations made in this thesis, we begin by introducing both foundational and related work in Chapter 2. Next, we introduce our network growing approach and explain how it is combined with multi-fidelity in Chapter 3. Afterward, we describe our experimental setup in Chapter 4. We then present the results of our thesis in Chapter 5, and conclude by answering the research questions and highlighting future work in Chapter 6.

2 Related Work

This thesis' considerations aim to alleviate the tuning effort associated with making neural networks of increasing depth usable in Reinforcement Learning. This chapter provides the necessary background information by first introducing the foundational work of each research area and current related approaches.

In Section 2.1, we introduce Reinforcement Learning, focusing on the Proximal Policy Optimization algorithm used in this thesis, to then provide an overview of some related papers discussing the properties of neural networks in DeepRL. Following that, we introduce the area of AutoML, focusing on Hyperparameter Optimization and Neural Architecture Search in Section 2.2, including a discussion of related work in AutoRL. Lastly, we discuss growing neural networks in Section 2.3.

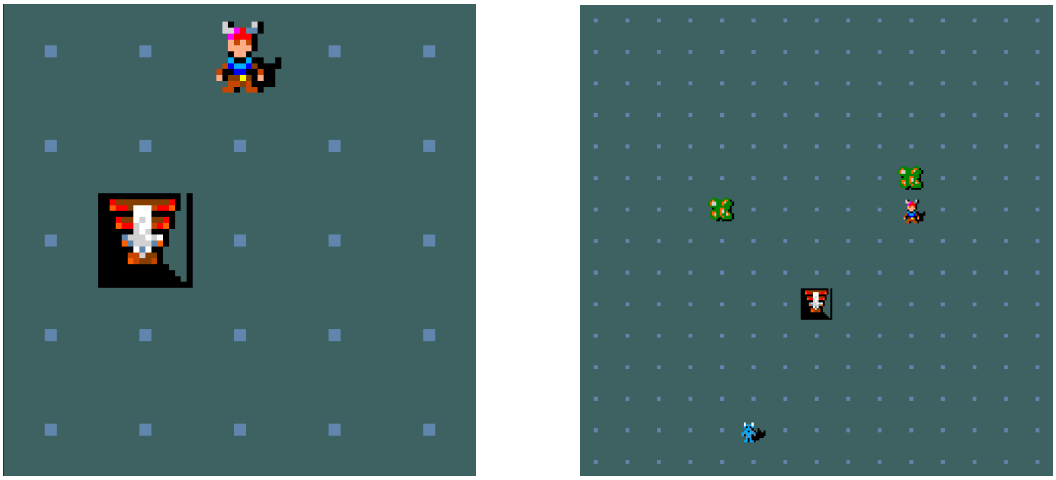
2.1 Reinforcement Learning

To provide an elementary understanding of Reinforcement Learning (RL), we first introduce the *MiniHack* and *Ant* learning environments in Section 2.1.1 to give us context for the introduction of the fundamental RL setup in Section 2.1.2. Building on these fundamentals, we introduce different algorithms in Section 2.1.3, including PPO used in this thesis. Lastly, in Section 2.1.4, we provide an overview of relevant research on the properties of neural networks used in RL and how these discoveries motivate our growing network approach.

2.1.1 Minihack

For development and initial experiments, we utilize MiniHack (Samvelyan et al., 2021), which is based on the NetHack Learning Environment (NLE) (Küttler et al., 2020), developed to train agents on the survival game NetHack. In NetHack, the

player controls an agent exploring a randomly generated dungeon to retrieve the Amulet of Yendor while avoiding being killed by enemies. To solve the overall task, the agent is tasked with various small subtasks, such as navigation and avoiding adversaries. This motivates Minihack, where the agent is only trained to solve a specific subtask. Focusing on one task at a time reduces training difficulty, while the consistent representation across different tasks allows for progressively training the agent on increasingly complex tasks.



(a) MiniHack Room 5x5 (b) MiniHack Room Ultimate 15x15

Figure 2.1.: Visual representation of the MiniHack Room 5x5 (2.1a) and MiniHack Room Ultimate 15x15 (2.1b) environments.

We focus on the navigation environments, specifically the Room environment, where the agent is tasked with moving toward a staircase to exit the room while avoiding adversaries. To achieve this, the agent can move one step in any direction based on the current representation of the entire room. Figure 2.1 depicts the varying task difficulties. In Figure 2.1a, the agent is placed in a 5x5 grid without adversaries. In Figure 2.1b, the agent is placed in a 15x15 grid containing a monster and two traps.

2.1.2 Fundamentals of Reinforcement Learning

The basic setup for Reinforcement Learning (RL) centers on the interactions between the Reinforcement Learning agent and its operating environment. The agent aims to accumulate rewards by iteratively interacting with the environment. A simplified visualization of the MiniHack Room environment can be seen in Figure 2.2, where

the interaction loop is portrayed on the left, and the environment state is portrayed on the right. Each time the agent provides an action to the MiniHack environment, the agent's position is updated, resulting in a positive reward upon reaching the goal and a negative reward when attempting to leave the grid (Sutton et al., 2018).

Formally, RL utilizes a Markov Decision Process (MDP) (Howard, 1960) consisting of the tuple (S, A, P, R). S denotes the state space, e.g., possible positions of the agent in an environment. A denotes the action space, e.g., moving $left$, $right$, up, or $down$ in a grid. $P : S \times A \times S \to [0, 1]$ maps each $state$, $action$, $state$ couple s, a, s' to a probability. This distribution defines how likely an agent currently in state s will transition into s' after executing action a. In Minihack, the uncertainty expresses itself in the random movement of adversaries, which move randomly. The reward function $R : S \to \mathbb{R}$ assigns a reward to a state; negative rewards are often assigned to illegal actions (leaving the grid), and positive reward is assigned for desirable behavior (the agent is at the goal position). A decision process can be formulated as Markovian for a sequence of states $(s_i, ..., s_j, s_{j+1})$, iff. the state s_{j+1} is independent of all previous states, provided s_j (Sutton et al., 2018).

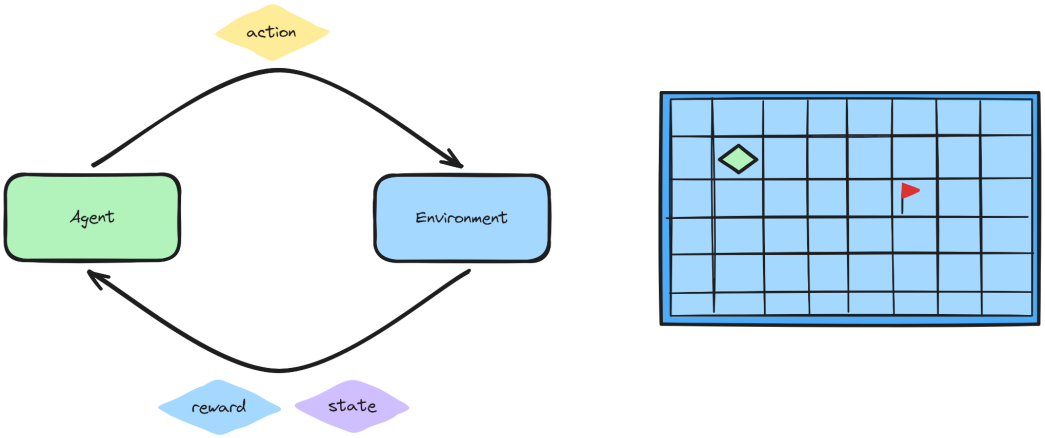


Figure 2.2.: The left image portrays the agent's and the environment's interaction. The agent chooses an action after being provided with an (initial) state. The environment, with its current state portrayed on the right, executes the action, resulting in a new state and reward provided to the agent.

However, the agent may be unable to observe the full state, increasing the difficulty. This is called a Partially Observable MDP (POMDP) (Lovejoy, 1991). In MiniHack, this occurs when the observations only provide information about grid positions within a certain distance from the agent. An example of this is shown in Figure 2.3, where the agent can only see the fields in the immediate neighborhood of its current position, further complicating the task.

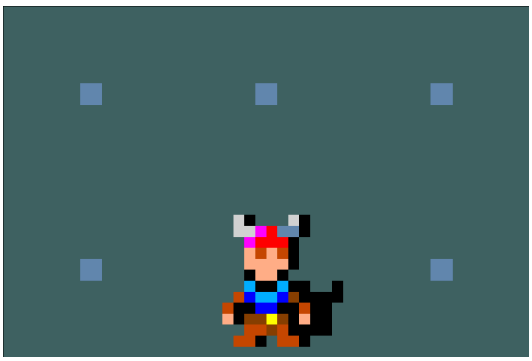


Figure 2.3.: Visualisation of a partially observable environment in MiniHack. The agent can only see its neighboring field, and all other fields are hidden from view.

When iteratively interacting with the environment, a sequence of states, actions, and rewards

$$\tau = (s_i, a_i, r_i, s_{i+1}, a_{i+1}, r_{i+1}, \ldots, s_n, a_n, r_n) \tag{2.1}$$

is generated, which is called a trajectory. While some environments allow arbitrary length trajectories, most automatically terminate them after n steps. If finite, trajectories are called episodes, and the underlying environment is denoted as episodic (Sutton et al., 2018). The discounted return gained by an episode

$$G_t = r_i + \gamma r_{i+1} + \cdots + \gamma^{n-i} r_n \tag{2.2}$$

is denoted G_t, with a parameter $\gamma \in [0, 1]$, configuring how immediate rewards are scaled against later rewards. (Sutton et al., 2018).

Given a state, the agent selects the next action based on a policy that maps each state to a probability distribution over actions:

$$\pi(a_i = a | s_i = s) \rightarrow [0, 1] \quad \forall s \in S, a \in A \tag{2.3}$$

For example, $\pi(left|s) = 0.7$ means that the agent will choose the action $left$ in the current state with a probability of 0.7.

Given a policy, the utility of being in each state can be quantified using the state value function

$$V^{\pi}(s_i) = \mathop{\mathbb{E}}_{(a_i, r_i, s_{i+1}, \ldots)} \left[\sum_{t=0}^{T} \gamma^t r_{i+t} \right] \tag{2.4}$$

which associates each state with the return gained through an episode by following π (Schulman et al., 2015).

A similar concept to the state value function is Q-values:

$$Q^{\pi}(s_i, a_i) = \mathop{\mathbb{E}}_{(r_i, s_{i+1}, a_{i+1}, \dots)} \left[\sum_{t=0}^{\infty} \gamma^t r_{i+t} \right] \tag{2.5}$$

Each Q-value associate, given a state s, each action with the expected reward generated by executing the action and then following π, (Sutton et al., 2018).

Utilizing these concepts, RL aims to find a policy

$$\pi^* \in \arg\max_{\pi} \mathbb{E}_{s \in S}[V^{\pi}(s)] \tag{2.6}$$

that performs optimally in expectation. RL agents often start with a randomly initialized policy and improve it iteratively (Sutton et al., 2018). In MiniHack, this might imply starting with a simple policy that avoids negative rewards and iteratively learning how to earn positive rewards until an agent can always walk directly toward the goal.

2.1.3 Reinforcement Learning Algorithms

In this thesis, we use Model-Free Reinforcement Learning, where the agent neither knows the MDP properties (e.g., the transition probabilities or rewards associated with an action) nor estimates them. Instead, the agent maximizes the expected rewards through trial and error (Shakya et al., 2023).

Value Based Algorithms For value-based algorithms, the process of finding an optimal policy is based on an iterative process of following a policy, updating the state or Q-values (Equation 2.4, and 2.5 respectively) to then update the policy. This means that in the beginning, the agent collects trajectories by following an initial, often randomly initiated, policy. Using these observations, learned state or Q-value estimates are collected, assigning a utility to different states or actions, which are then extracted to a policy. Prominent examples of extracting a policy are: For each state, always select the action with the highest Q-value, or convert each Q-value to a probability distribution using the softmax function (Sutton et al., 2018).

Initial value-based algorithms, such as Tabular Q-Learning (Watkins, 1989), used a lookup table to associate each state with a Q-value. Over time, many improvements to this paradigm, such as Double Q-Learning (Hasselt, 2010), which tackled the overestimation of Q-values by introducing a second Q-table were proposed (Sutton et al., 2018). However, introducing artificial neural networks in Reinforcement Learning has led to new state-of-the-art results with the advantage of replacing the Q-table with a neural network (Mnih et al., 2013). Instead of the states and actions in a lookup table, a feature vector is extracted from a state, which is then passed to a neural network. This also allows for generalization between states under the assumption that similar states have similar feature representations. Again, many algorithmic additions were proposed, increasing the agent's capabilities (Hessel et al., 2018; Schwarzer et al., 2023).

Policy Search Algorithms Instead of optimizing the policy through a proxy, as in Q-learning, Policy Search Algorithms aim to directly search for the optimal policy π^* either by gradient-free optimization, possibly with evolutionary strategies (Chrabaszcz et al., 2018; Qian et al., 2021) or searching for the optimal policy with gradient descent, which is the focus in this thesis.

For gradient following policy search algorithms, the policy $\pi(a|s, \theta)$ is parametrized using a neural network θ. Again, the agent then follows this policy to generate trajectories, compute the gradients for each step. Then, the network's parametrization is modified using gradient ascend to optimize the objective function. The base objective is

$$J(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t, s_t) G_t \right] \tag{2.7}$$

with G_t as the return starting from timestep t.

In the actor-critic paradigm, G_t is replaced by an advantage:

$$\hat{A}_t = -\hat{V}(s_t) + r_t + \gamma \cdot R(s_t + 1) + ... + \gamma^{T-t+1} R(s_{T-1}) + \hat{V}(s_T) \tag{2.8}$$

To compute this advantage, we utilize a state value estimate, called the critic, to center the advantage function around zero. This means that actions that result in a lower-than-estimated return reduce the objective, while actions resulting in a higher-than-expected return have a positive impact on the objective.

Applying the actor-critic architecture in MiniHack can help to effectively distinguish between suboptimal and optimal policies. Consider a scenario where the agent has reached the position next to the target but takes unnecessary steps under a suboptimal policy before reaching the goal. Here, the expected return G_t decreases due to these extra steps. Without a critic, returns resulting from following the suboptimal policy and those from choosing the optimal action to reach the target immediately would appear similarly beneficial. However, by employing the actor-critic method, the suboptimal actions result ina negative objective. Conversely, selecting the optimal action directly results larger objective.

However, policy search algorithms may suffer from the problem that unfortunate gradient updates might worsen the policy, effectively unlearning knowledge. This occurs partially due to excessively large gradient steps. Therefore, TRPO (Schulman et al., 2015) and PPO (Schulman et al., 2017) add Trust Regions to the optimization process, preventing overly large policy updates. In the MiniHack example, this follows the following intuition: Had the algorithm already learned to navigate the grid without accumulating negative rewards, a well-chosen trust region would only contain policies with the same property. Therefore, the algorithm would not unlearn previous knowledge.

To create trust regions, the algorithms quantify the difference between an old policy π_{old} and an updated policy π_{new} (Schulman et al., 2015; Schulman et al., 2017). Based on the difference, measures are taken for the update to fall into the trust region. These differ between PPO and TRPO. While PPO and TRPO perform similarly (Engstrom et al., 2020), we only investigate PPO in this thesis.

PPO As mentioned earlier, PPO (Schulman et al., 2017) utilizes an actor-critic approach during learning. To that end, PPO iteratively generates one or multiple episodes to then update the policy while dissuading excessively large updates.

The network, which produces a state value prediction and an action distribution, is depicted in Figure 2.4. Given a state representation (e.g., an image of the current grid in MiniHack), informative features are extracted. The value head then uses this feature representation to estimate the current state value $\hat{V}_{s_t}$. The policy head produces a score for each action (Schulman et al., 2017), which is converted into a probability distribution. During learning, the agent will sample actions from this distribution, resulting in different trajectories. The score generated by these

trajectories is then used to compute the advantage which is used in the objective function.

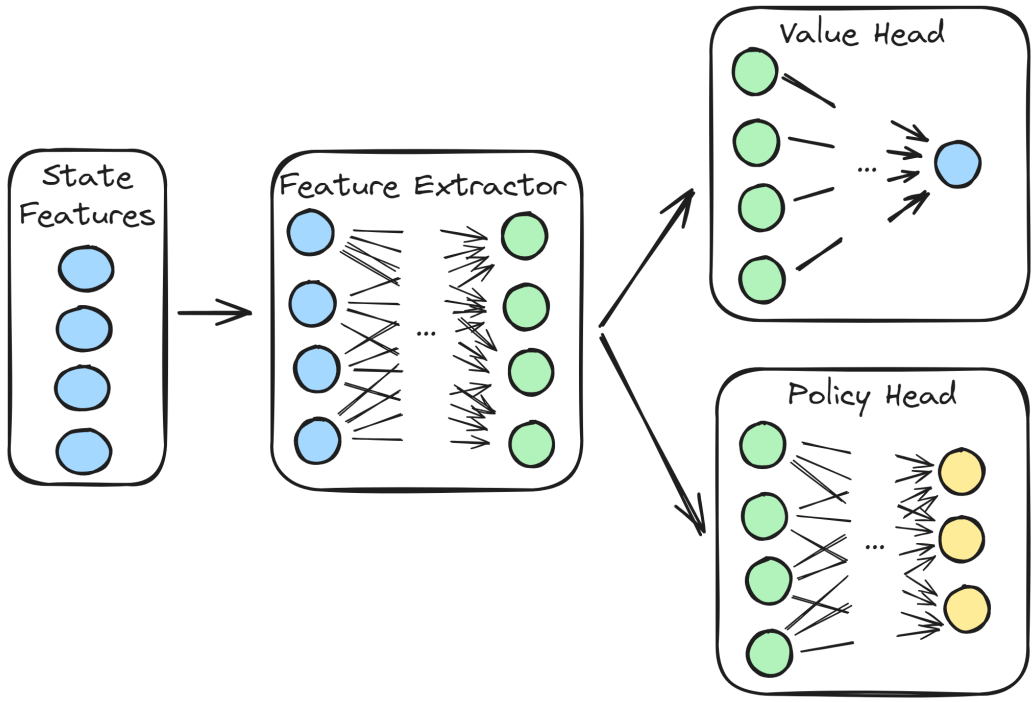


Figure 2.4.: Visual Representation of a network used for PPO. The initial state features are passed into a feature extractor, followed by a value head and policy head. If optimized via gradient descent, the joint objective will change the weights in all components.

To avoid overly large updates when updating the policy based on the objective several changes are made. Firstly, the actor-critic objective is replaced by a modified objective $J^{CLIP}(\theta)$ dissuading excessively large updates by using conservative policy iteration (Kakade et al., 2002) and objective clipping. Additionally, a value function objective (inverse value loss) $J^{VF}(\theta)$ is added to optimize the value head. Lastly, an entropy bonus $S[\pi_0](s)$ is added, facilitating exploration. This results in the following PPO objective

$$J(\theta) = \hat{\mathbb{E}}\left[J^{CLIP}(\theta) + c_1 J^{VF}(\theta) + c2S[\theta]\right] \tag{2.9}$$

weighted by the hyperparameters c_1, and c_2 (Schulman et al., 2017).

Note that both the clipping parameter ϵ and the weights in the final objective c_1, c_2 impact the learning process and must be tuned.

2.1.4 Neural Networks in Reinforcement Learning

Without the extensive amount of resources spent on developing increasingly complex Network Architectures, the recent rise of Deep Learning would not have been possible. These impressive results serve as a major influence on architectures in Reinforcement Learning. For example, the Transformer architecture (Vaswani et al., 2017) is utilized for a wide array of tasks in Reinforcement Learning (W. Li et al., 2023). However, at the same time, the Reinforcement Learning Community often still uses relatively shallow Multi-Layer Perceptrons and often struggles to make deeper CNN's work consistently (Parker-Holder et al., 2022; Obando-Ceron et al., 2024).

Dormant Neurons In Sokar et al., 2023, the authors discovered that Deep Q-learning agents often do not utilize all neurons effectively. Instead, a few neurons become dormant and will not help training or evaluation performance. Importantly, once a neuron becomes dormant, it will likely stay dormant. Similarly, in Abbas et al., 2023, the authors show that standard PPO also suffers from dormant neurons, resulting in a loss of plasticity, which implies that the agent will struggle to learn new behaviors. Both papers also introduce algorithms to reinitialize the neurons, thereby increasing the learning capabilities.

Architectures for Deep Reinforcement Learning Various architectures originally developed for supervised learning have been employed in Reinforcement Learning. In the foundational work of Deep Q-Learning, the authors train agents on Atari Games utilizing a, by today's standards, shallow and trivial CNN (Mnih et al., 2013). Building on this work, CNNs are considered the network architecture of choice for environments that process images (Parker-Holder et al., 2022). However, using the same trivial CNN for vastly different environments is counterintuitive, prompting the development of different CNN networks, for example, the IMPALA CNN (Espeholt et al., 2018; Parker-Holder et al., 2022), which results in higher performance. In Cobbe et al., 2019, the authors further explored the Impala CNN and the effect of other regularisation techniques from supervised learning. They showed that many of them can be helpful for Reinforcement Learning as well.
In addition to using CNNs, the RL community utilizes LSTMs for different learning

goals; for example, researchers enabled agents to learn the complex game of DOTA, even beating the world champions (Berner et al., 2019).

On the other side, Reinforcement Learning does not profit from increasingly heavy networks to the same extent as standard supervised learning (Sinha et al., 2020; Ota et al., 2021; Obando-Ceron et al., 2024). Additionally, Henderson et al., 2018 showed how sensitive the learning process can be to network architecture changes. Obando-Ceron et al., 2024 therefore proposes to make increasingly heavy networks usable by replacing linear layers with the soft mixture of experts architecture, where a gating mechanism is employed to activate/deactivate network parts based on the observation.

Student Teacher in Curriculum Learning While our approach of transferring the weights of an agent trained on an easier task to warm start the training on a related, more complicated task is rather trivial, it can be considered a handcrafted Curriculum Learning schedule. In contrast to that, Matiisen et al., 2017 proposes to automatize the selection of tasks on which the agent is trained using a teacher agent that selects tasks.

2.2 Automated Machine Learning

As mentioned above, in machine learning, algorithms differ in performance based on the task and their configuration. Thus, both the algorithm and its hyperparameter configuration impact the quality of a machine-learning solution. Selecting them may require immense expertise and will be accompanied by trial and error. This motivates Automated Machine Learning (AutoML) (Hutter et al., 2019), with its Hyperparameter Optimization (Bischl et al., 2023) subtask.

In Section 2.2.1, we first introduce the Hyperparameter Optimization Problem, followed by a discussion of prominent fundamental AutoML approaches, including but not limited to HPO, in Section 2.2.2. Building on that, we discuss where AutoML tools struggle when applied to RL in Section 2.2.3, and how they can be adapted by introducing Automated Reinforcement Learning.

2.2.1 Hyperparameter Optimization

Hyperparameter Optimization (HPO) aims to automatically equip an algorithm A with well-performing hyperparameter configurations inside a provided hyperparameter search space Λ. In supervised learning, this usually means selecting one lowest-cost hyperparameter configuration

$$\lambda^* \in \arg\min_{\lambda \in \Lambda} \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathcal{D}} \left[\mathcal{L}(A_\lambda, D_{train}, D_{valid}) \right] \tag{2.10}$$

trained on a dataset D_{train} and evaluated on D_{valid} from the space of possible datasets $\mathcal{D}$ (Hutter et al., 2019).

In contrast, in RL, this means finding the hyperparameter configuration

$$\lambda^* \in \arg\min_{\lambda \in \Lambda} \mathbb{E} \left[c(A, \lambda) \right] \tag{2.11}$$

regarding a cost metric c often the negative return over evaluation episodes; either from one, or from multiple environments (Eimer et al., 2023).

2.2.2 Approaches

Over the years, many HPO approaches have been introduced, ranging from simple methods such as Grid and Random Search (Bergstra et al., 2012), where configurations are selected on a grid or randomly, to more advanced approaches. Interesting comparisons to our methodology are Bayesian Optimization (Snoek et al., 2012), Multi-Fidelity Optimization (L. Li et al., 2017), and evolutionary algorithms (Loshchilov et al., 2016a). Our focus on the network architecture and dynamic network changes also motivates a comparison to Neural Architecture Search (Zoph et al., 2017) and Dynamic Algorithm Configuration.

Bayesian Optimization Given a costly-to-evaluate target function, such as training a neural network, Bayesian Optimization (BO) introduces a cheap-to-evaluate proxy, a so-called surrogate model trained to predict the target function.

Concretely, BO uses an iterative process of suggesting a new hyperparameter configuration based on the surrogate, evaluating it, and then updating the surrogate. Importantly, surrogate models, i.e., gaussian processes (Rasmussen et al., 2006) and random forests (Breiman, 2001), also provide uncertainty estimates. When selecting a new configuration, exploitation (selecting a configuration with high predicted performance) and exploration (selecting a configuration with high predicted uncertainty) are balanced using an acquisition function, e.g., expected improvement (EI) (Jones et al., 1998). Initially, an uncertain surrogate will lead to high exploration of the search space, to become more exploitive as the surrogate's uncertainty reduces (Snoek et al., 2012; Bischl et al., 2023). Over time, BO will grasp the hyperparameter search space, hopefully resulting in a well-performing hyperparameter configuration.

Multi Fidelity Optimization With a similar motivation as in BO, Mutli Fidelity Optimization approaches aim to reduce the cost of iterative training iterations while finding well-performing hyperparameter configurations. However, building on the belief that configurations with poor performance on reduced training resources will not achieve competitive final performance, they focus on early stopping the training for poorly performing hyperparameter configurations.

To that end, a dynamically assigned budget is introduced, such as the amount of epochs used to train a neural network. Then, configurations are sampled and trained for a reduced budget. Famously, Successive Halving (Jamieson et al., 2016) proposed using bandits to assign this budget dynamically while sampling hyperparameter configurations randomly. Starting with n configurations, the authors propose repeatedly disregarding half until only well-performing configurations remain. L. Li et al., 2017 follow a similar idea but proposes running hyperband multiple times, effectively conducting grid search on budgets schedules.

BOHB: Bayesian Optimization meets Hyperband As the title suggests, BOHB extends Hyperband by sampling the hyperparameter configurations with Bayesian Optimization. By combining the two approaches, one can save resources by stopping poorly chosen configurations early and using a surrogate to focus on promising parts of the configuration space (**BOHB**).

Evolutionary Algorithms Evolutionary Algorithms, inspired by evolution, where every generation, the population of considered data points, in this case, hyperparameter configurations, builds on the knowledge of the last generation. (Loshchilov et al., 2016b).
There are several different ways in which the information is transferred between generations. In Loshchilov et al., 2016b new configurations are sampled from a normal distribution, that is updated with each generation. However, the knowledge can also be transferred between generations after mutating a previously existing solution (Hutter et al., 2019). Loshchilov et al., 2016a applied these strategies for HPO and especially showcased the potential gains from parallelizing the execution inside of a population.

Neural Architecture Search To achieve state-of-the-art performance in deep learning tasks, it is essential to select not only optimal hyperparameters but also the most suitable deep learning model. Traditionally, this depends on the machine learning engineer's expertise in selecting either a preexisting or constructing a novel architecture. However, much like hyperparameter tuning, choosing or even constructing the appropriate network is a costly process relying on iterations of model training runs, thus motivating the idea of Neural Architecture Search, where a Machine Learning Algorithm is employed instead (Elsken et al., 2019b; White et al., 2023).
Many different approaches have been used in the area of Neural Architecture Search. Essentially, they are all based on the idea of defining a search space of considered architectures, then explored using approaches from Reinforcement Learning to one-shot methods (White et al., 2023). To utilize Reinforcement Learning for NAS, the agent iteratively decides on architecture components and is provided with the validation score as a reward (Zoph et al., 2017). Evolutionary algorithms, on the other hand, iteratively refine a population of architectures by sampling new structures, combining existing ones, or applying random mutations (Elsken et al., 2019a; White et al., 2023). Bayesian optimization approaches encode the architecture search space as a hyperparameter search space with a distance metric that compares different architectures (Bergstra et al., 2013; White et al., 2023). One-shot methods train all architectures in the search space simultaneously, often using meta gradients to optimize the architecture (Liu et al., 2018; White et al., 2023).

Dynamic Algorithm Configuration Dynamic Algorithm Configuration (DAC) can be seen as a generalization of the hyperparameter optimization framework. Instead of selecting hyperparameter configurations/a schedule of hyperparameter configura-

tions before training, DAC proposed to adjust them dynamically over the training process. This offers the possibility to, for example, dynamically adjust the learning rate of a neural network during training to avoid divergence, convergence to local optima and speed up the training process. This methodology was successfully used to adjust the evolutionary CMA-ES (Loshchilov et al., 2016a) algorithm dynamically (Adriaensen et al., 2022).

2.2.3 Auto RL

While Supervised Learning and Reinforcement Learning are closely related research fields, specific challenges in RL lead to research results not necessarily transferring between the two areas. For example, AutoML approaches are often developed for supervised learning, and the training behavior of RL algorithms adds its challenges (Parker-Holder et al., 2022). This motivates Automated Reinforcement Learning, geared towards either adapting existing AutoML approaches for RL or developing new approaches tailored directly to Reinforcement Learning (Parker-Holder et al., 2022).

Often cited challenges in the application of AutoML tools for RL are

- the erratic learning curves of Reinforcement Learning (Nguyen et al., 2020a)
- the impact environment and agent seeding have on training success (Henderson et al., 2018; Parker-Holder et al., 2022)
- the need for dynamic hyperparameter changes (Mohan et al., 2023)

However, contrary to the belief that these reasons lead to AutoML tools failing for Reinforcement Learning, standard hyperparameter optimization tools can still be employed as shown in “Hyperparameters in Reinforcement Learning and How To Tune Them”. Here, the authors employed standard HPO methods and concluded that tuning the hyperparameters for each environment results in large performance gains, even with high dimensional spaces.

Standard HPO for RL Similarly, HPO methods have already been successfully employed to tune not only hyperparameters but also a network architecture in practical

use cases. In (Runge et al., 2019), the authors employed BOHB, with wallclock time as fidelity to design RNA sequences. Similarly, during the development of AlphaGO (Silver et al., 2016), the authors used Bayesian Optimization to tune the Hyperparameters of their Reinforcement Learning Algorithm (Y. Chen et al., 2018).

To tackle the problem of HPO's eratic learning behavior (Nguyen et al., 2020b) proposed to compress an agent's learning curve into one numeric value, which does not only take into account the final performance but also the training process.

RL Specific Approaches Evolutionary Algorithms have been employed to tune hyperparameters in Reinforcement Learning. In (Jaderberg et al., 2017), the authors utilize evolutionary algorithms to optimize the hyperparameters of the Reinforcement Learning Algorithms. Their approach enables the discovery of hyperparameter schedules by transferring the neural networks of well-performing agents across generations. This process involves using the trained network of a successful agent as a warm start for agents with different hyperparameter configurations in the following generation. This approach has been extended to also adapt the network architectures (Franke et al., 2021). (Wan et al., 2022) built upon this idea by additionally using network distillation techniques to warm start the training of new architectures. They additionally use BO to determine both the trained architectures and used hyperparameters.

Lastly, the prominent DARTS (Liu et al., 2018) approach has also been successfully deployed in Reinforcement Learning, showing potential to employ existing Neural Architecture Search approaches in RL (Miao et al., 2022).

2.3 Growing Neural Networks

Like Neural Architecture Search, Growing Neural Networks also aim to optimize Neural Networks for their specific task. To that end, they incrementally grow a network to reach well-performing, appropriately sized networks. By following this approach, one can avoid excess computational power by training a computationally heavier network from scratch while also arriving at the desired network performance (T. Chen et al., 2016).

Fundamentals Utilizing Growing Neural Networks requires answering the three following questions (Evci et al., 2022):

- What triggers the growth of the neural network?
- How is the Neural Network grown?
- Where is the Neural Network grown

Notably, while some approaches focus on adding (single) neurons automatically during training,e.g., (Wu et al., 2019), others work on intervention often combined with a large increase of network parametrization (T. Chen et al., 2016; Wei et al., 2016). In our work, we focus on the second methodology. The initial network is called the teacher, and the resulting network is called the student.

Algorithms As a reference point to the other approaches discussed, we first introduce the Net2Net algorithm (T. Chen et al., 2016) utilized in this thesis. Net2Net is motivated by the observation that training a complex neural network often requires multiple iterations moving from smaller to more complex networks. To avoid excess computing and speedup training, the authors propose to warm start the training of an increasingly complex network with a smaller network. To that end, they propose Net2DeeperNet, which adds a layer, and Net2WiderNet, which increases the parametrization of a layer by splitting preexisting neurons.
This style of operation was later extended upon and called Network Morphism (Wei et al., 2016). They have also been used in Neural Architecture Search to warmstart increasingly complex models (Elsken et al., 2019a), and employed to train the DOTA agents in Reinforcement Learning where Berner et al., 2019 use a Net2Net style function operation to double the width of the used LSTM model's hidden state. In (Feng et al., 2020), the authors propose incrementally growing the network with Net2Net style operations upon adding additional training data, which can be seen as conducting multi-fidelity optimization over dataset subsets.

In (Pham et al., 2024), the authors propose to grow a network in width or depth by recombining already learned parameters of the layers. However, in contrast to Net2Net, these added parameters, while providing a strong initialization, change the output of the resulting network of increased parametrization. In (Wen et al., 2020), a CNN is iteratively grown in depth until the decrease in loss falls under a predefined threshold. The authors also challenge the idea of network morphisms by

showing superior results when adding minor noise levels. In (Mitchell et al., 2023), the authors determine when, where, and how (either in width or depth) to grow the neural network based on the perceived gradients.

In contrast to these major network transformations, another school of thought focuses on single neurons. (Bengio et al., 2005) phrases the number of neurons in a network as a convex optimization problem, yielding an algorithm that inserts hidden units while preserving the rest of the neural network until a budget is exceeded. Similarly, Wu et al., 2019 focus on deciding which neurons to split to learn lightweight, efficient architectures utilizing the gradients. In (Evci et al., 2022), the focus is on adding neurons to leave local optima by analyzing the gradient not to improve performance immediately but the resulting training dynamics.

3 Approach

In this chapter, we discuss our approach to integrating network growth into Hyperparameter Optimization, thereby enabling the usage of deeper networks to train Reinforcement Learning Agents.
In Section 3.1, we provide a general overview of our approach. Building on that, we explain the process of growing the policy network in Section 3.2. Lastly, we explain how we adapt Multi-Fidelity Optimization to integrate growing neural networks in Section 3.3.

3.1 Approach Overview

PPO aims to optimize its policy by iteratively updating the policy network based on accumulated returns. The policy is improved via bootstrapping, from random initialization to a well-performing policy. The training process must be configured with fitting hyperparameter configurations to perform well (Eimer et al., 2023). Importantly, in the original PPO approach, the policy network's architecture is static (Schulman et al., 2017). In contrast our approach explores combining multi-fidelity optimization with the growth of the policy network to early stop poorly performing training runs and enable training agents with increased policy complexity.

Growing Neural Networks As mentioned, we treat the parametrization of the network as fidelity, and upon transferring between fidelities, we grow the network by applying a network morphism. This means retaining the preexisting representation and adding additional neurons such that the underlying function is not changed. Thus, a trained navigation agent will not unlearn navigation but might use the additional weights to learn and avoid adversaries.
Our approach considers two network adaptations *Net2DeeperNet* and *Net2WiderNet* proposed by T. Chen et al., 2016. *Net2DeeperNet*, which adds an entirely new layer, and *Net2WiderNet* adds neurons to a preexisting layer. We discuss the details of both network morphisms in 3.2.

Network Parametrization as Fidelity Our approach to multi-fidelity Optimization is motivated by Bayesian Optimization Hyperband (Falkner et al., 2018) but works with a static fidelity schedule similar to Successive Halving (Jamieson et al., 2016). It involves sampling configurations, training them on stepwise increasing fidelity, and discarding poorly performing configurations. The resulting process is shown in Figure 3.1.

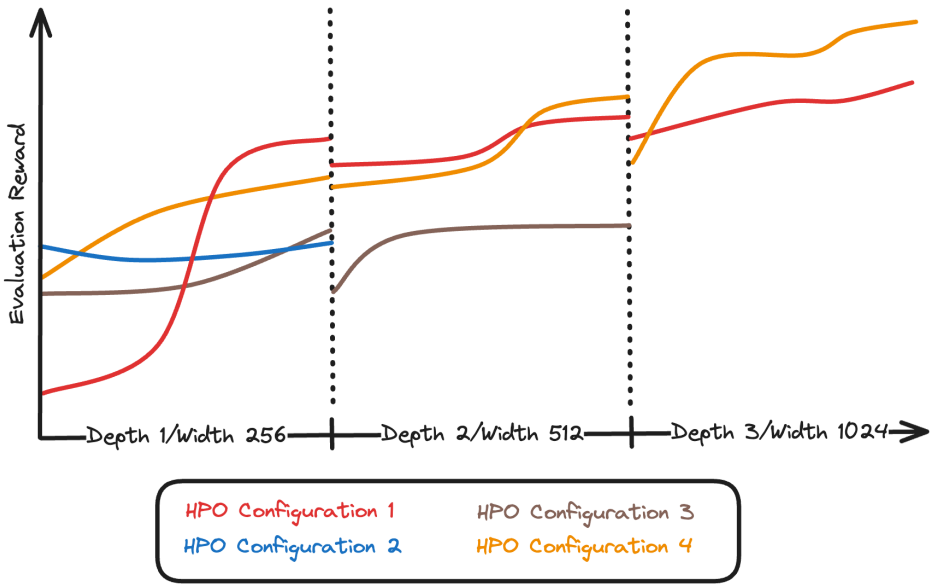


Figure 3.1.: Visualization of the proposed combination of multi-fidelity optimization, with the network parameterization as the fidelity.

In the beginning, a predefined number of hyperparameter configurations is sampled. Then, for each hyperparameter configuration, PPO is used to optimize a policy network over a predefined amount of environment interactions. Then, poorly performing agents and their hyperparameter configurations are disregarded while the well-performing transition to higher fidelity. Upon transitioning from a lower to a higher fidelity, the feature extractor is grown, resulting in an agent equipped with additional parameters. This process is iterated until a predefined amount of parameters in the policy-network is reached.
After reaching the maximum parametrization, the process is restarted. While, at first, configurations are sampled randomly, the following iterations utilize Bayesian Optimization.

3.2 Feature Extractor Growth

To expand the neural network, we employ the transformations *Net2DeeperNet* and *Net2WiderNet* as introduced in "Net2Net: Accelerating Learning via Knowledge Transfer" (T. Chen et al., 2016). In the following paragraphs, we explain how and where we apply the morphisms.

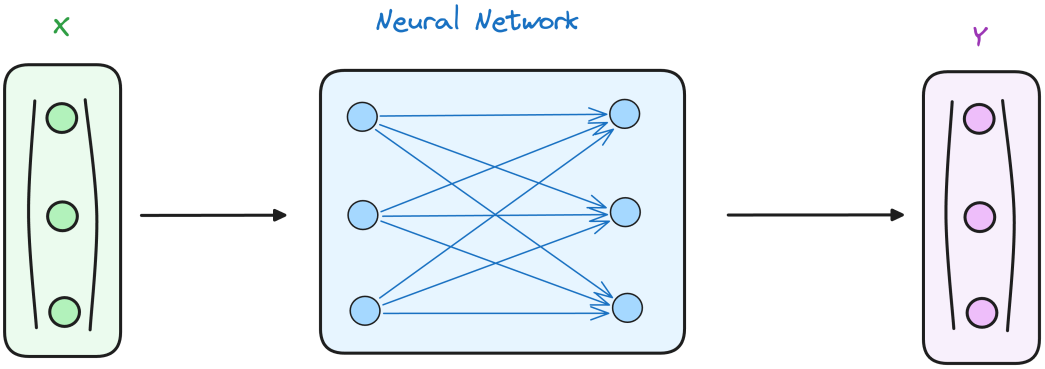
Where is the Neural Network Grown? As mentioned in Section 2.1.3, PPO utilizes a policy network containing a separate feature extractor whose output is utilized in both the policy head and value head. To maximize the impact, we, therefore, grow the network of the feature extractor.

For Net2DeeperNet, we add layers at the end of the feature extractor. This allows for the added layers to profit from and refine the representations already learned on the previous layers. This follows the research result that deeper layers may learn more complex structures (LeCun et al., 2015).

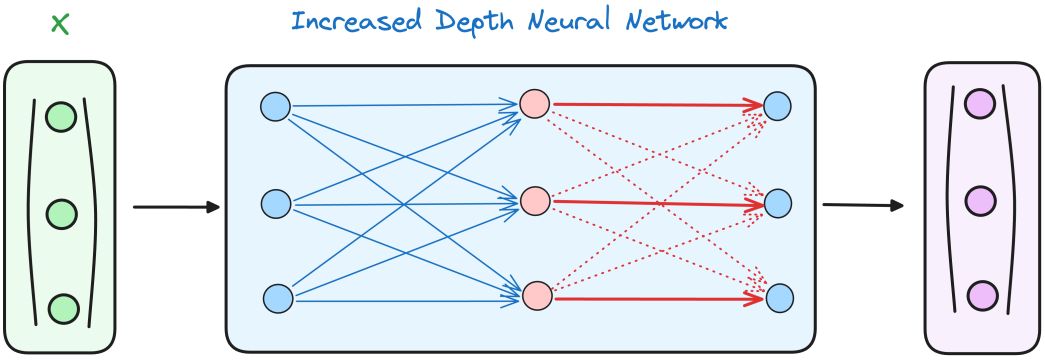
For Net2WiderNet, we propose simultaneously growing the width of all feature extractor layers while not adapting the input and output size to the feature extractor. This allows the feature extractor to learn more complex representations internally but forces it to compress them to the same size as before growing. By this approach, we allow the network to utilize the increased parameterization internally without affecting any of the other network components.

How does Net2DeeperNet work? Net2DeeperNet increases the depth of a feature extractor by appending an identity layer to the end of the feature extractor. Therefore, the added layer does not adapt the network functionality. The transformation is represented in Figure 3.2, with its Subfigures before transformation (represented in Subfigure 3.2a) and after transformation (represented in Subfigure 3.2b).

Note that additional conditions must hold for Net2DeeperNet not to change the network. Firstly, the added layer must not contain a bias term; otherwise, the added bias will change the network output. Secondly, since every layer is followed by an activation function, only idempotent activation functions may be used. This means that only an activation function f, not changing the input vector if applied a



(a) Visualization of the network state before applying **Net2DeeperNet**. A feature vector X is passed to the network, generating a putout vector Y.

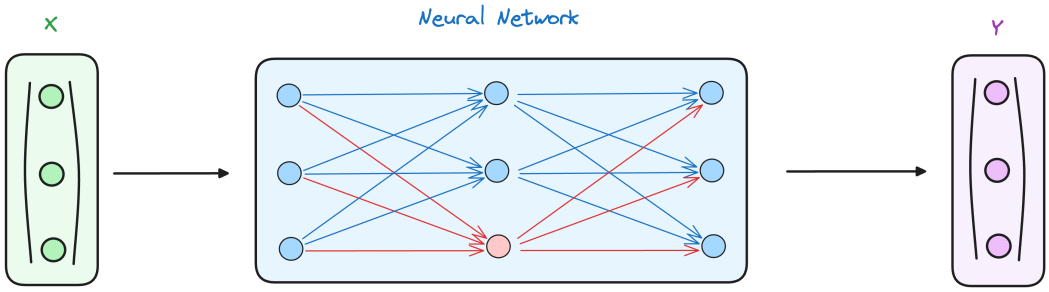


(b) Visualization of the network state after applying **Net2DeeperNet**. A feature vector X is passed to the network, generating an output vector Y. The additional red layer and its connections are added as an identity. Solid red arrows represent a weight of 1, and dashed red arrows represent a weight of 0.

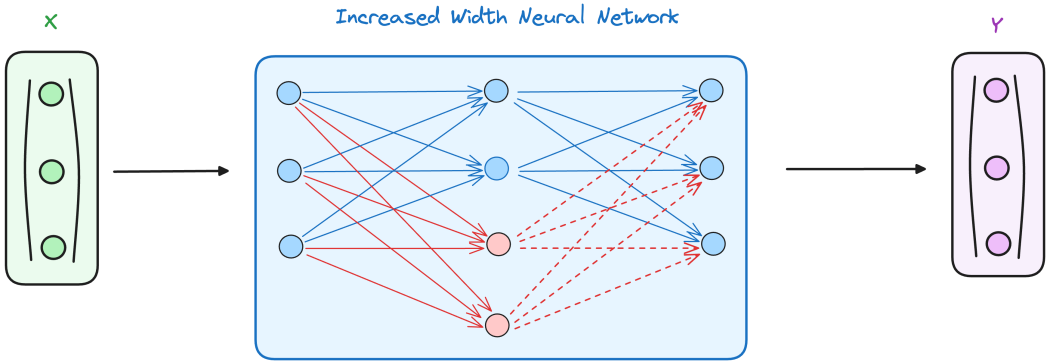
Figure 3.2.: Visualization of the **Net2DeeperNet** operation. Figure 3.2a visualizes the network state before, Figure 3.2b visualizes the network state after transformation. In both figures, the feature vector X results in the identical output vector Y.

second time $f(x) = f(f(x)) \forall x \in \mathbb{R}^k$, may be used. Otherwise, adding a layer would change the network due to the activation function. In our approach, we use ReLU, the activation function highlighted in the original approach.

How does Net2WiderNet work? The operation Net2WiderNet increases the network parameterization by adding additional neurons to a preexisting layer. This is achieved by splitting existing neurons as represented in Figure 3.3.



(a) Visualization of the network state before applying **Net2WiderNet**. A feature vector X is passed to the network of depth two, generating an output vector Y.



(b) Visualization of the network state after applying **Net2WiderNet**. A feature vector X is passed to the network, generating an output vector Y. The solid red arrows represent copying the input weight to the preexisting neuron to both red neurons. The dashed lines represent previously existing weights divided by 2.

Figure 3.3.: Simplified visualization of the **Net2WiderNet** operation. Figure 3.3a visualizes the network state before, and Figure 3.3b visualizes the network state after transformation. In both figures, an identical green feature vector X is passed to the neural network, and the network generates an identical purple output vector Y.

Figure 3.3a shows the initial neural network. The initial network takes a feature vector X as an input, passes its values through the neural network, and computes a prediction vector y.
After transformation, the network includes an additional neuron as shown in Figure 3.3b. This is achieved by splitting one of the internal neurons into two. This means copying the input weights and dividing the output weights by two. Then, the output vector has the same weight as above.

More generally, a preexisting neuron can be split n-ways by copying the input weights and dividing the output weights by n. This results in $n - 1$ additional

neurons. Given a layer of width m, to apply Net2WiderNet with a growth factor of two to double the number of neurons, Net2WiderNet randomly samples m neurons with replacement. A neuron sampled $k-1$ times is then split into k new neurons with copied input weights; the output weights are divided by k.

However, splitting the weights equally would mean that all resulting neurons contribute the same to the prediction, meaning they also receive the same gradient updates. As a result, the training of the neural network would always yield the same updates for their weights. Therefore, there would be no effective increase in parameterization. To avoid this, a small amount of noise is added to the split weights so they learn a different representation. Note that this means that Net2WiderNet does, strictly speaking, change the representation learned in the initial network. In order to keep the input and output shapes constant, one needs at least two layers in the feature extractor, otherwise either the input or output dimension would change.

3.3 Muilti-Fidelity Optimization for Network Growth

This section outlines key considerations for employing multi-fidelity optimization to enhance the policy network. Fundamentally, our approach modifies BOHB (Falkner et al., 2018) to only contain one bracket with a predefined schedule. Our algorithm explanation is divided into the inner loop for evaluating hyperparameter configurations and the outer loop to sampling new configurations.

Inner Loop The inner loop is represented in Algorithm 1. The algorithm takes n hyperparameter configurations $[\lambda_1, ..., \lambda_n]$, a maximum budget, and a budget schedule $b_1, ..., b_m$ as parameters. $b_1 = n$ specifies the number of initially sampled configurations that utilize the initial network; b_2 the number of configurations evaluated after growing the network once; b_m the number of configurations evaluated on the highest fidelity.
In total, there are m executions of the inner loop, one for each budget. Each loop iteration starts by training the current policy network configured with the given hyperparameter configurations. The policy networks are then evaluated and ranked based on the accumulated reward generated by following the policy for 10 evaluation episodes. Building on the rankings, only the best b_{i+1} configurations remain, and the others are disregarded. If not on the highest fidelity, a Network Morphism is

applied to increase the policy network parameterization.

Algorithm 1 Inner Loop

Require: $[\lambda_1, ..., \lambda_n]$ *// Initial hyperparameter configurations*
Require: $[b_1, b_2, ..., b_m]$ *// Budget configuration schedule*
$[\theta_1^1, ..., \theta_n^b] \leftarrow$ PolicyNetworks() *// Initialize policy networks*
Current $\leftarrow (\lambda_1, \theta_1^1), ..., (\lambda_n, \theta_n^1)$
for $i = 1$ to m **do**
    $r_1, \ldots, r_{b_i} \leftarrow$ PPO(Current) *// Train and evaluate policy networks*
    Ranking $\leftarrow$ Rank(Current, $r_1, \ldots, r_{b_i}$) *// Rank by Evaluation Rewards*
    Current $\leftarrow$ Update(Current,b_i) *// Remove poorly performing Configs*
    New $\leftarrow$ []
    for λ_j, θ_j^i in Current **do**
        if $i < m$ **then** *// If not Highest Budget, apply Morphism*
            $\theta_j^{i+1} \leftarrow$ Network Morphism(θ)
            New.append($\lambda_j, \theta_j^{i+1}$)
        else
            New.append(λ_j, θ_j^i)
        end if
    end for
    Current $\leftarrow$ New *// Update Current Configs*
end for

To ensure fair rankings, we consider 5 different initial policy networks. Each sampled hyperparameter combination is then used to train on all five policy networks and average the generated results. When transitioning between fidelities, we grow all 5 copies of the policy network.

Outer Loop The outer loop is represented in Algorithm 2. In addition to a budget schedule and the amount of PPO executions n_trials, the min_trials parameter must be provided to determine on which fidelity the surrogate model is fit.
During the outer loop, the algorithm mainly samples hyperparameter configurations followed by executions of the inner loop. Random configurations are sampled until the number of configurations evaluated on the minimum budget exceeds min_trials. Starting then, configurations are sampled using Bayesian Optimization on the highest fidelity for which already min_trials configurations are evaluated.
As a result the approach will sample configurations performing well on the lowest fidelity in the first BO runs. Later configurations will be sampled to perform well for the highest budget trained with a stepwise increasing parameterization.

The loop continues until PPO has been n_trials times. Importantly, exceeding n_trials will also immediately stop the inner loop.

Algorithm 2 Outer Loop

Require: n_trials CommentMax of PPO Execution
Require: $[b_1, b_2, ..., b_m]$ *// Schedule of Configurations per Budget*
Require: min_trials *// Trials needed to Fit Surrogate Model*
$[t_0, ..., t_m] \leftarrow [0, ..., 0]$ *// Counter of Evaluated Trials per Budget*
while $max\_ppo\_executions$ not exceeded **do**
  if $min\_trials \geq t_0$ **then** *// If not enough Data on lowest Budget*
    $[\lambda_1, ..., \lambda_n] \leftarrow SampleRandom()$ *// Sample Configurations randomly*
  else
    $data \leftarrow SelectDatapoints([t_0, ..., t_m])$ *// Select Data for BO*
    $[\lambda_1, ..., \lambda_n] \leftarrow SampleBo(data)$ *// Sample Configurations with BO*
  end if
  Inner Loop($\lambda_1, ..., \lambda_n$)
end while

4 Experiments

To address our research questions on how growing neural networks enhance the performance of reinforcement learning agents, we conduct a series of experiments. In Section 4.1, we provide a detailed description of the different environments used, including discussions on potential preprocessing steps and the reward structure. Section 4.2 introduces the various experiments conducted. Here, we present both our approach and the baseline models, and discuss the experimental setup and reproducibility considerations.

4.1 Environments

To introduce the conducted experiments, we build on the general discussion of the Minihack environment provided in Section 2.1.1, focusing on the observations available to the agent, including potential preprocessing steps, the action space, and the reward structure. This expansion is detailed in Section 4.1.1. Subsequently, in Section 4.1.2, we address the Ant environment.

4.1.1 Minihack

As mentioned, our experiments focus on the MiniHack environments "*MiniHack Room Random* and *MiniHack Room Monster*. Importantly, we utilize different room sizes. Most experiments are conducted on a $10x10$ grid size, and only a limited amount of evaluations are conducted on the much harder $15x15$ grid.

MiniHack Room Random In the randomized version of the MiniHack Room, both the start position and destination are randomized. Therefore, the agent must learn to navigate within the grid by understanding its location relative to the destination and how to move toward it. The agent receives a reward of $+1$ upon reaching the goal

position and -0.01 if the agent tries to leave the room by running into a wall. This results in a reward structure limited to $[-T \cdot 0.01, 1]$ with T as the maximum episode length, which differs by environment. For 10×10 grids, episodes are terminated after 200 steps; for 15×15 grids, episodes are terminated after 300 steps.

MiniHack Room Monster In the monster version of the environment, *MiniHack Room Random* is extended by introducing a monster to the grid. It is randomly selected from 560 different Monsters in Nethack (Küttler et al., 2020; Samvelyan et al., 2021). They differ in their behavior from peaceful to hostile and in how they interact with the agent. Importantly, if a monster kills the agent, the episode terminates; however, no negative reward is assigned. Instead, the previously introduced reward structure is applied here as well.

Observation Handling MiniHack provides different representations of the current state of the game. In this thesis, we utilize the representation called $chars$, encoding the 21×79 grid into a vector of shape 21×79, of values $0 - 64$. If the playing field is smaller than the grid, then the empty fields are characterized as 32. Figure 4.1 provides an example, and Figure 4.2 shows the accompanying render.

```
..........................................................................................
... 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 ...
... 32 64 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 32 ...
... 32 46 46 46 46 46 46 46 46 46 46 46 46 46 46 62 32 ...
... 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 ...
..........................................................................................
```

Figure 4.1.: Representation provided to the agent as an array. **32** indicates a field not used in the current environment. **64** indicates the position of the agent. **62** indicates the goal position. **46** indicates a field the agent can walk on.

However, "empty," "goal," and "agent" positions are wrongfully encoded on an ordinal scale (Samvelyan et al., 2021). We, therefore, first one-hot encode each char and then use one convolution layer on each resulting encoding for a better

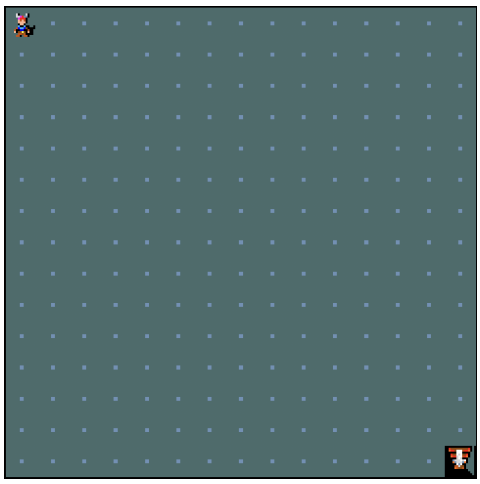


Figure 4.2.: Image of a game status. Note that empty content, normally indicated by black pixels, is cut off in the figure, while the agent is still presented with it.

representation. The resulting vector is then downscaled to shape 512. For simplicity, we do not consider this preprocessing as be part of the feature extractor.

Action Space The MiniHack environment contains all actions allowed in Nethack. However, in our experiments, we reduce the space of possible actions to control the agent's movement inside the environment, resulting in 8 different actions moving towards all directions, including diagonally.

4.1.2 Ant

The Ant environment, part of the Mujoco RL environments (Schulman et al., 2016), is designed to train agents to control the behavior of a small ant robot within the Mujoco physics simulation (Todorov et al., 2012). An image of the Ant is shown in Figure 4.3.

During training, the agent receives the current position, the joint angles, and the joint velocities. Every leg of the ant has two joints. Observations are then given as a vector of dimension 27 and are tasked with adjusting the joints by applying torque. The goal is for the agent to control the Ant's movement so that it moves forward;

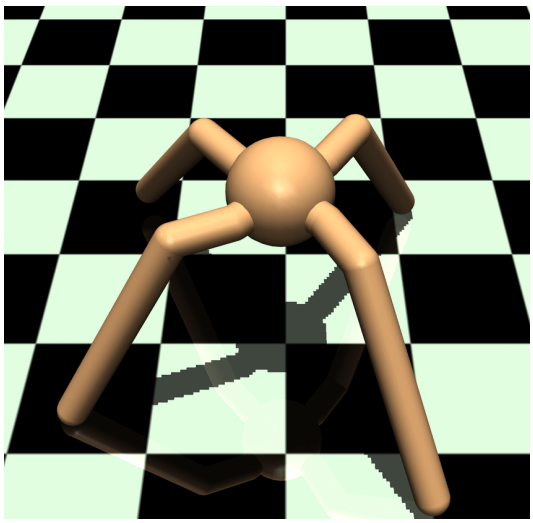


Figure 4.3.: Visual representation of the Mujoco Ant environment.

however, it may also receive negative rewards if it moves to the sides or uses very large actions. Overall, the reward is calculated as

$$R_{Ant}(s) = R_{Healthy}(s) + R_{Forward}(s) + R_{Control}(s) \tag{4.1}$$

with $R_{Healthy}(s) = 1$ if the agent is healthy, and $R_{Healthy}(s) = 0$ if the agent is unhealthy (torso outside of predefined interval or state space values not finite). The term

$$R_{Control}(s) = -0.5 \cdot \sum_{a_i \in a} a_i^2 \tag{4.2}$$

penalizes large actions. An episode ends if the agent is unhealthy or after $1,000$ timesteps. This means that if the ant does not move, it generates a positive reward of 1.000.

4.2 Experimental Setup

To evaluate the effectiveness of the proposed approach, we conduct a series of experiments comparing policies from incrementally grown networks and static-sized networks. We introduce the different experiments per growing approach and baseline in Section 4.2.1. Following this, we outline the setup of SMAC in 4.2.2, and

our general evaluation setup in Section 4.2.3. Lastly, we describe our reproducibility decisions in Section 4.2.4.

4.2.1 Baselines

To assess the performance of agents trained with our multi-fidelity optimization and feature extractor growth approach, we compare both their final results and training processes to those of baselines. To that end, we utilize standard PPO agents with hyperparameters optimized with SMAC's Black Box Optimization Facade (Lindauer et al., 2022). The baseline agents are configured to be the size of the networks of the size at which we arrive after network growth. We additionally add baseline of depth one for additional context.

Our evaluation focuses on the returns generated by the different agents on evaluation episodes. This means that to evaluate the agent's quality, we utilize separate environments. One assesses the agent's training process, and one assesses the final performance. For each evaluation, we utilize 10 evaluation episodes.

4.2.2 SMAC Setup

To provide a reasonable evaluation, it is essential that both the Proximal Policy Optimization (PPO) training and the trial quota of the Hyperparameter Optimization (HPO) tool facilitate a fair comparison. We consider two main factors:
First, the amount of samples given to PPO needs to be configured fairly. In the above example, a network incrementally grown to four layers, with 500,000 steps allotted per PPO loop, undergoes training for 2,000,000 steps. Accordingly, the baseline networks are also provided with a budget of 2,000,000 steps.
Second, the hyperparameter configuration tool's trial quota is set to ensure that the number of environment interactions for the baseline is equal to or greater than that for the incrementally growing approach. As a result, the growing approach—where PPO trains for 500,000 steps per evaluation—is allowed less than or equal to four times the number of PPO executions compared to the baseline, which trains for 2,000,000 steps.

The Budget assigned to the different approaches is 100 Million for environmental interactions. For the baselines, this is represented by 50 iterations of 2 Million environment interactions each. For our network growth approach, the amount of trials is determined by the amount of network growth steps, as explained above.

Hyperparameter Search Space To reduce computational complexity, our experiments use a reduced number of PPO hyperparameters, both for the baselines and the incremental growth approach. An overview of the hyperparameters tuned is provided in Table 4.1.

hyperparameter-name	type	choices
entropy coefficient	uniform float	[0.0, 0.3]
learning rate	uniform float	[0.0001, 0.01]
number of epochs	uniform int	{5, 6, ..., 20}
batch size	categorical	{32, 64, 128, 256}

Table 4.1.: List of hyperparameters used to configure both the Baselines and the Growing Neural Network Approaches.

All other hyperparameters are fixed, either to the default values provided by Stable-Baselines3 (Raffin et al., 2021) for MiniHack and well-performing defaults reported by Raffin, 2020 for Ant.

4.2.3 Evaluation Setup

All of our experiments were executed on an HPC using a combination of SMAC (Lindauer et al., 2022) and the hypersweeper (T. Eimer, 2024). We run a main job configured with 8 CPU cores and 8 GB RAM, which repeatedly schedules 5 worker jobs, in which the PPO is used to train agents on MiniHack. The workers utilize an $H100$ GPU, 16 cores, and 16 GB RAM. Inside these jobs, we utilize the PPO implementation of stable-baselines3 (Raffin et al., 2021).
Our results are logged to a SQL database using the PyExperimenter library (Tornede et al., 2023).

4.2.4 Reproducibility

To ensure that all of our evaluations are reproducible, we seed every component and even adapted the codebase of MiniHack and NetHack.
Concretely, this means the following: SMAC is seeded identically for all experiments, which results in SMAC always sampling the same hyperparameter configurations initially.
To evaluate the quality of a sampled configuration, we utilize five workers whose networks are initialized with seeds $0, 1, 2, 3, 4$, and whose environments are seeded with the same training, progress validation, and final validation seeds. This allows for assessing the extent to which the random network initialization leads to differing behavior in the same environment.

5 Evaluation Results

This thesis aims to answer the following three research questions:

1. How can network growing approaches (prominent in other Machine Learning areas) be employed in RL?

2. In what way does the learning process change upon network adaptation?

3. When moving from simpler to more difficult problems, does iteratively increasing the depth or width of a pre-trained neural network yield improved results compared to starting with the final/initial network size?

The first research question, concerning the application of network growth approaches in reinforcement learning (RL), is addressed primarily through theoretical discussions presented throughout the thesis and through a comparison between agents trained with a growing policy network and those with a static configuration. The second and third research questions are explicitly addressed by the following experiments: Section 5.1 focuses on the second research questions, and Section 5.2 focuses on the third questions.

5.1 Static Environments

We conducted a set of experiments to assess the impact of our proposed network growth methodology on the learning process. Given a static environment, before and after growth, we utilize Net2DeeperNet and Net2WiderNet to grow the feature extractor. This aims to answer the second research question, on the impact of network growth on the training process, and the third research question comparing Net2WiderNet and Net2DeeperNet (T. Chen et al., 2016).
Our results on the MiniHack environment are detailed in Section 5.1.1, and our additional results on applying Net2DeeperNet on Ant are provided in Section 5.1.3.

5.1.1 MiniHack

Our evaluation of MiniHack is split into three parts: We first explore the results of our experiments on Net2DeeperNet and Net2WiderNet separately. Building on these findings, we then present our conclusions.

Net2DeeperNet

For Net2DeeperNet, we start with a depth one feature extractor and progressively expand it to two or four layers. Each layer is 512 neurons wide. As baselines, we use static feature extractors of depth one, two, and $four$.

Optimization Process The evaluation results of the incremental feature extractor growth combined with Multi-Fidelity Optimization are shown in Figure 5.1 for MiniHack Room Random and in Figure 5.2 for MiniHack Room Monster. These figures visualize the optimization of the different approaches for MiniHack Room Random and MiniHack Room Monster and compare the behavior of all evaluated strategies and baselines. The amount of environment interactions is shown on the x-axis, and the average return of the five workers with their respective seeds on the y-axis.

The optimization process of MiniHack Room Random is visualized in Figure 5.1. At the beginning of the optimization process, the agents show differing performance. After the first trial is evaluated, the depth one agents, and network growing approaches outpefrom the other baselines.
After 20 million interactions, the static approaches of depths two and four perform poorly compared to the network growth approaches and the agent configured with a static depth of 1. While the poorly performing agents continue discovering incumbents, they only marginally improve the cost and do not manage to reach a cost of 0. This means that the depth two and depth four baselines cannot reliably avoid negative returns.
In contrast, the depth 1 baseline and grown networks improve over a cost of 0 after less than 20 million environment interactions. They perform similarly and hit a plateau at about 40 million interactions with a cost smaller than -0.5. This implies they solve the problem for at least 50% of the instances. From this plateau on, only minor gains are being made, and the baseline with a static depth of 1 masters the

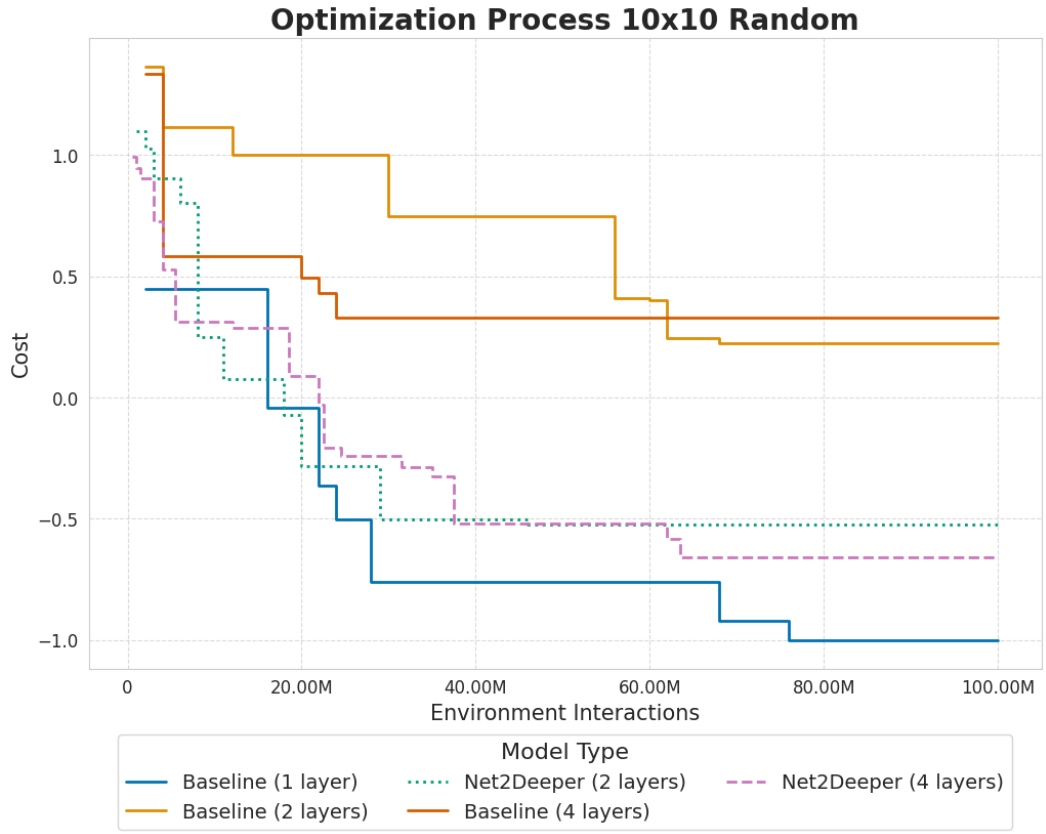


Figure 5.1.: Optimization Process of incremental growth with Net2DeeperNet and static baselines on Minihack Room Monster 10×10.

environment at about 80 million interactions.
At the end of the optimization process, the static baselines of depth 1 outperform all other approaches, and the incrementally grown approaches beat their respective static configuration. When comparing network growth strategies, the feature extractor grown to a depth of four outperforms the one grown to a depth of two.

On MiniHack Room Monster, visualized in Figure 5.2, the results are quite similar; however, the static baselines of depths two and four continue their performance gains after 20 million interactions. Between 40 million and 50 million environment interactions, all approaches manage to improve over a cost of 0. At about 80 million interactions, the static baseline of depth 1 again masters the environment.
At the end of the optimization process, incremental growth to depth two outperforms its baseline, while incremental growth to depth four does not.

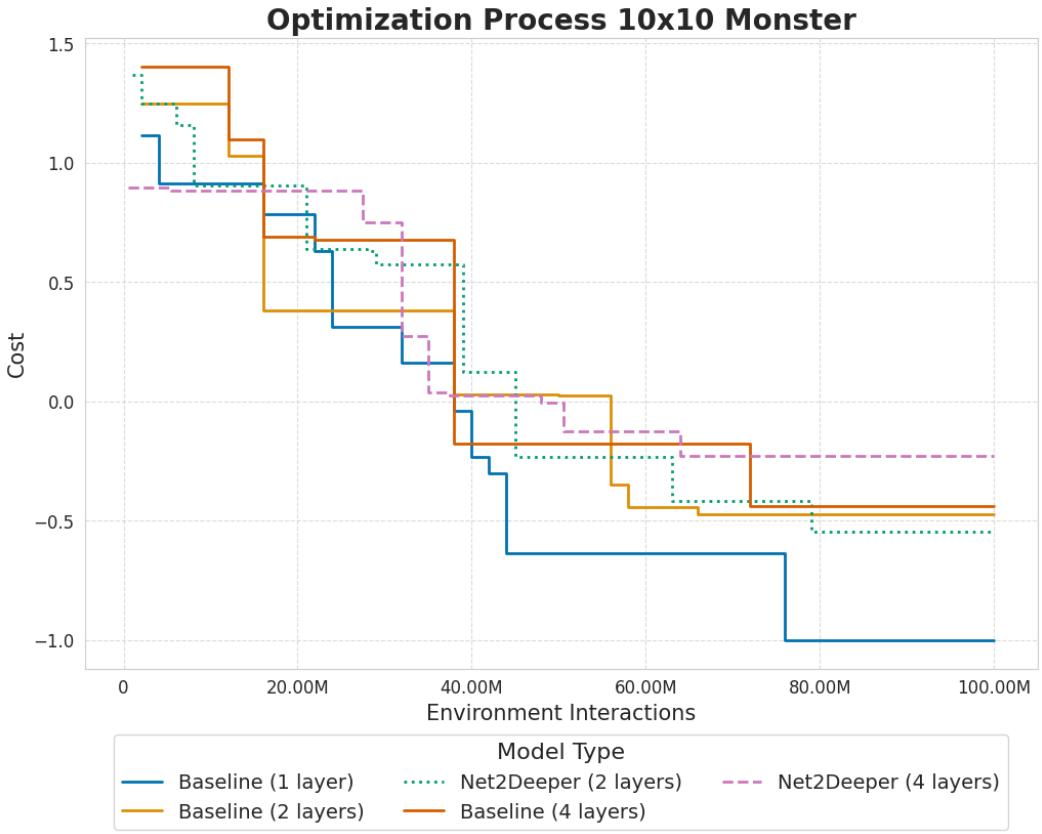


Figure 5.2.: Optimization Process of incremental growth with Net2DeeperNet and static baselines on Minihack Room Monster 10×10.

Additionally, we evaluated how well agents' performance correlates between the different budgets using DeepCave (Sass et al., 2022) as shown in Appendix A.1.1 The results show that the lower fidelities serve as good predictors for the higher fidelities on MiniHack.

In the final incumbent's hyperparameter configurations, shown in Appendix A.1.2, the only visible trend is that all network growing approaches select 32 as batch size.

Incumbent Training Process To answer how network growth impacts the agent's learning behavior, we also compared the training process of the different incumbents yielded by the results presented above.

In the following figures, the amount of environment interactions is plotted on the x-axis, and the mean of the workers' interquartile mean evaluation return is plotted on the y-axis. The uncertainty intervals contain the 95% confidence intervals of the interquartile means. A black vertical line indicates the feature extractor growth.

The resulting plots, portraying the achieved returns over the training process on **MiniHack Room Random** 10×10, are shown in Figures 5.3 and in Appendix A.1.3 for depth four.

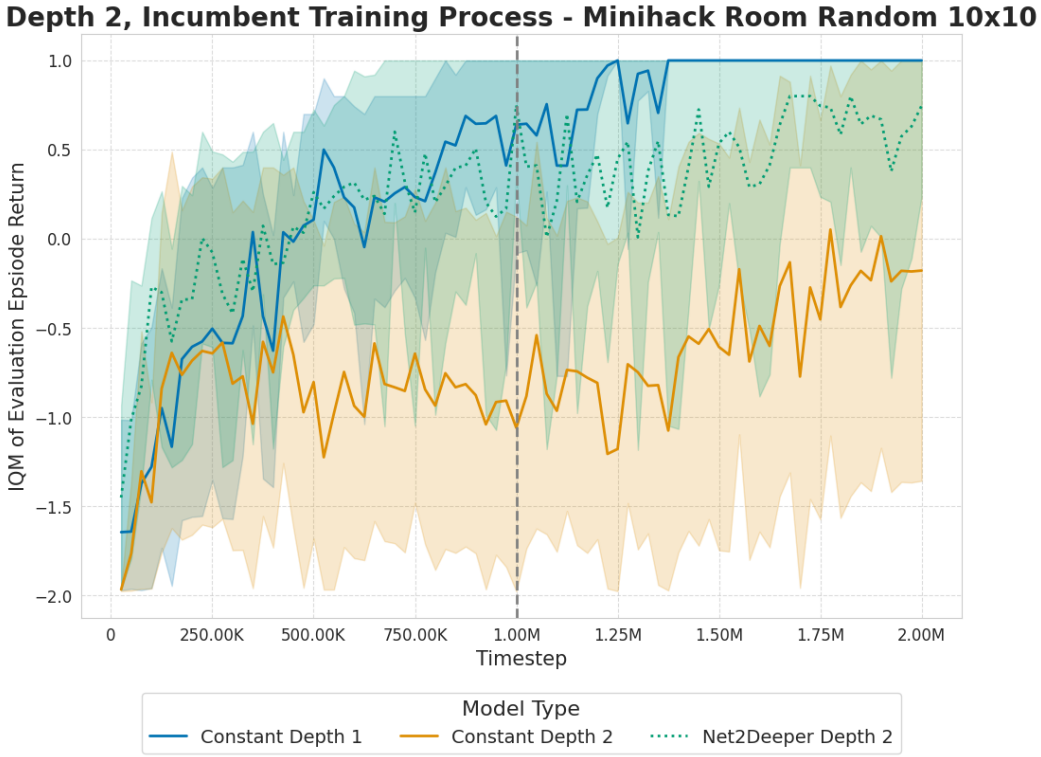


Figure 5.3.: Training Process of the incumbent trained with Net2DeeperNet with a depth of two on Minihack Room Random 10×10. Baselines include constant depths one and two.

For two layers, our results show that all approaches perform similarly well for the first 250,000 steps. At 500,000 timesteps, agents equipped with a feature extractor of depth one, either due to static configuration or because the feature extractor is not grown yet, improve over an average return of 0. They, therefore, beat the baseline configured with a static depth of two, currently yielding on average a negative return of -1.
At $1,000,000$ timesteps, Net2Deeper is used to grow the feature extractor from one to two layers, which does not cause a drop in performance. The grown agent continues

to perform similarly for the rest of the training. However, the static one-layer feature extractor agents continue improving, solving the environment consistently after less than 1.5 million interactions.
Compared to both approaches, the baseline equipped with a static depth of two continues performing poorly for the rest of the training. While the agent improves somewhat, it cannot avoid negative returns on average. However, its rather large uncertainty intervals show that for some seeds, the agent appears to understand the environment and can even obtain positive returns quite often.
Importantly, the confidence intervals, which are calculated based on five workers overlap for most the training process. This is caused by a large performance difference between the workers, based on the network initalisation.

For depth four, we obtained very similar results, which can be seen in Appendix A.1.3. The main difference is that upon feature extractor growth to three, the Net2DeeperNet agent appears to suffer from a drop in performance, from which it soon recovers. Also, after growth the incumbent continues improving.

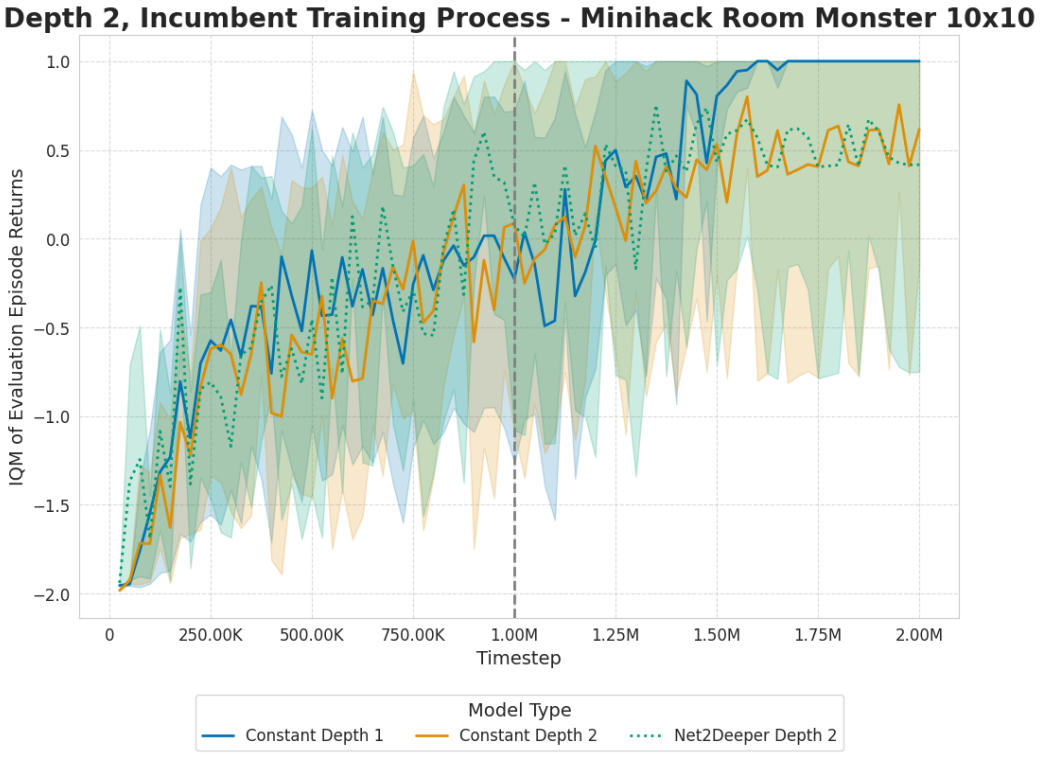


Figure 5.4.: Training Process of the incumbent trained with Net2DeeperNet with a depth of two on Minihack Room Monster 10×10. Baselines include constant depths one and two.

The results for **MiniHack Monster** 10×10 can be seen in the following figures: Figure 5.4 evaluates the training performance of the depth two feature extractors on MiniHack Room Monster. In contrast to the random environment, the 1-layer feature extractor baseline reaches the maximum return of 1 later in the optimization process.
The main difference between the two-layer feature extractors is that both the agent equipped with a grown and the agent with a static policy network perform similarly well and reach an evaluation performance of approximately 0.5 by the end of the training process. This means that the baseline performs very well compared to the results on Room Random, while the grown feature extractor performs similarly to before.

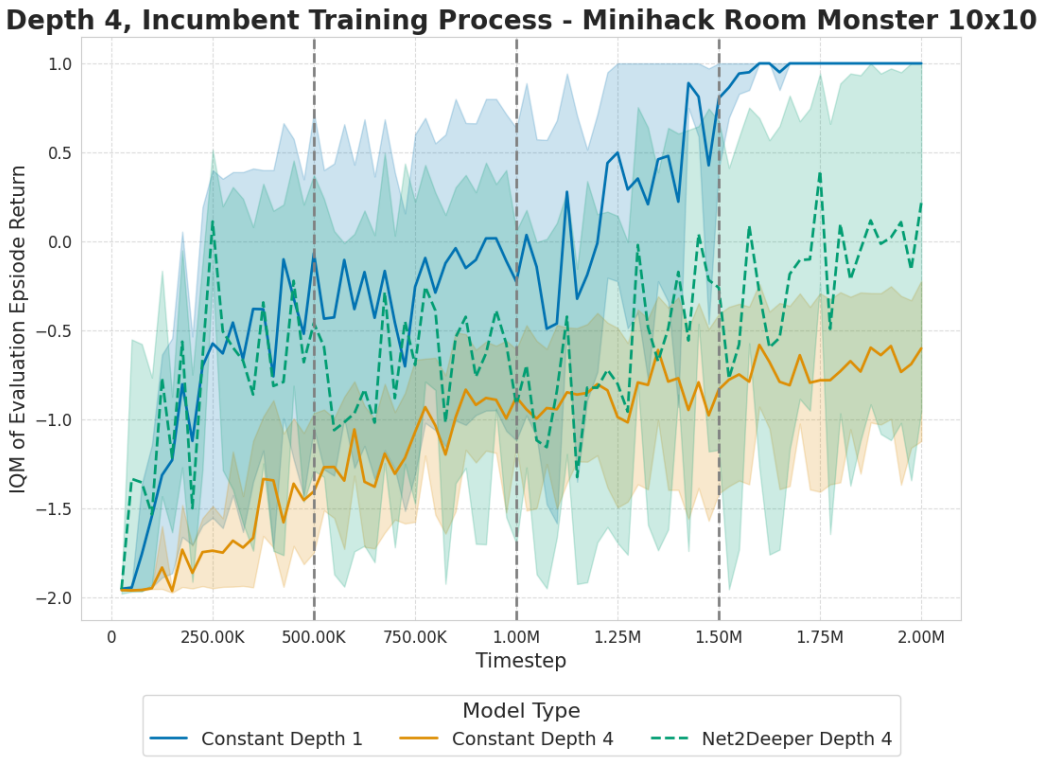


Figure 5.5.: Training Process of the incumbent trained with Net2DeeperNet with a depth of two on Minihack Room Monster 10×10. Baselines include constant depths one and four.

Figure 5.5 evaluates the training performance of the depth four feature extractors on MiniHack Room Monster. Again, the results are similar to the results on MiniHack Room Random, with the agent trained with four layers for the entire training process being outperformed by the incrementally grown agent, which is

outperformed by the agent equipped with a static one-layer feature extractor.

We also explored why baseline agents of depth 2 and depth 4 perform better if a monster is present. Our initial belief that the agents might learn to die to avoid negative rewards appears false, as shown in Appendix A.1.3. However, due to the monsters, many episodes are terminated earlier for suboptimal policies, such as running into a wall. Due to this early termination, the agent is trained on more episodes, which might help the learning process.

Network Weights To further enhance our understanding of the effect Net2DeeperNet has on the learning process, we also analyze the weights learned by the feature extractor. A subset of the weight matrix from the 4-layer network grown by the first worker of incumbent on MiniHack Room Random is visualized in Figure 5.6. Each pixel represents the size of a weight utilized internally by the network. To simplify the representation, we consider only a 50×50 subset of weights.

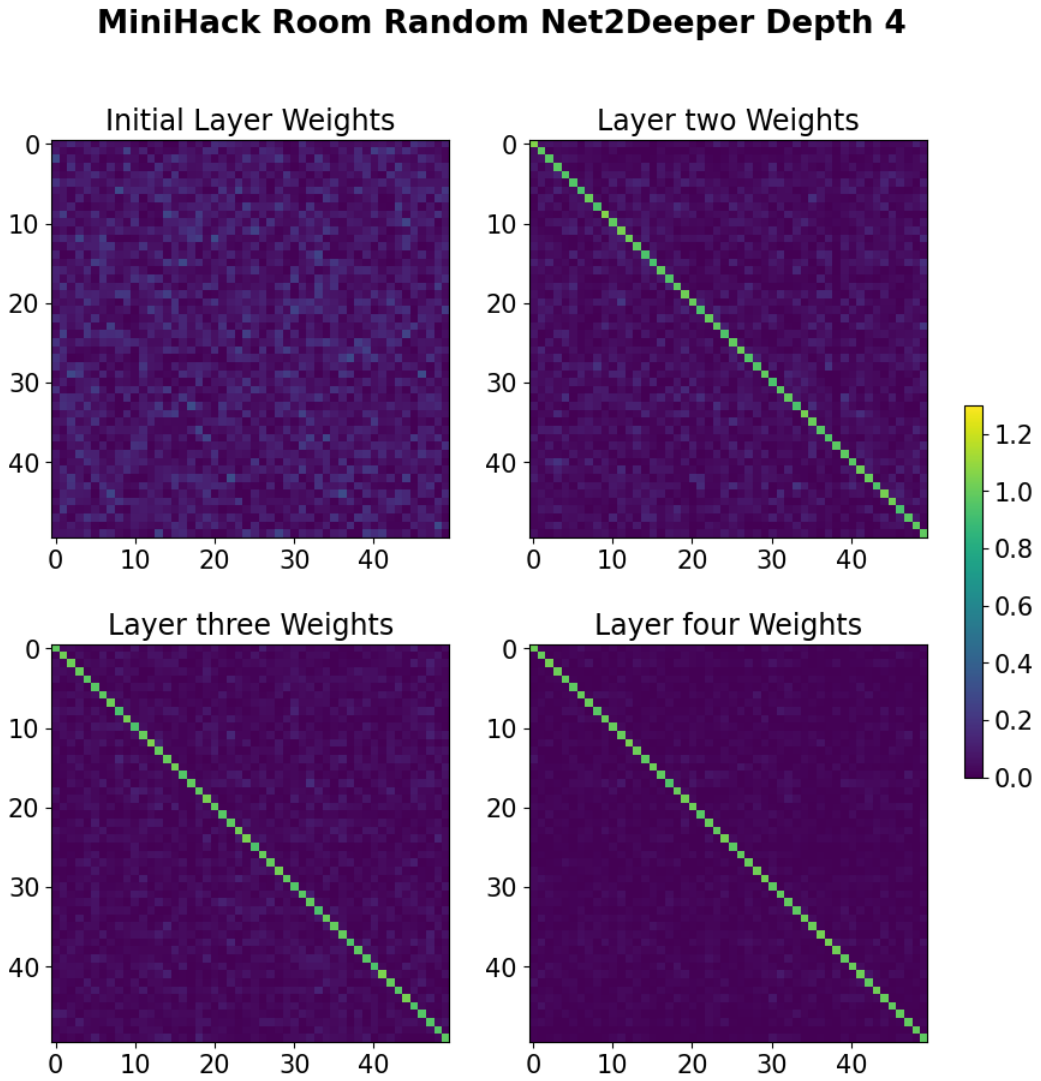


Figure 5.6.: Visualization of a 50×50 subset of weights learned inside the feature extractor.

As shown in the figure, the weights learned in the initial layer differ significantly from those in the subsequent layers added during network growth. In these later layers, the original identity weights are still visible, while the other weights, initialized to zero, remain relatively small compared to the initial layer. As the network depth increases, this disparity grows. This could imply that each additional layer refines the representation from the previous one. However, it may also be a byproduct of the learning process, as the earlier layers have experienced more timesteps. This trend is consistent across other learning scenarios, as shown in Appendix A.5.1.

Interpretation From the results above, one can conclude: Firstly, in the explored MiniHack Room environments, a one-layer feature extractor performs superior to the other approaches. However, incrementally growing the feature extractor with the proposed methodology results in improved agents, i.e., the proposed procedure can help make deeper feature extractors usable. We attribute it to the network morphisms not resulting in the agent forgetting the previously learned representation, as indicated in all evaluations. Additionally, network morphisms appear to not hinder the learning process if the agent does not show an almost perfect policy already. These results are further backed by the network weights, which show that the added layers only marginally differ from their initialization as identity.

Net2WiderNet

The combination of the above results shows that shallow feature extractors perform better on MiniHack Room and Net2WiderNets' requirement of at least two layers; we decided to conduct our experiments using a two-layer feature extractor. Despite maintaining constant input and output dimensions of 512, the internal layer width is expanded once or three times, disregarding the feature extractor's input and output neurons. The baselines for this method are scenarios where the feature extractor remains a static structure achieved after expansion.

Optimization Process The evaluation process results for MiniHack 10×10 Random and 10×10 Monster can be found below. The plots are structured as in the Net2DeeperNet evaluation above.

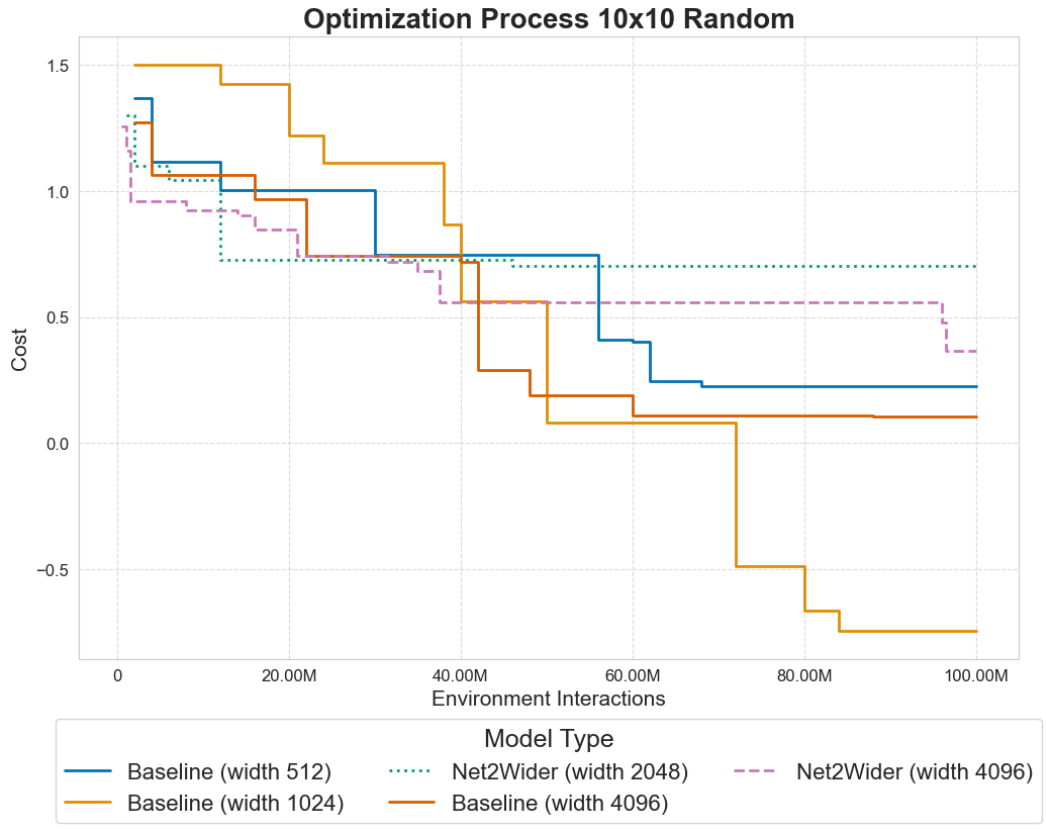


Figure 5.7.: Optimization Process of incremental growth with Net2WiderNet and static baselines on Minihack Room Random 10×10.

For 10×10 Random, shown in Figure 5.7, the different approaches perform similarly until 60 million environment interactions. After this threshold, the optimization process largely stagnates, with only the width 1024 feature extractor baseline progressing. Importantly, only this baseline reliably passes the threshold of cost 0, meaning that all other approaches do not reliably avoid negative returns. After 100 million environment interactions, the width 1024 feature extractor baseline outperforms all other approaches, and the static feature extractor baselines outperform the network growing approaches.

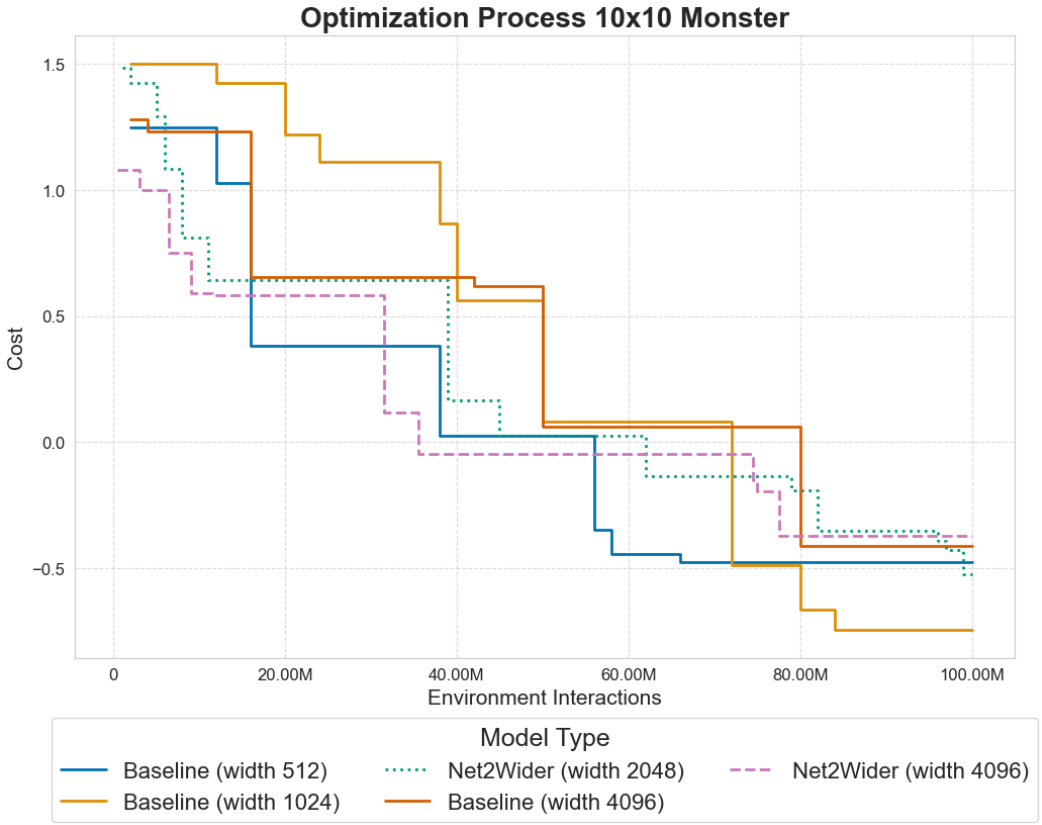


Figure 5.8.: Optimization Process of incremental growth with Net2WiderNet and static baselines on Minihack Room Monster 10×10.

In contrast, for the Monster environment, shown in Figure 5.8, all approaches continue optimization even after 60 million steps and manage to collect positive returns, meaning the growing approaches largely stay competitive while not outperforming the baselines. However, the approach using a static width 1024 feature extractor outperforms the other approaches, and the baselines outperform the network growing approaches.

Additionally, we also evaluated the budget correlations for Net2WiderNet, shown in Appendix A.2.1. The rank correlations show that the correlations of Net2WiderNet are much smaller than those of Net2DeeperNet on MiniHack Room Random and slightly smaller on MiniHack Room Monster.

An analysis of the incumbent's hyperparameter configurations, shown in Appendix A.2.2, shows no clear trend.

Incumbent Training Process Figures 5.9 and 5.10 show the incumbent training performance on MiniHack Room Random. As shown in the figures, the agents utilizing a grown feature extractor consistently perform poorly. Additionally, they are outperformed by the static approaches. At no point in their optimization did the incumbents consistently manage to improve over a return of 0.

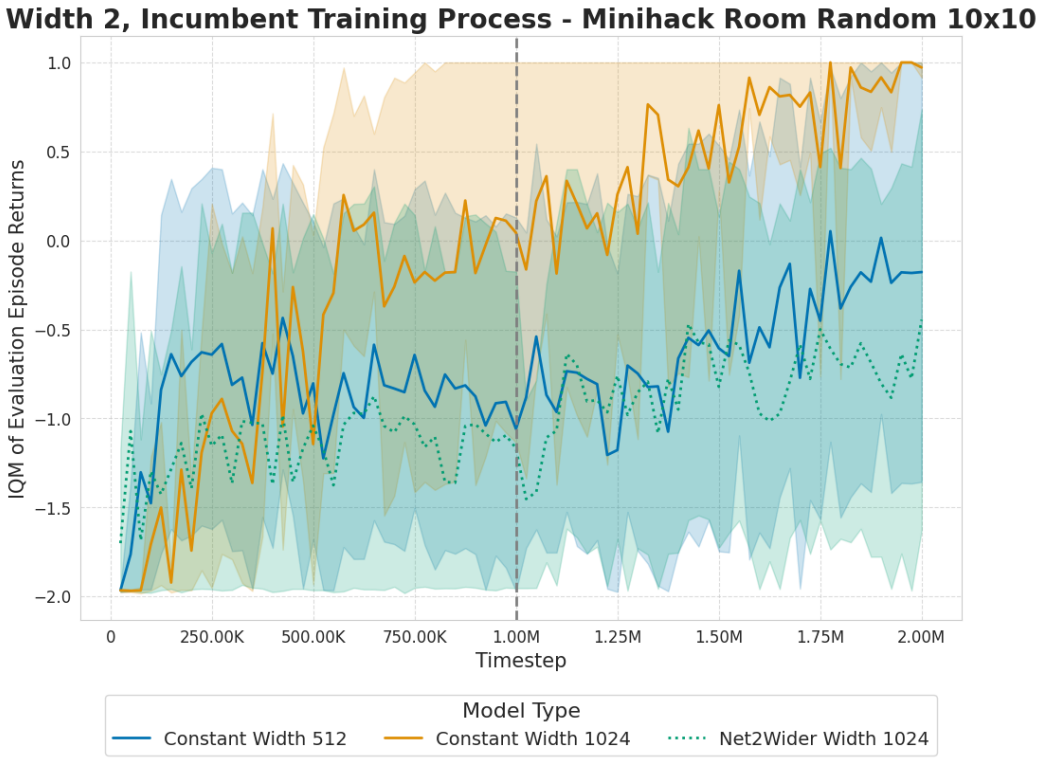


Figure 5.9.: Training Process of the incumbent trained with Net2WiderNet grown once on Minihack Room Monster 10×10. Baselines include constant depths one and two.

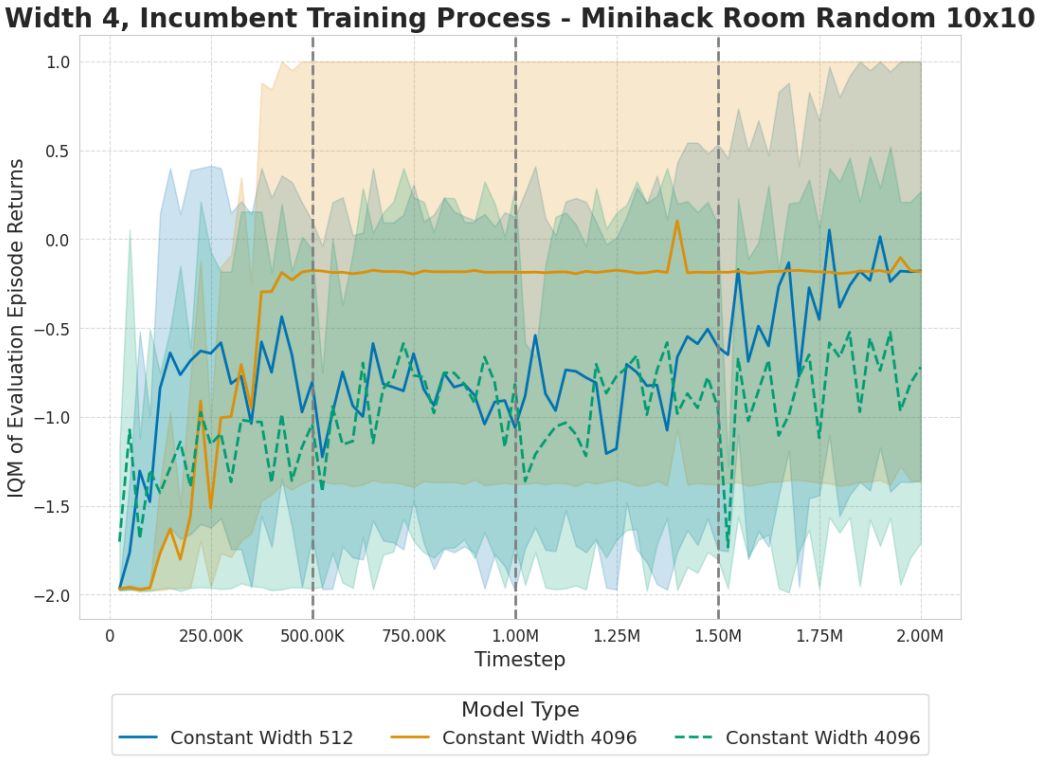


Figure 5.10.: Training Process of the incumbent trained with Net2WiderNet grown once on Minihack Room Monster 10×10. Baselines include constant depths one and two.

In contrast, on MiniHack Room Monster, the network growing approaches remain competitive with their respective baselines.
In Figure 5.11, we plot the incumbent's training performance of the growing approach and baselines on MiniHack Room Monster 10×10. The plots portray a competitive learning behavior for all approaches, with the constant width of 512 outperforming the wider approaches that master the environment. Importantly, the growing approach sees a major drop in performance after applying Net2WiderNet. It recovers and is competitive with the other approaches again after 250,000 timesteps.

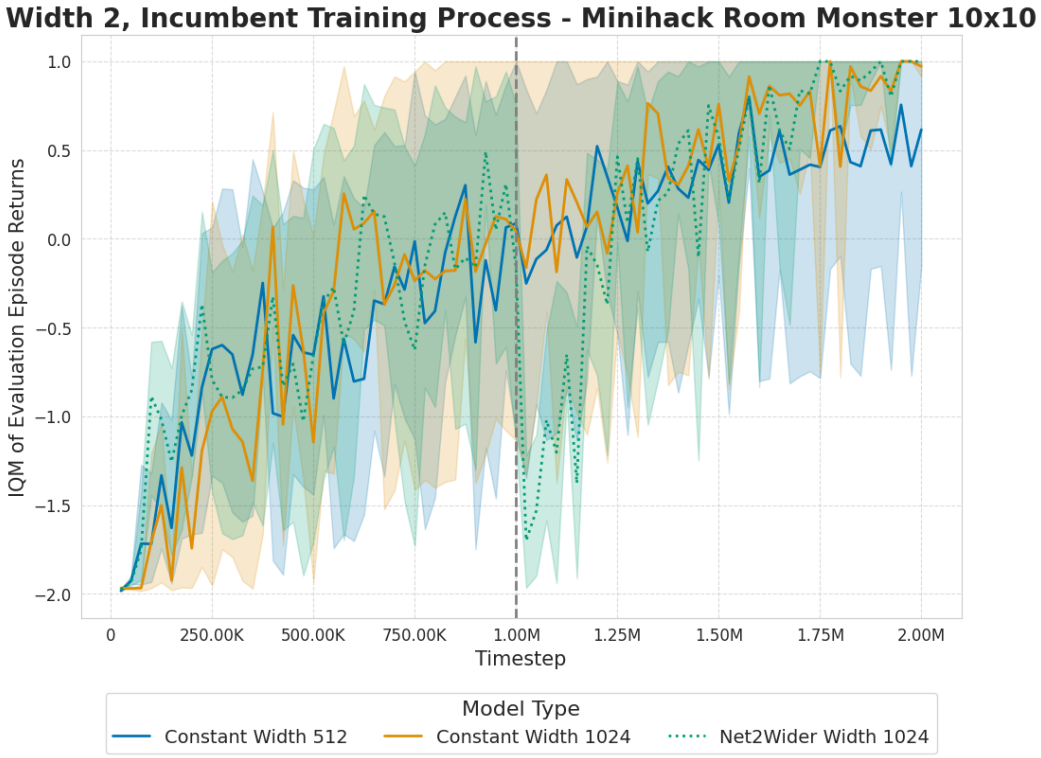


Figure 5.11.: Training Process of the incumbent trained with Net2WiderNet grown once on Minihack Room Monster 10×10. Baselines include constant depths one and two.

Similarly, the learning process of the width 4096 feature extractors portrayed in Figure 5.12 shows a very similar training performance between incremental growth and utilizing the wide network statically. An analysis of the worker results shows that the learning behavior differs between the different workers.

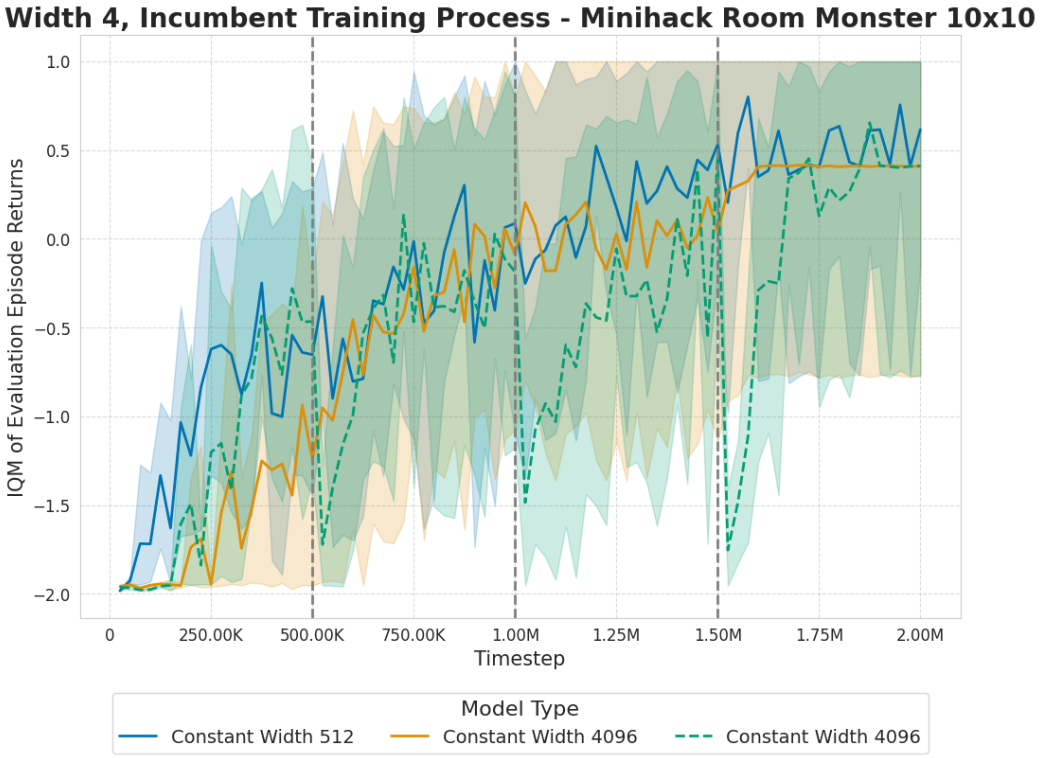


Figure 5.12.: Training Process of the incumbent trained with Net2WiderNet grown three times on Minihack Room Monster 10×10. Baselines include constant depths one and two.

Network Weights As above, the weights of the four times grown feature extractor on MiniHack Room Random are shown in Figure 5.13.

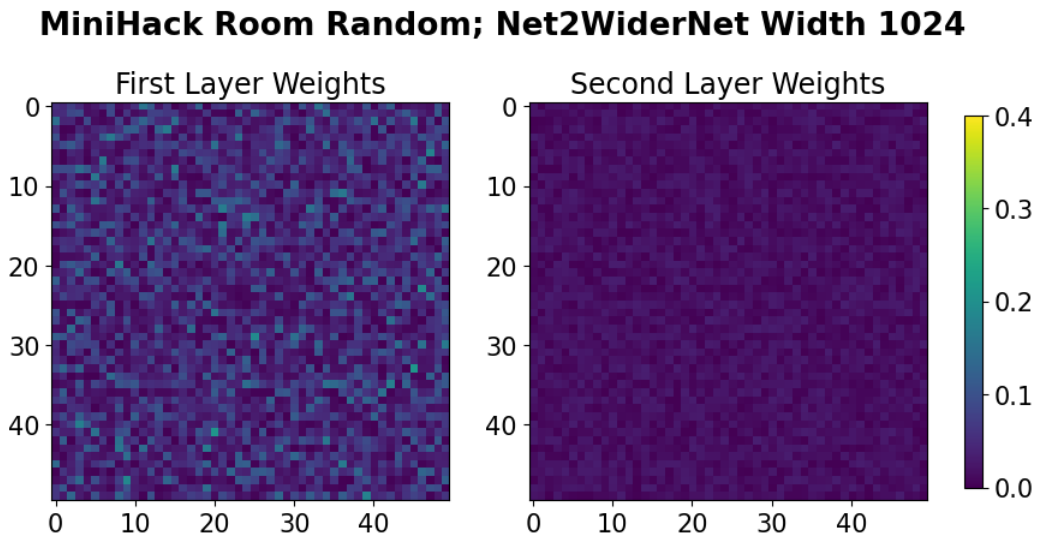


Figure 5.13.: Visualization of a 50×50 subset of the weights learned inside the feature extractor.

As shown in the Figure, the weights learned in both layers differ in scale. The first layer appears to learn higher weights than the second layer. This result does not offer any relevant insights. The weights learned by the other incumbents can be seen in Appendix A.5.2.

Interpretation While the results indicate that using a width greater than 512 can be an improvement, applying Net2WiderNet on MiniHack Room Random hinders the training process since agents utilizing a grown feature extractor do not manage to beat their respective baselines. We attribute this to the considerable drop in performance upon network growth and the lower budget corrleations.
In contrast, in the easier MiniHack Room Monster environment, the results show that growing the feature extractor does not hinder the training process, but it also does not help it. The network weights offer no additional insights.

5.1.2 Conclusion

The main conclusions that can be drawn from the above experiments are three-fold. Firstly, our experiments indicate that for MiniHack Room Random and MiniHack

Room Monster agents configured with a depth, one feature extractor appears to outperform their competitors. Secondly, while agents configured with Net2DeeperNet do not perform as well as depth one agent, the network growth combined with MultiFidelity appears to help them, compared to agents with static feature extractors. Lastly, Net2WiderNet appears to hinder the training process.

5.1.3 Ant

Based on the results on MiniHack Room, where the one-layer feature extractor masters the environments, this section explores whether this effect persists in the Ant environment. To that end we evaluate Net2DeeperNet on the Ant environment. Based on a preceding execution of the static baselines, we consider iteratively growing the feature extractor to depths four and eight in these experiments.

Optimization Process To expand the previous evaluation of feature extractor growth in MiniHack, we decided on additional evaluations on the Ant environment, for which we expected that agents with a feature extractor of depth one do not beat deeper agents. The experiment, which compares the performance of an incrementally grown feature extractor with a static baseline, can be seen in Figure 5.14.

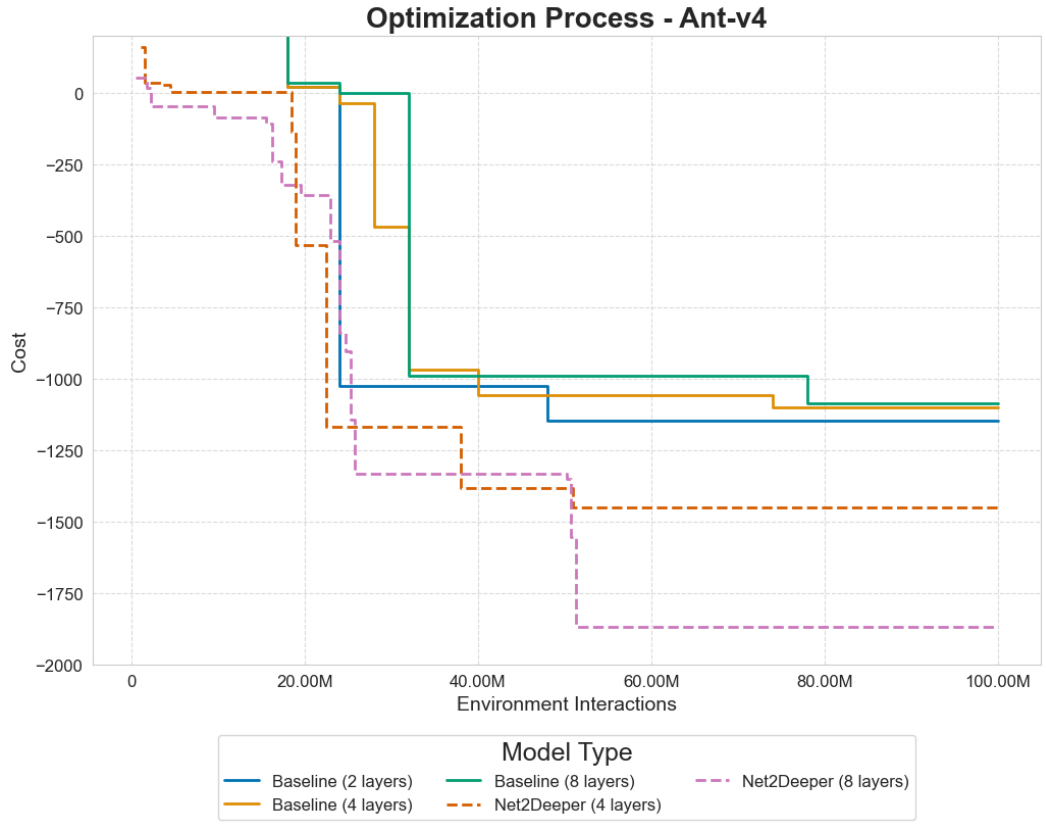


Figure 5.14.: Optimization Process of incremental growth with Net2WiderNet and static baselines on the Ant environment.

At the beginning of the optimization process, when random configurations are sampled, none of the approaches show proper learning behavior. At approximately 30 Million steps, most approaches find incumbents that perform better than the cost of -1000, which can be generated by avoiding death by not acting. From here on, the static baselines only find incumbents with minor performance boosts, while network growth approaches improve significantly. Ultimately, the network growth approaches outperform the static approaches significantly.

The budget correlations of both growing to depth four and growing to depth 8 look quite similar to previous results, with higher budget correlations for the approach associated with better performance. They can be seen in Appendix A.3.1.

In our analysis of hyperparameter configurations, shown in Appendix A.3.2, we find that in the Ant environment, both baselines and approaches select almost identical learning rates around $0,0001$.

Incumbent Training Process Figure 5.15 plots the agent statically configured with depth four against the incrementally grown agent.

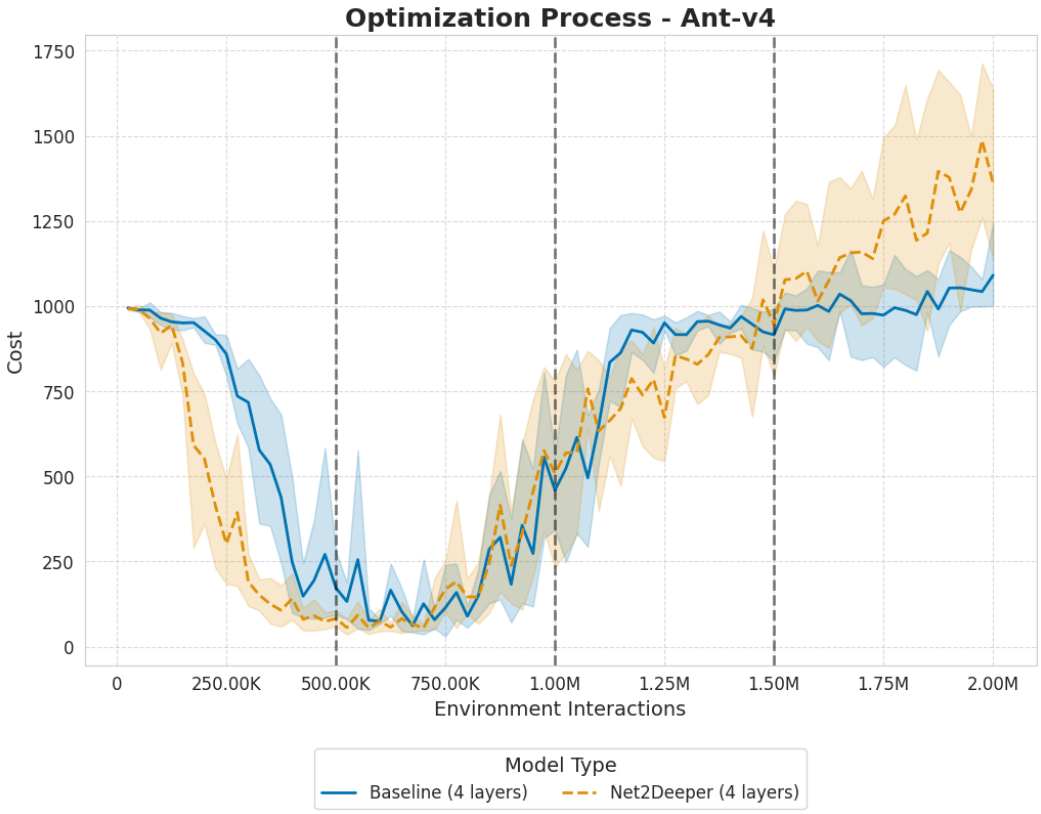


Figure 5.15.: Comparison of the incumbent configuration's training process on the Ant environment. The plot contains the agent configured with feature extractors of depth four.

Both approaches start at a return of approximately 1000. Then they drop in performance to a return of approximately 100. At approximately $750,000$ interactions, the agents seem to pick up training, and at $1,5$ Million environment interactions, they reached the initial return of $1,000$. From here on, both approaches appear to continue learning, with the network-growing approach vastly outperforming the approach with static configuration.

In Figure 5.16, we compare the two approaches for depth 8. The axes are shown as in the previous training process plots.

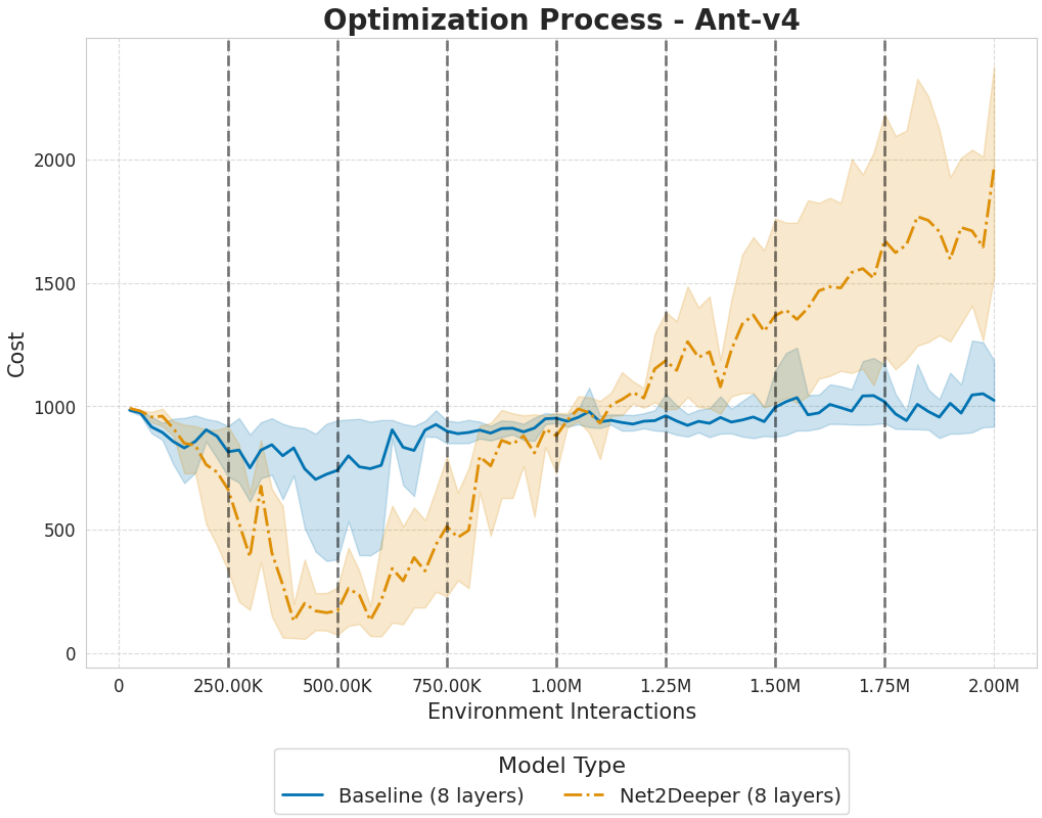


Figure 5.16.: Comparison of the incumbent configuration's training process on the Ant environment. The plot contains the agent configured with feature extractors of depth eight.

For depth eight, the learning process of the growing approach is quite similar. However, the agent recovered from the initial drop in performance faster and achieved a greater return in the end. In contrast the baseline has a less extensive performance drop, from which it also soon recovered. However, the agent barely improves over an average return of 1.000 on average.

We hypothesize that the drop in performance at the beginning of the training process appears to result from the reward function, introduced in 4.1 used in the Ant environment, which penalizes large actions heavily. This is explored in Figure 5.17. The plot compares the training process of the agent incrementally grown to depth 8, with the size of actions. As can be seen in the plot, the initial drop in performance is accompanied by an increasing action size, and when the agent picks up training, the action size appears to decrease. At approximately $1,000,000$ timesteps, the action

size is perceived to slowly increase again, accompanied by an improved evaluation performance.

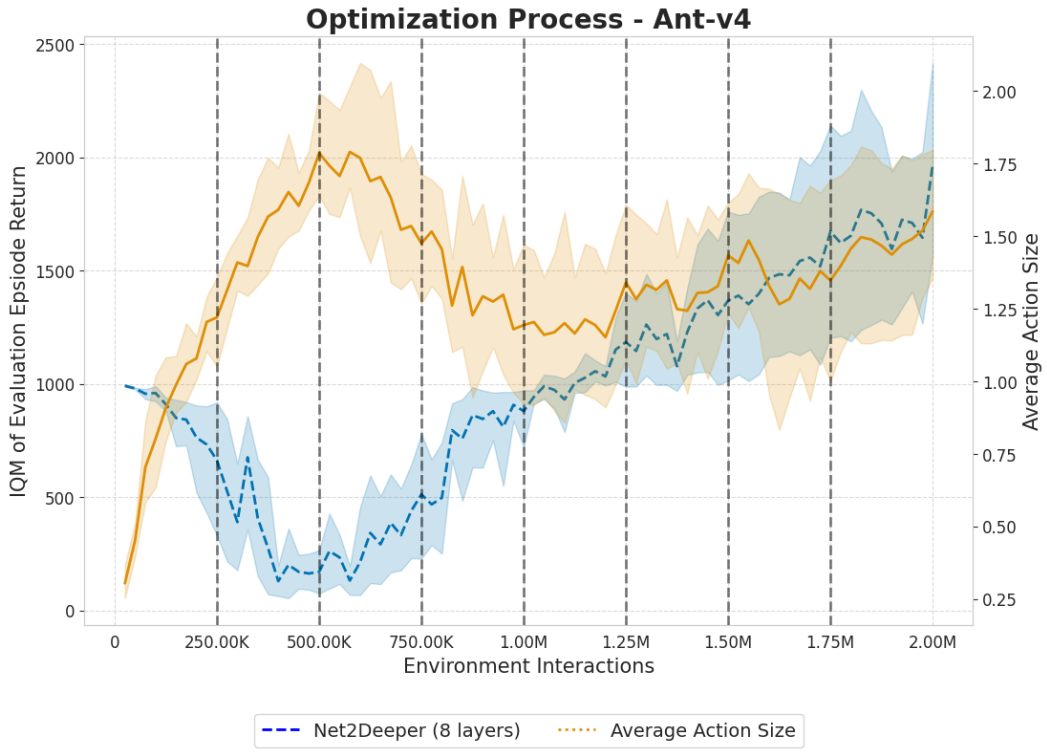


Figure 5.17.: Portrayal of the grown depth 8 incumbent vs. its average action sizes.

From these results, we interpret that agents may initially heavily explore the usage of large actions until the agent realizes that these actions are not helpful. It then reduces action sizes and explores suitable actions, resulting in an increased return. Since this analysis requires a re-execution of the experiments, we cannot verify this for the competitors, but based on the reward curve, we expect a similar result.

Network Weights Figure 5.18 shows the network weights of the depth 8 incumbent on the Ant environment. The plots show the same behavior as on MiniHack. The additional plots can be found in Appendix A.5.3.

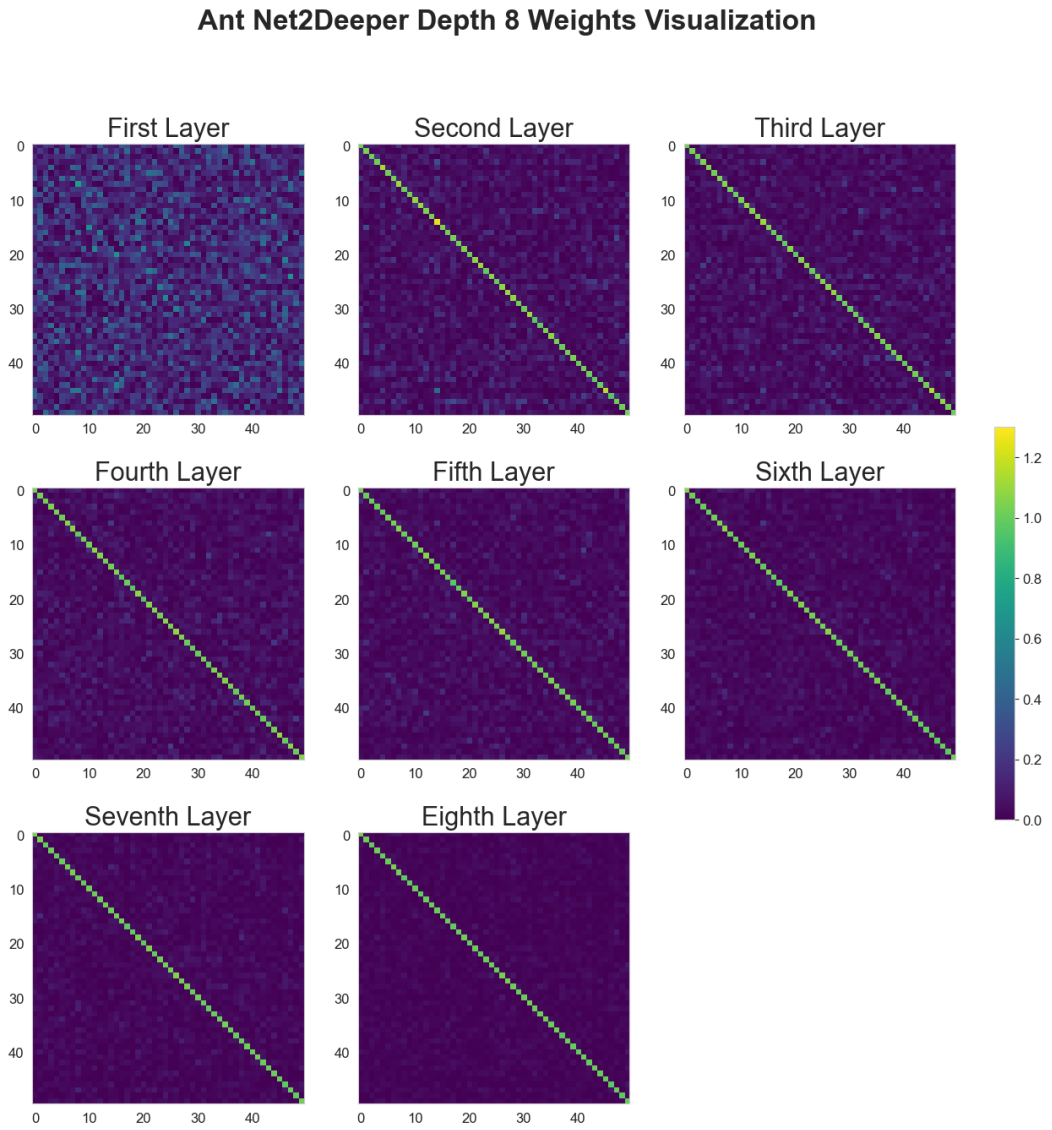


Figure 5.18.: Visualization of a 50×50 subset of the weights learned inside the feature extractor.

Conclusion From our experiments, we can conclude that our approach of using Network Depth as fidelity outperforms Black Box Optimization with a fixed-size feature extractor on the Ant environment.

5.2 Increasingly Difficult Environments

Research question three asks whether the network growth mechanism can improve the learning process when transitioning from easier to more complicated environments. To explore this research question, we conduct experiments using the Net2Deeper network growth approach, transitioning from a 10×10 grid to a 15×15 grid.

We conducted this experiment for both the growth of the feature extractor to depth two and the growth to depth four. For depth two, we increase the network difficulty when the neural network is grown; for depth four, we explore increasing environment difficulty at half or three-quarters of the training process. Our baselines, i.e., statically sized agents configured with Black Box HPO, transition between fidelities after the same number of timesteps in the training process.

Optimization Process In Figure 5.19, we compare the returns of the different resulting approaches throughout the Hyperparameter Optimization Process on MiniHack Random 15×15.

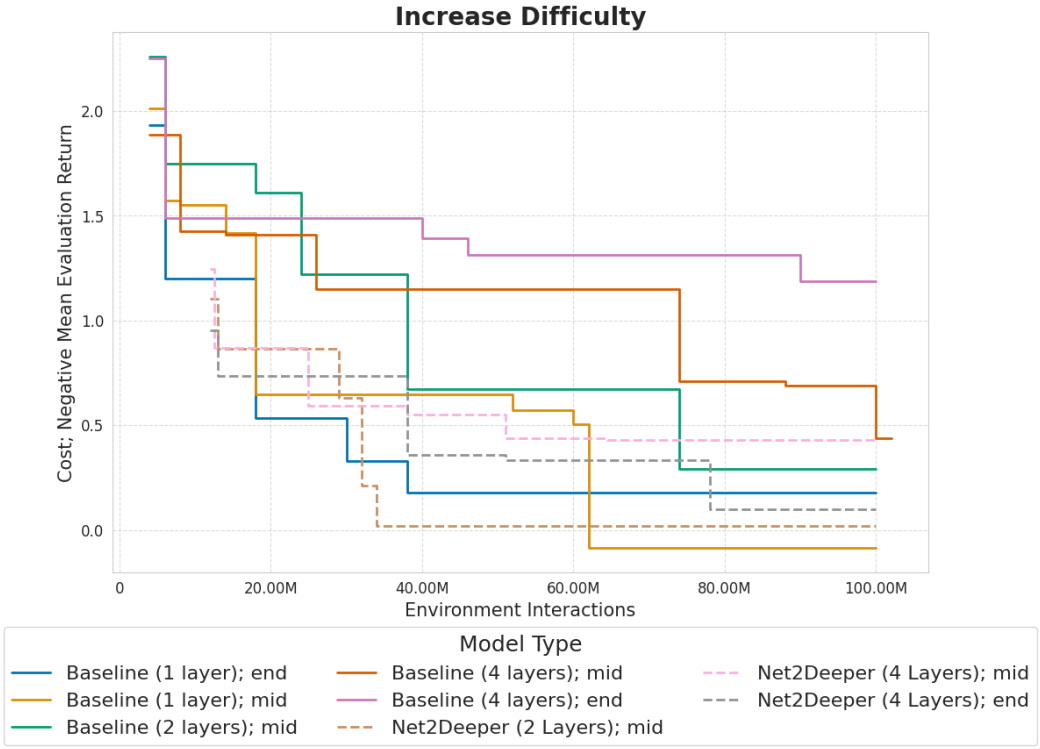


Figure 5.19.: Optimization Process of incremental growth combined with environment difficulty increases. For the first 1 Million steps, the approaches are trained on MiniHack Room Random 10×10, and transition to MiniHack Room random 15×15.

The plot shows that none of the approaches manage to master the 15×15 environment, and only the depth one baseline with increased difficulty in the middle of the training process reliably avoids negative rewards.

Over the entire optimization process, the plots indicate that the network growth approaches outperform the baselines equipped with their maximum depth. The only competitive baseline appears to be trained with depth one.

From 18 to 62 million environment interactions, the depth two network growth approach outperforms all other approaches. At 62 Million Interactions, the depth one baseline transitioning to 15×15 after 1 Million timesteps finds a new incumbent yielding a cost of slightly less than 0. At the end of the optimization process, all network growth approaches outperform their respective max depth baselines. However, the depth one baseline remains competitive and, if configured to transition between difficulties after 1 Million timesteps, beats all other approaches.

The budget correlations of the different approaches shown in A.4.1 indicate similar behavior as in the initial Net2DeeperNet results. The main difference is that upon transition between environments, the correlations are decreased.

Our analysis of the hyperparameter configuration, shown in Appendix A.4.2, shows very similar results as for the previous Net2Deeper Experiments on MiniHack. Except for network growth selecting small batch sizes, there is no clear trend visible.

Incumbent Training Process As previously, the figures below compare the training performance of incumbents and their respective baselines. The black vertical lines indicate network growth, and the red vertical line indicates network growth combined with increased difficulty.

Figure 5.20 visualizes the training process of the network growth approach equipped with two layers, along with static baselines equipped with either one or two layers. As can be seen in the plot, up to the network growth at 1 Million timesteps, the depth one approaches appear to outperform the two-layer static baseline. In contrast to the depth two baseline, which collects negative returns, they manage to avoid them on average.

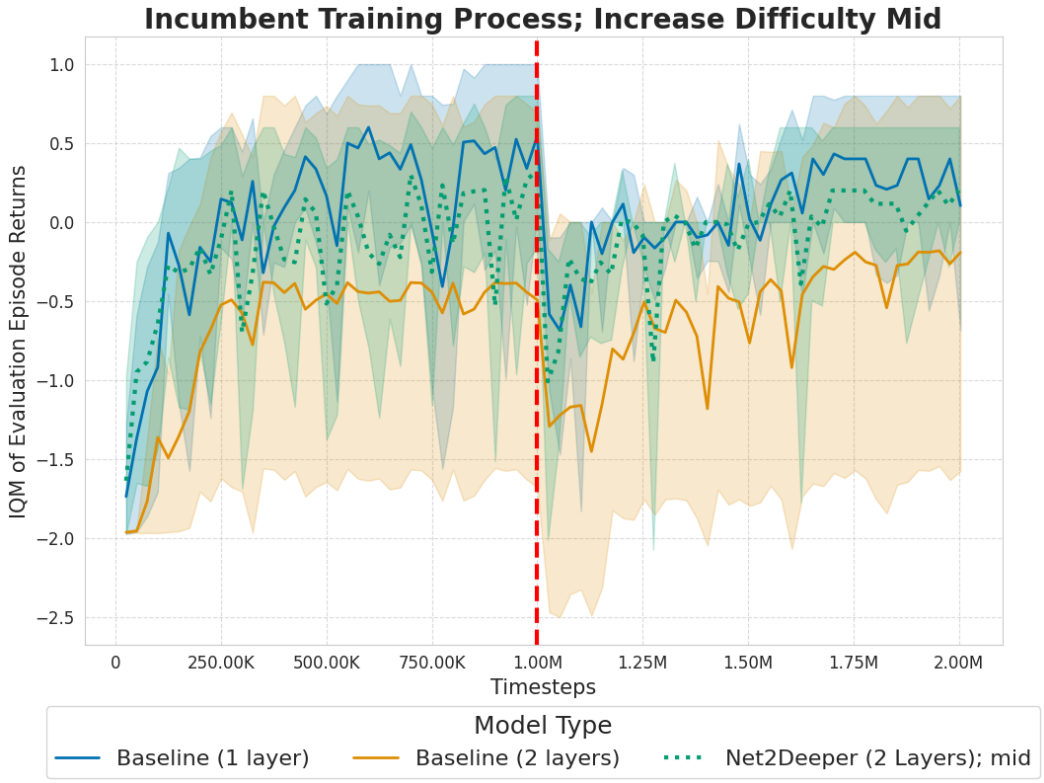


Figure 5.20.: Comparison of the incumbent configuration's training process on the MiniHack environment. The plot contains agents configured with feature extractors of depth two.

After transitioning between fidelities and increasing environmental difficulty at 1 Million timesteps, all approaches suffer from a drop in performance. While the depth one baseline and network growth approach suffer from a smaller drop in performance and remain superior to the depth two baseline, their difference in performance decreases over the optimization process. In the end the depth one baseline slightly outperforms the network growth approach.

Figure 5.21 portrays the results for depth four, with the difficulty increased after 1 Million timesteps.

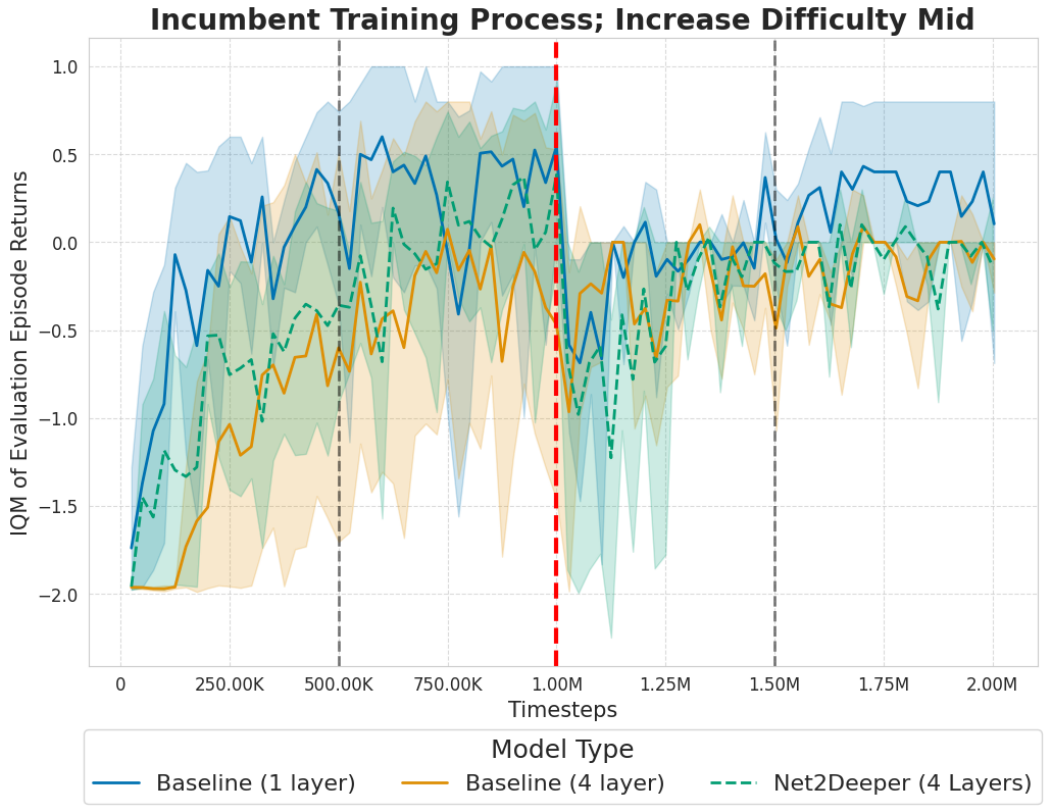


Figure 5.21.: Comparison of the incumbent configuration's training process on the MiniHack environment. The plot contains agents configured with feature extractors of depth four.

Up to 1 Million timesteps, we perceive very similar training behavior to the depth two approaches. However, after transitioning fidelities, the depth 4 approach settles for an average return slightly less than 0, while the depth one baseline yields slightly more than 0.

In contrast, a later difficulty increase appears to distinguish the approaches, as shown in Figure 5.22.

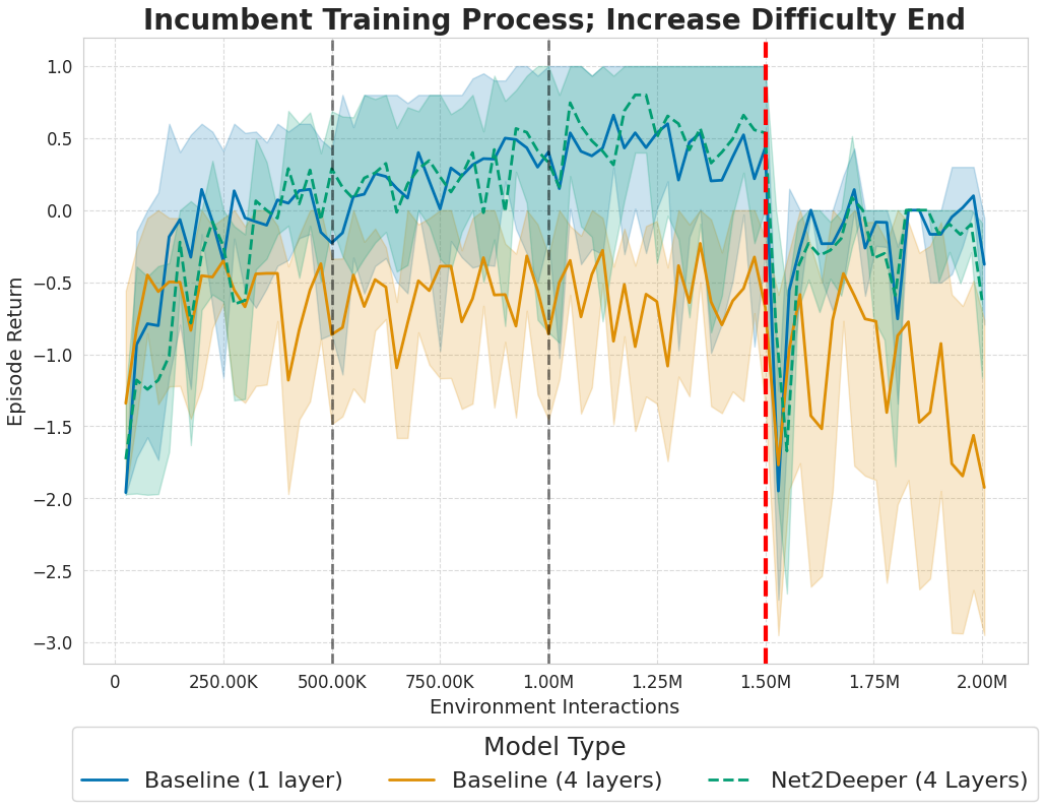


Figure 5.22.: Comparison of the incumbent configuration's training process on the MiniHack environment. The plot contains agents configured with feature extractors of depth four.

While the figure again shows similar training behavior until the difficulty is increased at 1.75 Million timesteps, the four-layer baseline struggles much more with the higher environment difficulty. The depth one baseline and grown approach again settle for a higher episode return. However, the achieved return is considerably smaller than if the difficulty is increased at 1 Million timesteps.

Network Weights Figure 5.23 shows the network weights learned by the feature extractor grown to depth two.

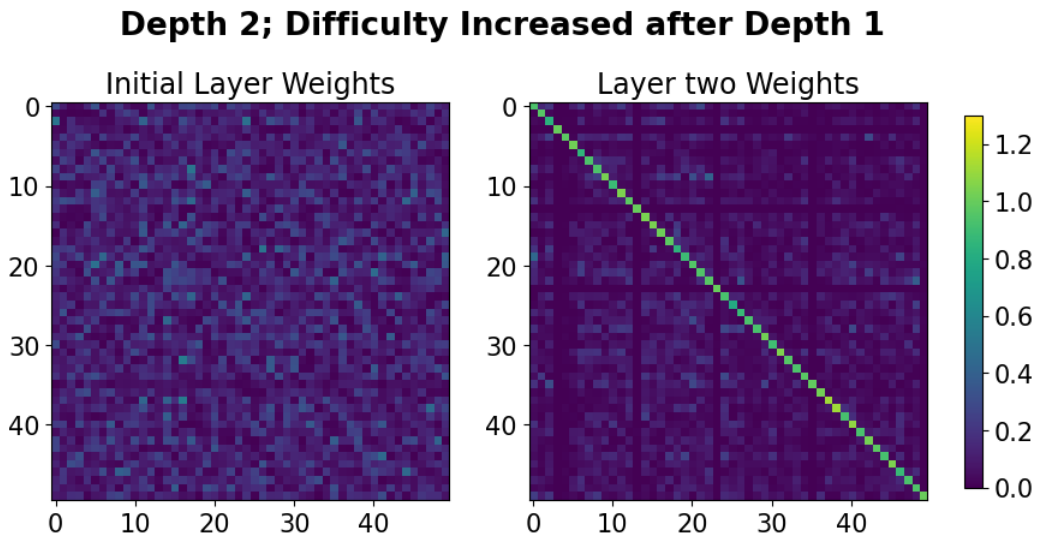


Figure 5.23.: Visualization of a 50×50 subset of weights learned inside the feature extractor.

The additional weight visualizations, which indicate similar behavior, are in Appendix A.5.3.

Conclusions From the results, we conclude that network growth with Net2DeeperNet does not only not hinder when transitioning from MiniHack Room 10×10 to MiniHack Room 15×15 but allows all network growth approaches to perform better than their respective baselines, with the growth approaches even remaining competitive with the depth one baseline.

Conclusion

6

In this chapter, we summarize the key findings of the thesis by revisiting the research questions raised in Chapter 1 and providing answers to them, along with additional insights, in Section 6.1. We then discuss the limitations of our work and propose future research directions in Section 6.2.

6.1 Main Takeaways

The first research question, "*How can network growing approaches (prominent in other machine learning areas) be employed in reinforcement learning (RL)?*", is explored throughout this thesis. Our approach demonstrated that network growth can enable using deeper networks by enabling knowledge transfer from smaller to larger networks. Additionally, the grown networks continue training on the higher fidelities, often outperforming the performance before growth.

The second research question, "*In what way does the learning process change upon network adaptation?*", was addressed in our experiments, i.e. in Section 5.1. We found that when using Net2DeeperNet, knowledge is largely preserved, and there is almost no drop in performance. Our analysis of the budget correlations also showed that well-ranked networks retain their performance after depth increases. Additionally, Net2DeeperNet allows to train agents of depth 4 on MiniHack Room, thus outperforming their respective baseline and beating all evaluated baselines in the Ant environment. Conversely, with Net2WiderNet, we observed a substantial performance drop following network transformations, and although agents eventually recover, they cannot beat their respective baselines. Additionally, the budget correlations are much weaker, indicating that network growth results in a loss of knowledge and that wider networks may require a hyperparameter change to retain performance. This suggests that while Net2DeeperNet aids training relative to a static baseline, Net2WiderNet tends to impair it.

The third research question, "*When transitioning from simpler to more complex problems, does iteratively increasing the depth/width of a pre-trained neural network yield better results than starting with the final/initial network size?*" was addressed in our experiments in Section 5.2. Similarly to the static environment, our results indicate that network growth performs better than the baselines trained with the full network from the beginning. Additionally, in contrast to the static environments, network growth remains competitive with the depth of one agent.

From these results, we conclude that in static environments, our approach to combining Multi Fidelity Optimization with Net2DeeperNet should be considered. If transitioning between environments, we propose to use network growth instead of static networks but remain that other approaches might perform even better.

6.2 Limitations and Future Work

As most limitations of our evaluation setup and proposed methodology directly result in future research directions, we discuss limitations and future work jointly. To that end, we first focus on limitations to then discuss future work that goes boyend them.

Limitations Several aspects of our evaluation process could be improved. First, to enhance the reliability of the results, additional evaluations should be conducted, particularly by running SMAC with more random seeds and evaluating the approach on a broader set of environments. Secondly, ablation studies on the hyperparameters of the multi-fidelity method, baseline models, and Net2WiderNet's hyperparameters would provide deeper insights into the sensitivity of the proposed approach.

Similarly, our evaluation used the weights of the neural network. While these can be seen as a proxy of their impact on the performance, they are less suitable for this evaluation than other measures, for example, an analysis of dormant neurons (Sokar et al., 2023). We therefore propose such an analysis of dormant neurons as a suitable extension,

The multi-fidelity approach and network growth were used in conjunction in this thesis, which makes it difficult to determine the extent to which the results can be attributed solely to network growth, the multi-fidelity schedule, or a combination of

both. It is important to evaluate these strategies independently to better understand their contributions, utilizing them as baselines for our conjunct approach.
Particularly for the research question tree, such and other additional baselines would help to further understand the effect of our approach further(Abbas et al., 2023).

Our experiments focused exclusively on feedforward neural networks, omitting other potential improvements to the reinforcement learning process, such as different network architectures such as CNNs (LeCun et al., 1989) or techniques that enhance training efficiency (Cobbe et al., 2019). As a result, our findings should not be taken as definitive proof that network growth will work on other architectures or in cases where such enhancements are already applied, since they may be solving the same problems (Cobbe et al., 2019).

Independent Future Work Firstly, our approach utilized PPO and introduced network growth to feature extractors. Future work should explore using similar ideas for other approaches, especially for Q-Learning (Mnih et al., 2015) where the replay buffer could further train the added neurons to utilize growth approaches that focus on recombining pre-existing structures instead of using network morphisms as proposed by(Pham et al., 2024).

Similarly, our approach showed that mutli-fidelity ideas might transfer into the field of Reinforcement Learning. However, there is little research directed into both validating pre-existing methods for RL and developing RL-specific methods. We believe that much research can be done here.

Thirdly, our approach showed that incrementally growing neural networks with Net2Net-style (T. Chen et al., 2016) operations may enhance the training progress in RL. As indicated, future work should, therefore, try to build upon these results and add additional growth methodologies, both major operations as proposed by (Wei et al., 2016), and minor operations such as proposed by (Wu et al., 2019). Additionally, we propose utilizing the DAC (Adriaensen et al., 2022) methodology to dynamically decide on network growth based on the learning process and the related teacher student curriculum learning to select which environments to train on (Matiisen et al., 2017).

Appendix A

A.1 Net2Deeper

A.1.1 Budget Correlation

Given the configurations used in a pairwise selection of the budgets 1, 2, 3, and 4, the rankings of configurations are compared using the Spearman Rank Correlation (Spearman, 1961). -1 indicates a perfect negative correlation, 0 no correlation, and 1 perfect correlation.

MiniHack Room Random

As represented in Table A.1, there is a high correlation between the different budgets on MiniHack Room Random 10×10, decaying for increasing budget differences. This validates our Multi-Fidelity approach and supports the idea that the surrogate selecting new configurations should be trained on the highest fidelity with sufficient data points.

Table A.1.: Correlation Matrices of the Net2Deeper optimization runs on 10×10 Random. Table A.1a shows the results on 4 layers, Table A.1b shows the results on two layers.

(a) Correlation Matrix over the different budgets in Net2Deeper Random with four layers.

	1 Layer	2 Layer	3 Layer	4 Layer
1 Layer	1.00	0.67	0.67	0.58
2 Layer	0.67	1.00	0.91	0.75
3 Layer	0.67	0.91	1.00	0.89
4 Layer	0.58	0.75	0.89	1.00

(b) Correlation Matrix over the different budgets in Net2Deeper Random with two layers.

	1 Layer	2 Layer
1 Layer	1.00	0.72
2 Layer	0.72	1.00

Minihack Room Monster 10x10

As represented in Table A.2, there is a high correlation between the different budgets on MiniHack Room Monster 10×10, decaying for increasing budget differences. This validates our Multi-Fidelity approach and supports the idea that the surrogate selecting new configurations should be trained on the highest fidelity with sufficient data points.

Table A.2.: Correlation Matrices of the Net2Deeper optimization runs. Table A.2a shows the results on 4 layers, Table A.2b shows the results on two layers.

(a) Correlation Matrix over the different budgets in Net2Deeper Random with two layers.

	1 Layer	2 Layer	3 Layer	4 Layer
1 Layer	1.00	0.70	0.63	0.72
2 Layer	0.70	1.00	0.90	0.85
3 Layer	0.63	0.90	1.00	0.91
4 Layer	0.72	0.85	0.91	1.00

(b) Correlation Matrix over the different budgets in Net2Deeper Random with two layers.

	1 Layer	2 Layer
1 Layer	1.00	0.94
2 Layer	0.94	1.00

A.1.2 Hyperparameter Configurations

Environment	Experiment	Batch Size	Entropy	Learning Rate	Epochs
Room	Baseline 1	32	0.0011	0.00014	5
Monster	Baseline 1	64	0.0035	0.00020	5
Room	Baseline 2	128	0.0010	0.00031	8
Monster	Baseline 2	64	0.0006	0.00020	10
Room	Baseline 4	256	0.0013	0.00017	16
Monster	Baseline 4	256	0.1174	0.00029	20
Room	Grow 2	32	0.0005	0.00020	10
Monster	Grow 2	32	0.00048	0.00019	10
Room	Grow 4	32	0.0019	0.00010	8
Monster	Grow 4	32	0.0008	0.00017	8

Table A.3.: The table shows the selected hyperparameter configurations for Net2DeeperNet

A.1.3 Incumbent Training Process

Figure A.1 shows the training process of an agent grown to depth four using Net2DeeperNet on MiniHack Room Random. As shown by the plot, the network growth approach vastly outperforms the depth 4 baseline but performs slightly inferior to the depth one baseline.

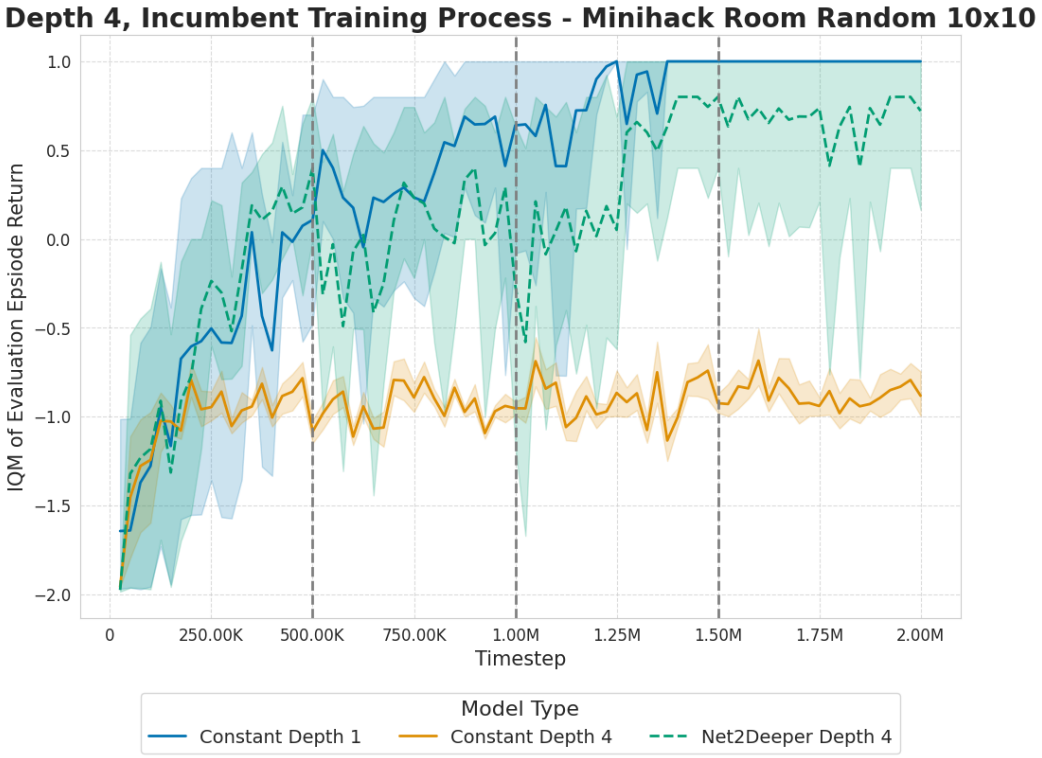
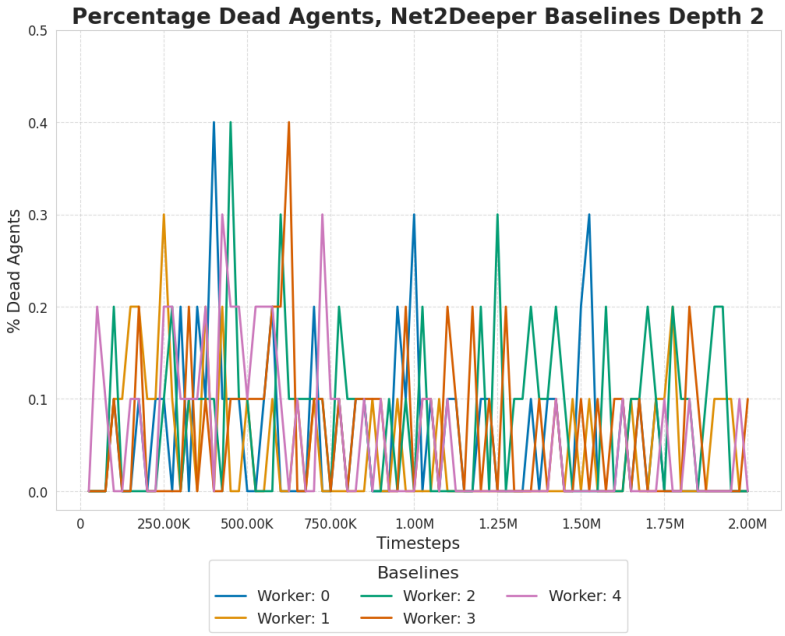


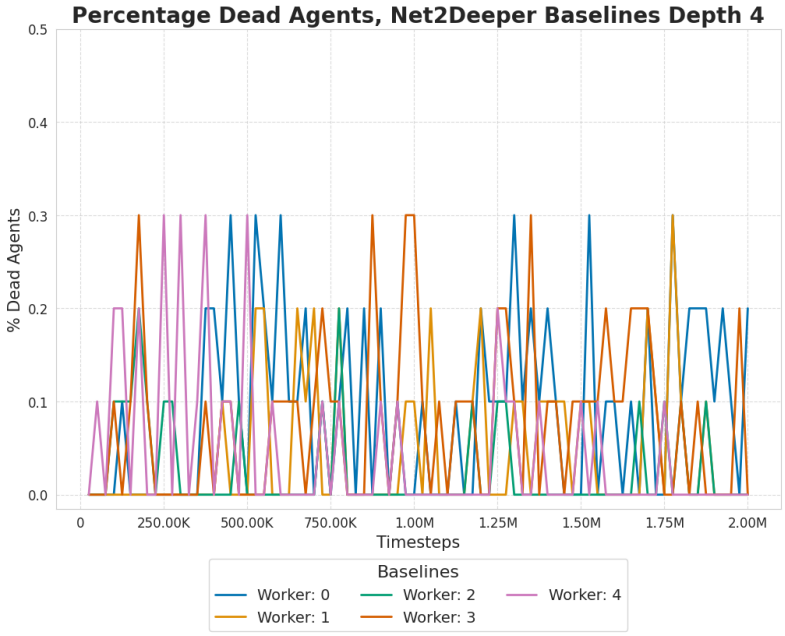
Figure A.1.: Comparison of depth 4 incumbent training process on MiniHack Room Random 10x10. The mean of the interquartile mean of 10 evaluation episodes on the y-axis with 95% confidence intervals. Number of environment interactions on the x-axis.

Interaction with Monsters

To explore whether agents learn to intentionally die in the Monster Environment, we plot the incumbent's training behavior in Figure A.2



(a) Net2Deeper Baseline agent with depth 2.



(b) Net2Deeper Baseline agent with depth 4.

Figure A.2.: Above figures Figure A.2a and A.2b, portray how often the agents die in the evaluation episodes during training. The figures show that the agents do not appear to learn that dying is associated with a 0 reward. The timesteps are plotted along the x-axis, and the percentage of dead agents is plotted on the y-axis.

A.2 Net2Wider

A.2.1 Budget Corrolations

Table A.4.: Correlation Matrices of the Net2Wider optimization runs on 10×10 Random. Table A.1a shows the results if grown to width 4096, Table A.4b shows the results if grown to 1024.

(a) Correlation Matrix over the different budgets in Net2Wider Random grown to feature extractor width 4096.

Width	512	1024	2048	4096
512	1.00	0.45	0.18	0.34
1024	0.45	1.00	0.49	0.10
2048	0.18	0.49	1.00	0.40
4096	0.34	0.10	0.40	1.00

(b) Correlation Matrix over the different budgets in Net2Wider Random grown to feature extractor width 2048.

Width	512	1024
512	1.00	0.95
1024	0.95	1.00

Table A.5.: Correlation Matrices of the Net2Wider optimization runs on 10×10 Monster. Table A.5a shows the results if grown to width 4096, Table A.5 shows the results if grown to 1024.

(a) Correlation Matrix over the different budgets in Net2Wider Random grown to feature extractor width 4096.

	512	1024	2048	4096
512	1.00	0.79	0.80	0.68
1024	0.79	1.00	0.82	0.77
2048	0.80	0.82	1.00	0.93
4096	0.68	0.77	0.93	1.00

(b) Correlation Matrix over the different budgets in Net2Wider Random grown to feature extractor width 2048.

	512	1024
512	1.00	0.57
1024	0.57	1.00

A.2.2 Hyperparameter Configurations

Environment	Experiment	Batch Size	Entropy	Learning Rate	Epochs
Room	Baseline 1	32	0.0011	0.00014	5
Monster	Baseline 1	64	0.0035	0.00020	5
Room	Baseline 2	256	0.0090	0.00020	5
Monster	Baseline 2	256	0.0013	0.00041	11
Room	Baseline 4	64	0.0084	0.00012	5
Monster	Baseline 4	128	0.00038	0.00020	9
Room	Grow 2	32	0.0053	0.00020	10
Monster	Grow 2	64	0.0033	0.00023	9
Room	Grow 4	128	0.0060	0.00020	5
Monster	Grow 4	256	0.0088	0.00050	7

Table A.6.: The table shows the selected hyperparameter configurations for Net2WiderNet

A.2.3 Incumbent Training Process on Minihack Room Random 10x10

As shown in Figure A.3 and Figure A.4, applying Net2WiderNet on MiniHack Room Random fails.

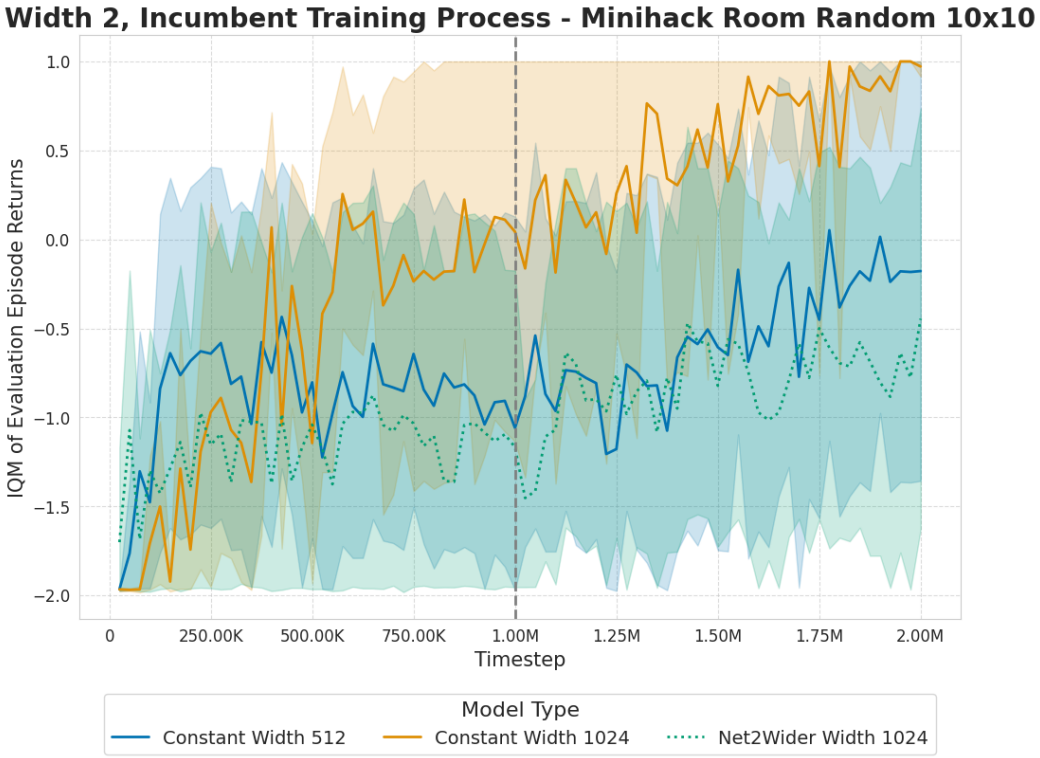


Figure A.3.: Comparison of width 1024 incumbent training process on MiniHack Room Random 10x10. The mean of the interquartile mean of 10 evaluation episodes on the y-axis with 95% confidence intervals. Number of environment interactions on the x-axis.

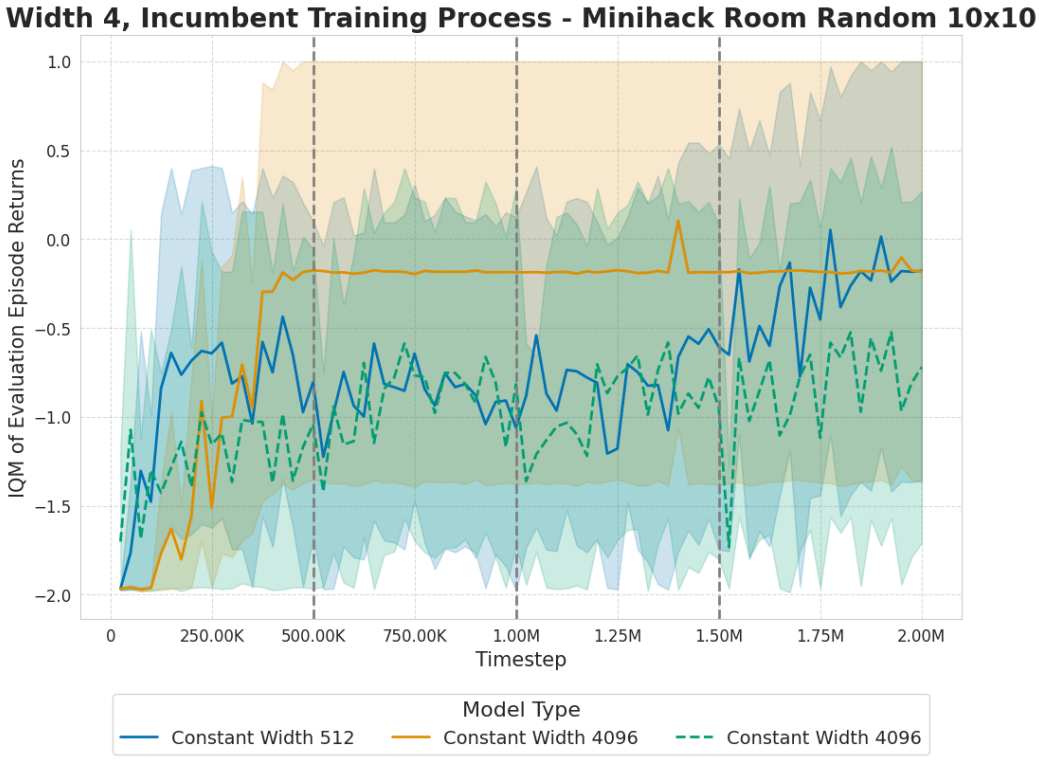


Figure A.4.: Comparison of width 4096 incumbent training process on MiniHack Room Random 10x10. The mean of the interquartile mean of 10 evaluation episodes on the y-axis with 95% confidence intervals. Number of environment interactions on the x-axis.

A.3 Ant

A.3.1 Budget Correlations

Table A.7.: Budget Correlations for Net2DeeperNet with depth four on Ant.

	Depth 1	Depth 2	Depth 3	Depth 4
Depth 1	1.00	0.81	0.79	0.52
Depth 2	0.81	1.00	0.82	0.80
Depth 3	0.79	0.82	1.00	0.74
Depth 4	0.52	0.80	0.74	1.00

Table A.8.: Budget Correlations for Net2DeeperNet with depth eight on Ant.

	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7	Depth 8
Depth 1	1.00	0.86	0.81	0.77	0.81	0.77	0.80	0.73
Depth 2	0.86	1.00	0.88	0.93	0.88	0.88	0.89	0.89
Depth 3	0.81	0.88	1.00	0.92	0.83	0.84	0.88	0.86
Depth 4	0.77	0.93	0.92	1.00	0.90	0.87	0.87	0.87
Depth 5	0.81	0.88	0.83	0.90	1.00	0.95	0.91	0.95
Depth 6	0.77	0.88	0.84	0.87	0.95	1.00	0.93	0.98
Depth 7	0.80	0.89	0.88	0.87	0.91	0.93	1.00	0.92
Depth 8	0.73	0.89	0.86	0.87	0.95	0.98	0.92	1.00

A.3.2 Hyperparameter Configurations

Experiment	Batch Size	Entropy	Learning Rate	Epochs
Baseline 2	256	0.0020	0.0001	8
Baseline 4	64	0.0017	0.0001	6
Baseline 8	32	0.0017	0.0006	5
Grow 4	32	0.0010	0.0001	5
Grow 8	256	0.0028	0.0001	12

Table A.9.: The table shows the selected hyperparameter configurations for Ant

A.4 Increase Difficulty

A.4.1 Budget Correlations

Table A.10.: Budget Correlations for Difficulty Increases using network growth to depth two. Upon switching fidelities, the network difficulty gets increased.

	Budget 1	Budget 2
Budget 1	1.00	0.54
Budget 2	0.54	1.00

Table A.11.: Budget Correlations for Difficulty Increases using network growth to depth four. Upon switching fidelities from depth two to depth three, the network difficulty gets increased.

	Budget 1	Budget 2	Budget 3	Budget 4
Budget 1	1.00	0.67	0.67	0.45
Budget 2	0.67	1.00	0.91	0.70
Budget 3	0.67	0.91	1.00	0.74
Budget 4	0.45	0.70	0.74	1.00

Table A.12.: Budget Correlations for Difficulty Increases using network growth to depth four. Upon switching fidelities from depth three to depth four, the network difficulty gets increased.

	Budget 1	Budget 2	Budget 3	Budget 4
Budget 1	1.00	0.67	0.39	0.51
Budget 2	0.67	1.00	0.49	0.62
Budget 3	0.39	0.49	1.00	0.57
Budget 4	0.51	0.62	0.57	1.00

A.4.2 Hyperparameter Configurations

Experiment	**Batch Size**	**Entropy**	**Learning Rate**	**Epochs**
Baseline 1 Layer Mid	256	0.0034	0.0007	5
Baseline 1 Layer End	64	0.0130	0.0004	6
Baseline 2 Layers Mid	256	0.0060	0.0006	5
Baseline 4 Layers Mid	256	0.03514	0.0002	6
Baseline 4 Layers End	256	0.0736	0.0020	20
Net2Deeper 2 Layers Mid	64	0.0188	0.0004	5
Net2Deeper 4 Layers Mid	32	0.0020	0.0001	8
Net2Deeper 4 Layers End	32	0.0005	0.0002	10

Table A.13.: The table shows the selected hyperparameter configurations for the experiments where we increase environment difficulty during the learning process.

A.5 Network Weights

A.5.1 MiniHack Room Net2DeeperNet

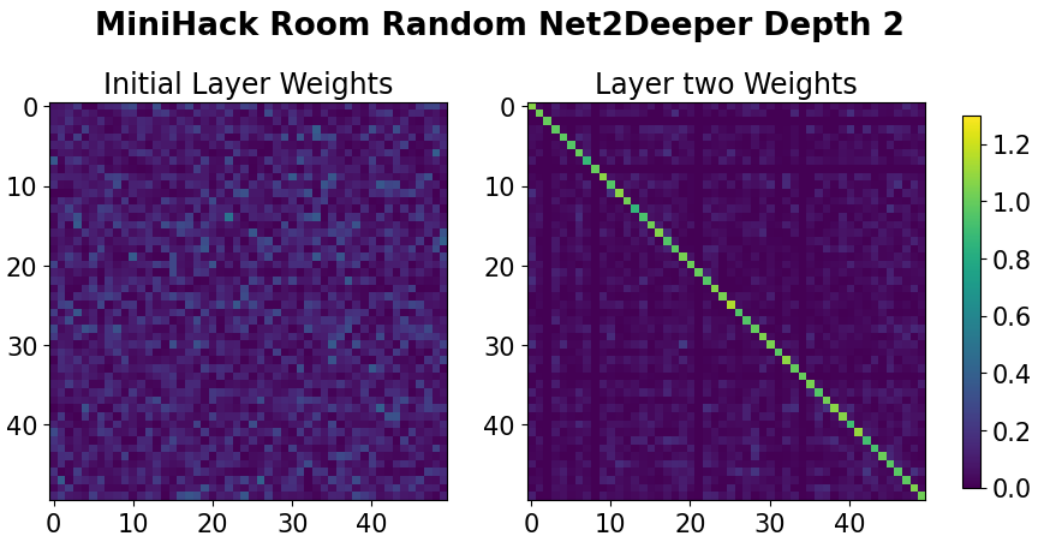


Figure A.5.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

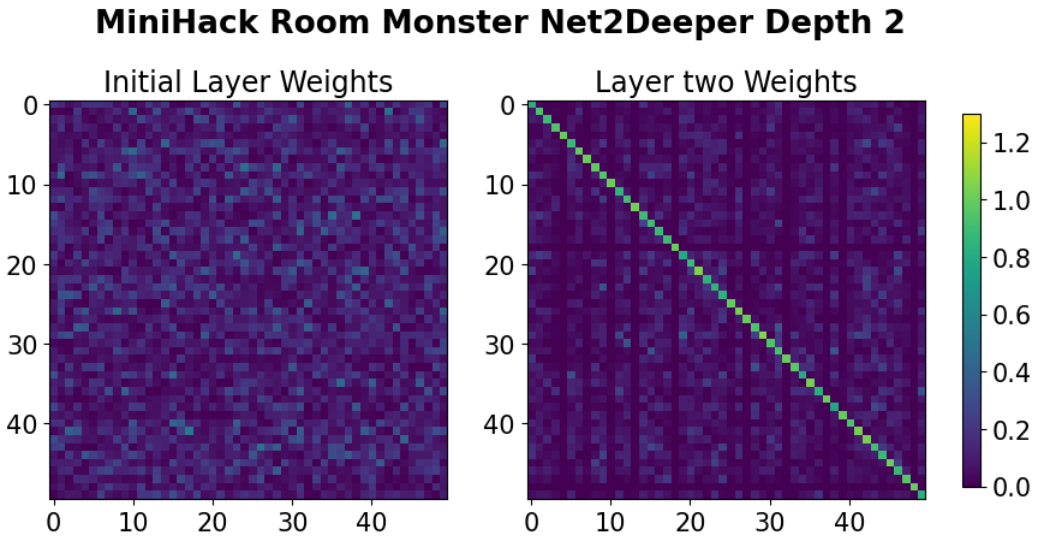


Figure A.6.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

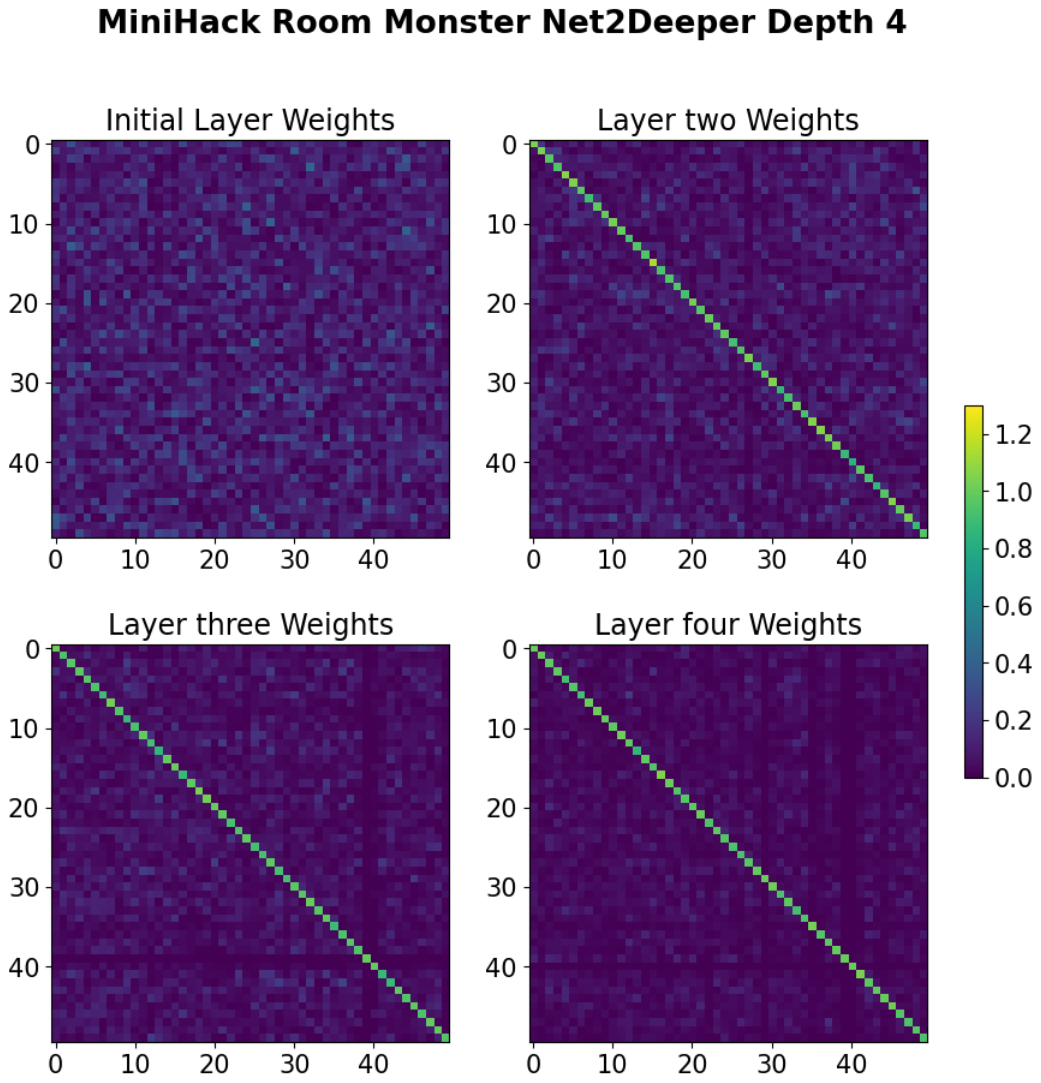


Figure A.7.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

A.5.2 MiniHack Room Net2WiderNet

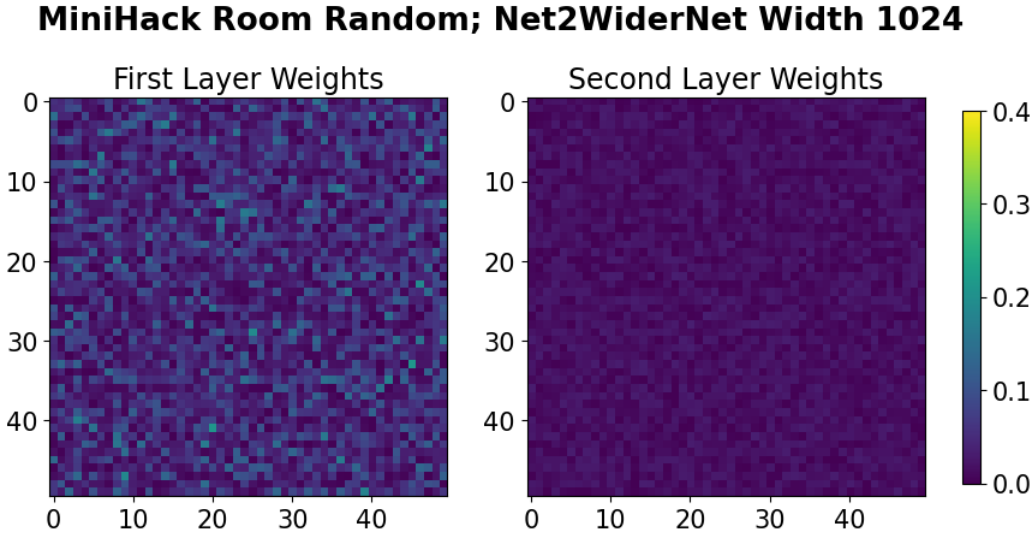


Figure A.8.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

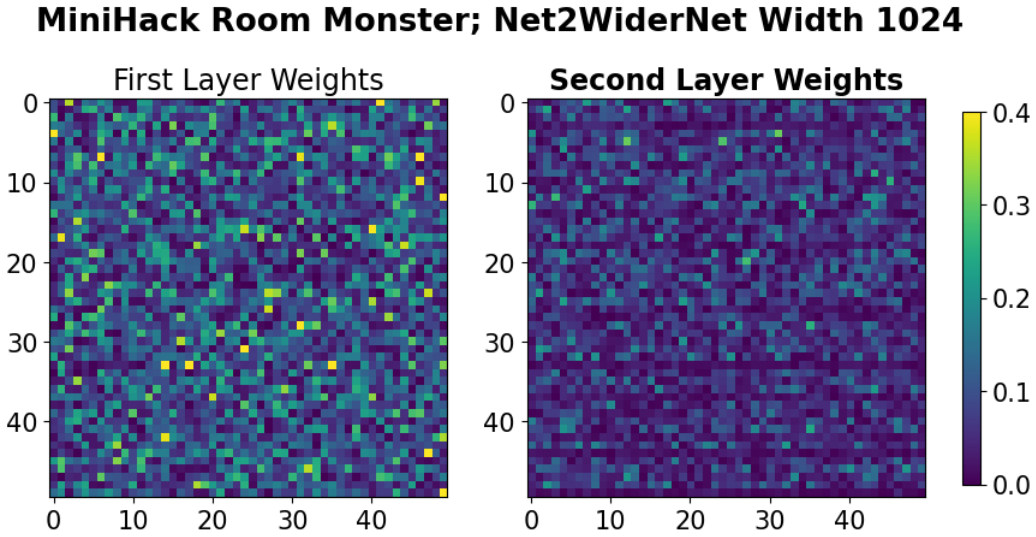


Figure A.9.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

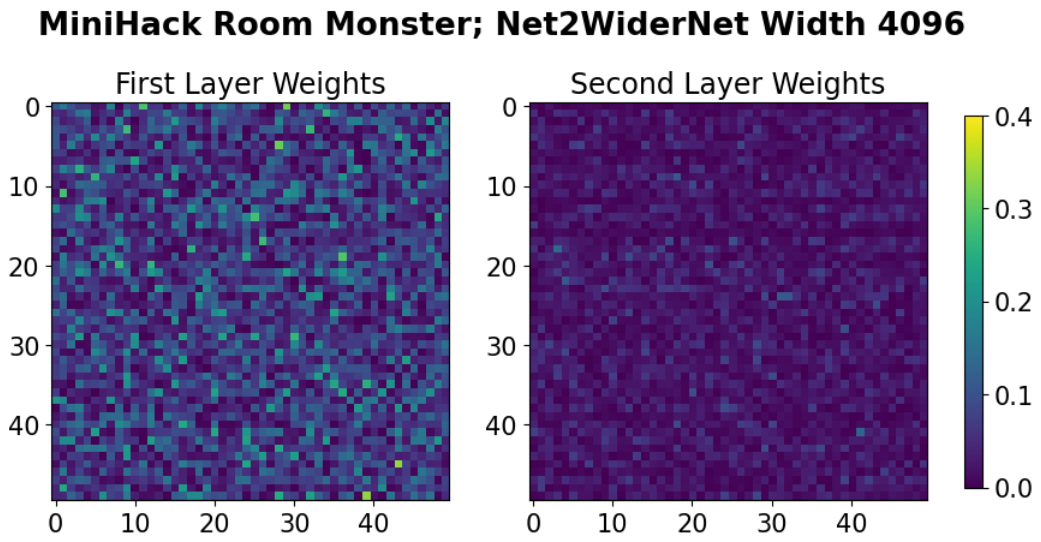


Figure A.10.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

A.5.3 Ant

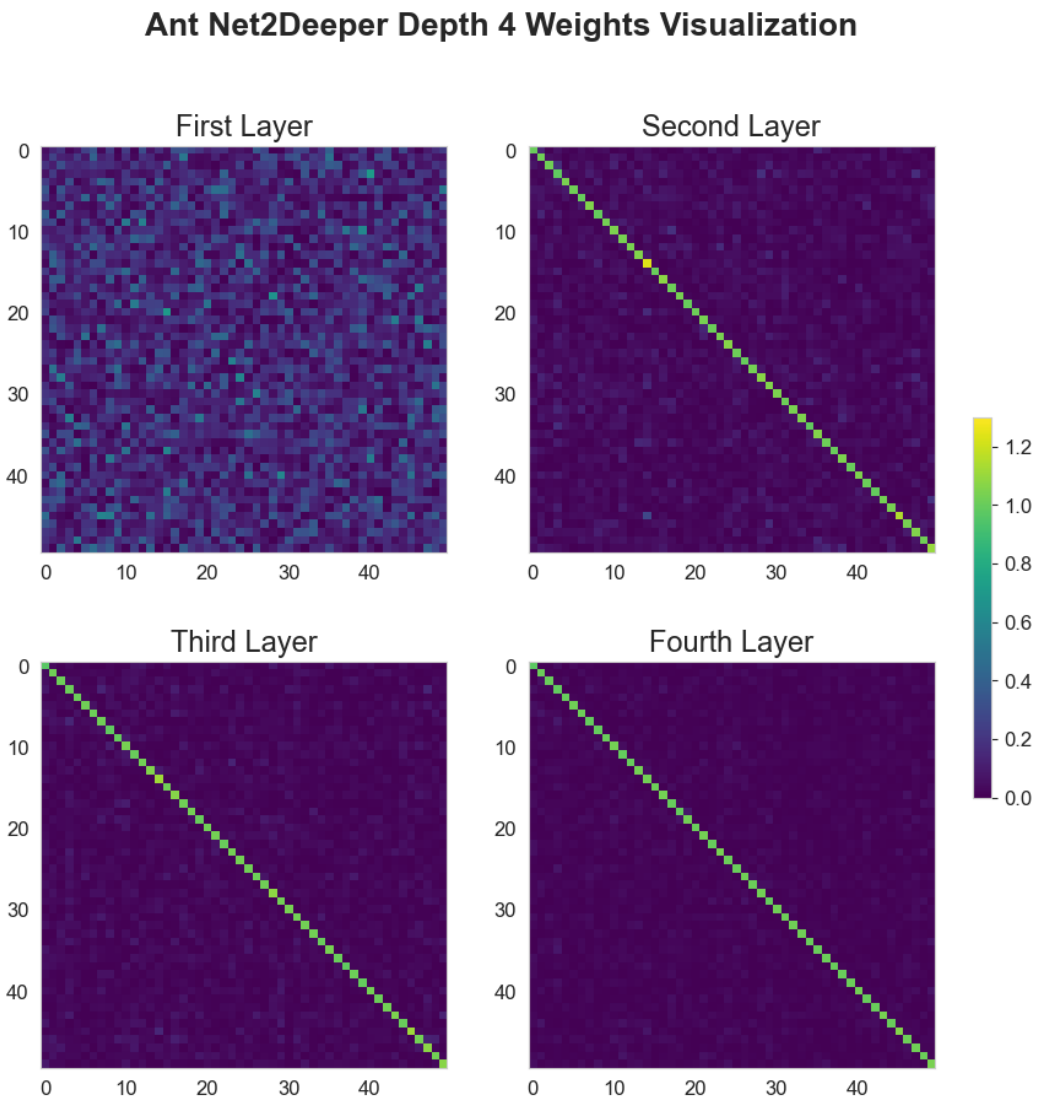


Figure A.11.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

A.5.4 Difficulty Increases

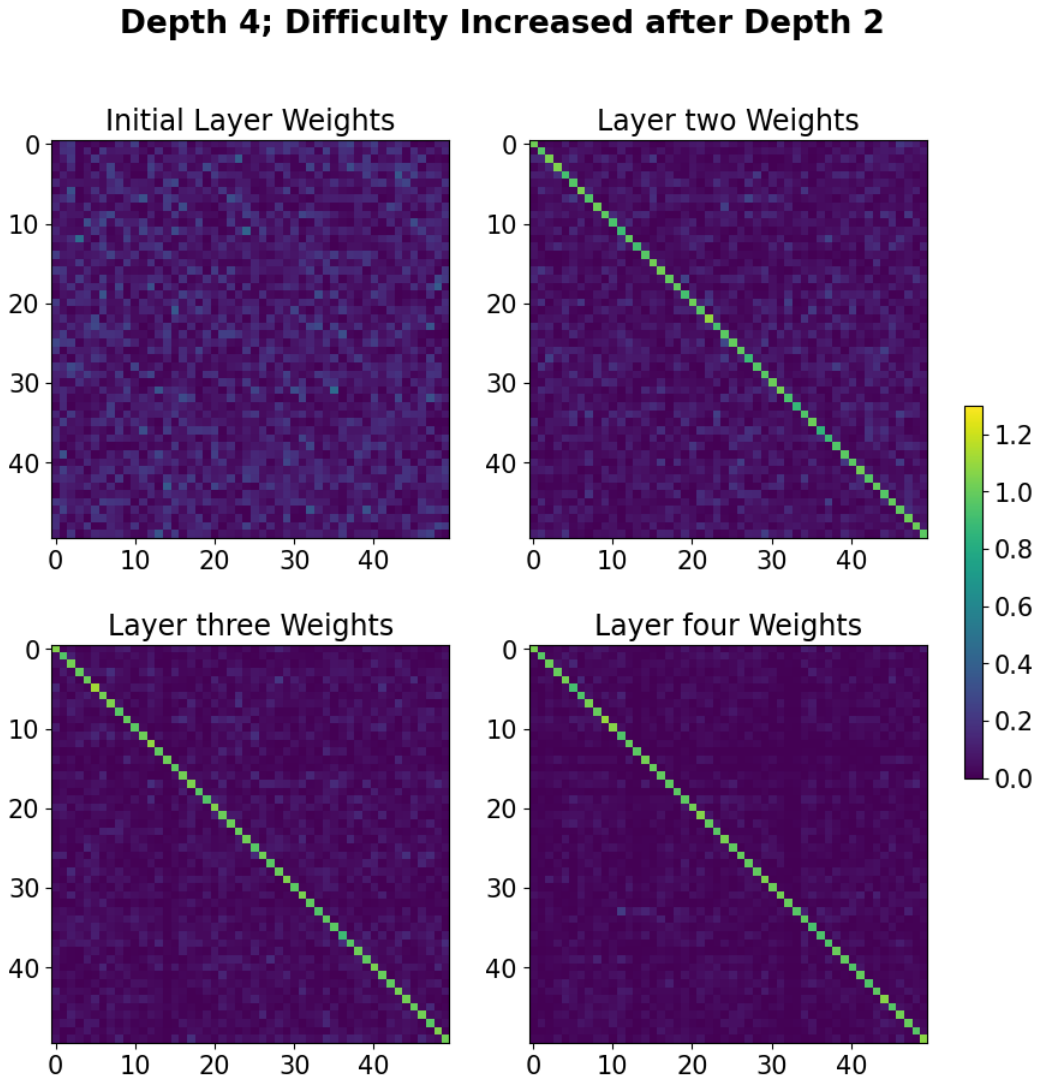


Figure A.12.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

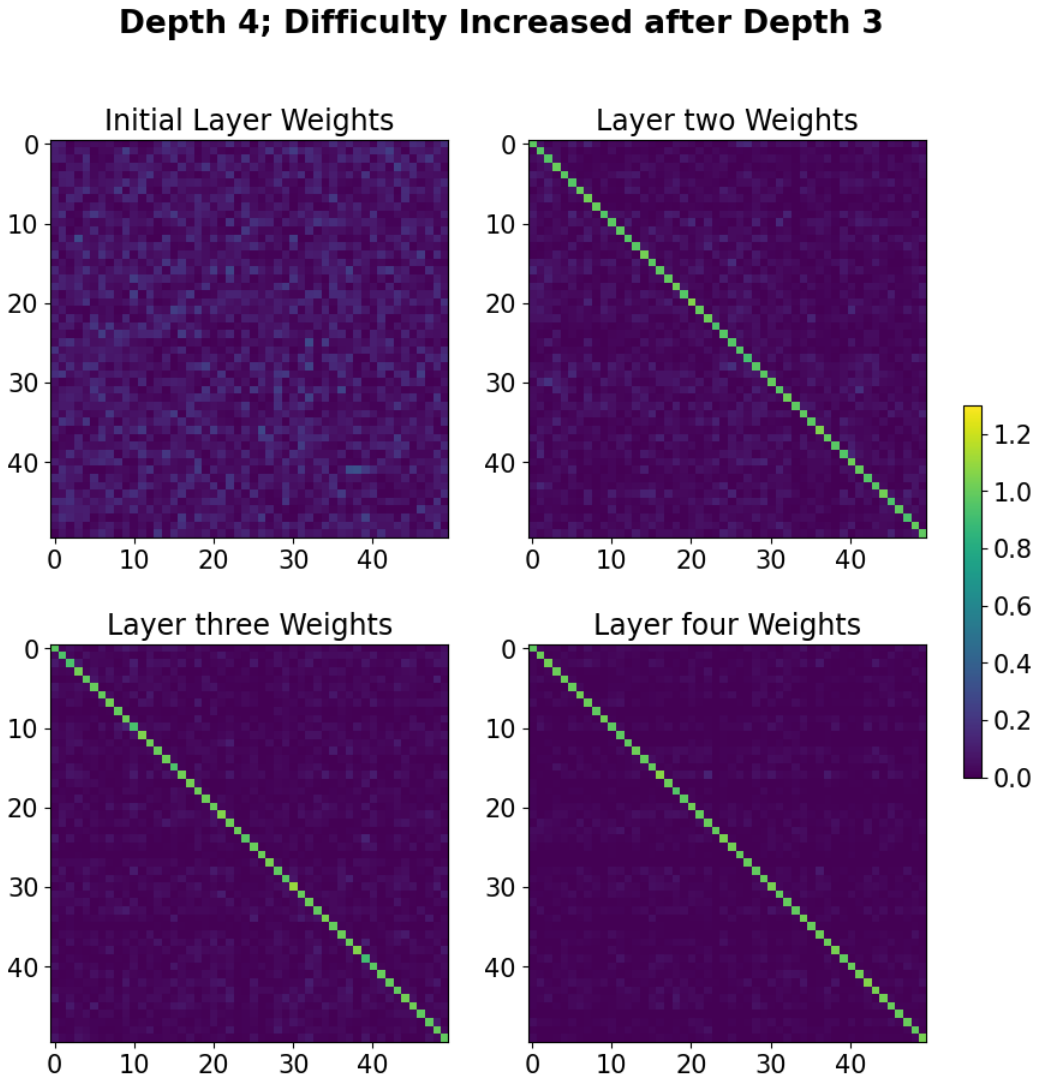


Figure A.13.: Visualisation of a 50 subset of the weights learned inside the feature extractor.

Bibliography

Zaheer Abbas, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C. Machado (2023). "Loss of Plasticity in Continual Deep Reinforcement Learning". In: *Conference on Lifelong Learning Agents, 22-25 August 2023, McGill University, Montréal, Québec, Canada*. Ed. by Sarath Chandar, Razvan Pascanu, Hanie Sedghi, and Doina Precup. Vol. 232. Proceedings of Machine Learning Research. PMLR, pp. 620–636. URL: https://proceedings.mlr.press/v232/abbas23a.html.

Steven Adriaensen, André Biedenkapp, Gresa Shala, Noor H. Awad, Theresa Eimer, Marius Lindauer, and Frank Hutter (2022). "Automated Dynamic Algorithm Configuration". In: *J. Artif. Intell. Res.* 75, pp. 1633–1699. DOI: 10.1613/JAIR.1.13922. URL: https://doi.org/10.1613/jair.1.13922.

Yoshua Bengio, Nicolas Le Roux, Pascal Vincent, Olivier Delalleau, and Patrice Marcotte (2005). "Convex Neural Networks". In: *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, pp. 123–130. URL: https://proceedings.neurips.cc/paper/2005/hash/0fc170ecbb8ff1afb2c6de48ea5343e7-Abstract.html.

James Bergstra and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *J. Mach. Learn. Res.* 13, pp. 281–305. DOI: 10.5555/2503308.2188395. URL: https://dl.acm.org/doi/10.5555/2503308.2188395.

James Bergstra, Daniel Yamins, and David D. Cox (2013). "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 115–123. URL: http://proceedings.mlr.press/v28/bergstra13.html.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang (2019). "Dota 2 with Large Scale Deep Reinforcement Learning". In: *CoRR* abs/1912.06680. arXiv: 1912.06680. URL: http://arxiv.org/abs/1912.06680.

Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, Difan Deng, and Marius Lindauer (2023). "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges". In: *WIREs Data. Mining. Knowl. Discov.* 13.2. DOI: 10.1002/WIDM.1484. URL: https://doi.org/10.1002/widm.1484.

Leo Breiman (2001). “Random Forests”. In: *Mach. Learn.* 45.1, pp. 5–32. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens (2016). “Net2Net: Accelerating Learning via Knowledge Transfer”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1511.05641.

Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas (2018). “Bayesian Optimization in AlphaGo”. In: *CoRR* abs/1812.06855. arXiv: 1812.06855. URL: http://arxiv.org/abs/1812.06855.

Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter (2018). “Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, pp. 1419–1426. DOI: 10.24963/IJCAI.2018/197. URL: https://doi.org/10.24963/ijcai.2018/197.

Karl Cobbe, Oleg Klimov, Christopher Hesse, Taehoon Kim, and John Schulman (2019). “Quantifying Generalization in Reinforcement Learning”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 1282–1289. URL: http://proceedings.mlr.press/v97/cobbe19a.html.

Eimer, Marius Lindauer, and Roberta Raileanu (2023). “Hyperparameters in Reinforcement Learning and How To Tune Them”. In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, pp. 9104–9149. URL: https://proceedings.mlr.press/v202/eimer23a.html.

T. Eimer (2024). URL: https://github.com/automl/hypersweeper.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter (2019a). “Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=ByME42AqK7.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter (2019b). “Neural Architecture Search: A Survey”. In: *J. Mach. Learn. Res.* 20, 55:1–55:21. URL: http://jmlr.org/papers/v20/18-598.html.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry (2020). “Implementation Matters in Deep RL: A Case Study on PPO and TRPO”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. URL: https://openreview.net/forum?id=r1etN1rtPB.

Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu (2018). "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 1406–1415. URL: http://proceedings.mlr.press/v80/espeholt18a.html.

Utku Evci, Bart van Merrienboer, Thomas Unterthiner, Fabian Pedregosa, and Max Vladymyrov (2022). "GradMax: Growing Neural Networks using Gradient Information". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. URL: https://openreview.net/forum?id=qjN4h%5C_wwUO.

Stefan Falkner, Aaron Klein, and Frank Hutter (2018). "BOHB: Robust and Efficient Hyperparameter Optimization at Scale". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 1436–1445. URL: http://proceedings.mlr.press/v80/falkner18a.html.

Aosong Feng and Priyadarshini Panda (2020). "Energy-efficient and Robust Cumulative Training with Net2Net Transformation". In: *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*. IEEE, pp. 1–7. DOI: 10.1109/IJCNN48605.2020.9207451. URL: https://doi.org/10.1109/IJCNN48605.2020.9207451.

Jörg K. H. Franke, Gregor Köhler, André Biedenkapp, and Frank Hutter (2021). "Sample-Efficient Automated Deep Reinforcement Learning". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. URL: https://openreview.net/forum?id=hSjxQ3B7GWq.

Hado van Hasselt (2010). "Double Q-learning". In: *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*. Ed. by John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta. Curran Associates, Inc., pp. 2613–2621. URL: https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger (2018). "Deep Reinforcement Learning That Matters". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, pp. 3207–3214. DOI: 10.1609/AAAI.V32I1.11694. URL: https://doi.org/10.1609/aaai.v32i1.11694.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver (2018). "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, pp. 3215–3222. DOI: 10.1609/AAAI.V32I1.11796. URL: https://doi.org/10.1609/aaai.v32i1.11796.

Ronald A Howard (1960). "Dynamic programming and markov processes." In.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, eds. (2019). *Automated Machine Learning - Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer. ISBN: 978-3-030-05317-8. DOI: 10.1007/978-3-030-05318-5. URL: https://doi.org/10.1007/978-3-030-05318-5.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu (2017). "Population Based Training of Neural Networks". In: *CoRR* abs/1711.09846. arXiv: 1711.09846. URL: http://arxiv.org/abs/1711.09846.

Kevin G. Jamieson and Ameet Talwalkar (2016). "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 240–248. URL: http://proceedings.mlr.press/v51/jamieson16.html.

Donald R. Jones, Matthias Schonlau, and William J. Welch (1998). "Efficient Global Optimization of Expensive Black-Box Functions". In: *J. Glob. Optim.* 13.4, pp. 455–492. DOI: 10.1023/A:1008306431147. URL: https://doi.org/10.1023/A:1008306431147.

Sham M. Kakade and John Langford (2002). "Approximately Optimal Approximate Reinforcement Learning". In: *Machine Learning, Proceedings of the Nineteenth International Conference (ICML 2002), University of New South Wales, Sydney, Australia, July 8-12, 2002*. Ed. by Claude Sammut and Achim G. Hoffmann. Morgan Kaufmann, pp. 267–274.

Ron Kohavi and George H. John (1995). "Automatic Parameter Selection by Minimizing Estimated Error". In: *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning, Tahoe City, California, USA, July 9-12, 1995*. Ed. by Armand Prieditis and Stuart Russell. Morgan Kaufmann, pp. 304–312. DOI: 10.1016/B978-1-55860-377-6.50045-1. URL: https://doi.org/10.1016/b978-1-55860-377-6.50045-1.

Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel (2020). "The NetHack Learning Environment". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/569ff987c643b4bedf504efda8f786c2-Abstract.html.

Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton (2015). "Deep learning". In: *Nat.* 521.7553, pp. 436–444. DOI: 10.1038/NATURE14539. URL: https://doi.org/10.1038/nature14539.

Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Comput.* 1.4, pp. 541–551. DOI: 10.1162/NECO.1989.1.4.541. URL: https://doi.org/10.1162/neco.1989.1.4.541.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar (2017). "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *J. Mach. Learn. Res.* 18, 185:1–185:52. URL: http://jmlr.org/papers/v18/16-558.html.

Wenzhe Li, Hao Luo, Zichuan Lin, Chongjie Zhang, Zongqing Lu, and Deheng Ye (2023). "A Survey on Transformers in Reinforcement Learning". In: *Trans. Mach. Learn. Res.* 2023. URL: https://openreview.net/forum?id=r30yuDPvf2.

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter (2022). "SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization". In: *J. Mach. Learn. Res.* 23, 54:1–54:9. URL: http://jmlr.org/papers/v23/21-0888.html.

Hanxiao Liu, Karen Simonyan, and Yiming Yang (2018). "DARTS: Differentiable Architecture Search". In: *CoRR* abs/1806.09055. arXiv: 1806.09055. URL: http://arxiv.org/abs/1806.09055.

Ilya Loshchilov and Frank Hutter (2016a). "CMA-ES for Hyperparameter Optimization of Deep Neural Networks". In: *CoRR* abs/1604.07269. arXiv: 1604.07269. URL: http://arxiv.org/abs/1604.07269.

Ilya Loshchilov and Frank Hutter (2016b). "CMA-ES for Hyperparameter Optimization of Deep Neural Networks". In: *CoRR* abs/1604.07269. arXiv: 1604.07269. URL: http://arxiv.org/abs/1604.07269.

William S Lovejoy (1991). "A survey of algorithmic methods for partially observed Markov decision processes". In: *Annals of Operations Research* 28.1, pp. 47–65.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman (2017). "Teacher-Student Curriculum Learning". In: *CoRR* abs/1707.00183. arXiv: 1707.00183. URL: http://arxiv.org/abs/1707.00183.

Yingjie Miao, Xingyou Song, John D. Co-Reyes, Daiyi Peng, Summer Yue, Eugene Brevdo, and Aleksandra Faust (2022). "Differentiable Architecture Search for Reinforcement Learning". In: *International Conference on Automated Machine Learning, AutoML 2022, 25-27 July 2022, Johns Hopkins University, Baltimore, MD, USA*. Ed. by Isabelle Guyon, Marius Lindauer, Mihaela van der Schaar, Frank Hutter, and Roman Garnett. Vol. 188. Proceedings of Machine Learning Research. PMLR, pp. 20/1–17. URL: https://proceedings.mlr.press/v188/miao22a.html.

Rupert Mitchell, Martin Mundt, and Kristian Kersting (2023). "Self Expanding Neural Networks". In: *CoRR* abs/2307.04526. DOI: 10.48550/ARXIV.2307.04526. arXiv: 2307.04526. URL: https://doi.org/10.48550/arXiv.2307.04526.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller (2013). "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning". In: *nature* 518.7540, pp. 529–533.

Aditya Mohan, Carolin Benjamins, Konrad Wienecke, Alexander Dockhorn, and Marius Lindauer (2023). "AutoRL Hyperparameter Landscapes". In: *International Conference on Automated Machine Learning, 12-15 November 2023, Hasso Plattner Institute, Potsdam, Germany*. Ed. by Aleksandra Faust, Roman Garnett, Colin White, Frank Hutter, and Jacob R. Gardner. Vol. 224. Proceedings of Machine Learning Research. PMLR, pp. 13/1–27. URL: https://proceedings.mlr.press/v224/mohan23a.html.

Vu Nguyen, Sebastian Schulze, and Michael A. Osborne (2020a). "Bayesian Optimization for Iterative Learning". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/69eba34671b3ef1ef38ee85caae6b2a1-Abstract.html.

Vu Nguyen, Sebastian Schulze, and Michael A. Osborne (2020b). "Bayesian Optimization for Iterative Learning". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/69eba34671b3ef1ef38ee85caae6b2a1-Abstract.html.

Johan S. Obando-Ceron, Ghada Sokar, Timon Willi, Clare Lyle, Jesse Farebrother, Jakob N. Foerster, Gintare Karolina Dziugaite, Doina Precup, and Pablo Samuel Castro (2024). "Mixtures of Experts Unlock Parameter Scaling for Deep RL". In: *CoRR* abs/2402.08609. DOI: 10.48550/ARXIV.2402.08609. arXiv: 2402.08609. URL: https://doi.org/10.48550/arXiv.2402.08609.

Kei Ota, Devesh K. Jha, and Asako Kanezaki (2021). "Training Larger Networks for Deep Reinforcement Learning". In: *CoRR* abs/2102.07920. arXiv: 2102.07920. URL: https://arxiv.org/abs/2102.07920.

Jack Parker-Holder, Raghu Rajan, Xingyou Song, André Biedenkapp, Yingjie Miao, Theresa Eimer, Baohe Zhang, Vu Nguyen, Roberto Calandra, Aleksandra Faust, Frank Hutter, and Marius Lindauer (2022). "Automated Reinforcement Learning (AutoRL): A Survey and Open Problems". In: *J. Artif. Intell. Res.* 74, pp. 517–568. DOI: 10.1613/JAIR.1.13596. URL: https://doi.org/10.1613/jair.1.13596.

Chau Pham, Piotr Teterwak, Soren Nelson, and Bryan A. Plummer (2024). "MixtureGrowth: Growing Neural Networks by Recombining Learned Parameters". In: *IEEE/CVF Winter Conference on Applications of Computer Vision, WACV 2024, Waikoloa, HI, USA, January 3-8, 2024*. IEEE, pp. 2788–2797. DOI: 10.1109/WACV57701.2024.00278. URL: https://doi.org/10.1109/WACV57701.2024.00278.

Hong Qian and Yang Yu (2021). "Derivative-free reinforcement learning: a review". In: *Frontiers Comput. Sci.* 15.6, p. 156336. DOI: 10.1007/S11704-020-0241-4. URL: https://doi.org/10.1007/s11704-020-0241-4.

Antonin Raffin (2020). *RL Baselines3 Zoo*. https://github.com/DLR-RM/rl-baselines3-zoo.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann (2021). "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *J. Mach. Learn. Res.* 22, 268:1–268:8. URL: http://jmlr.org/papers/v22/20-1364.html.

Carl Edward Rasmussen and Christopher K. I. Williams (2006). *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press. ISBN: 026218253X. URL: https://www.worldcat.org/oclc/61285753.

John R. Rice (1976). "The Algorithm Selection Problem". In: *Adv. Comput.* 15, pp. 65–118. DOI: 10.1016/S0065-2458(08)60520-3. URL: https://doi.org/10.1016/S0065-2458(08)60520-3.

Frederic Runge, Danny Stoll, Stefan Falkner, and Frank Hutter (2019). "Learning to Design RNA". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=ByfyHh05tQ.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel (2021). "MiniHack the Planet: A Sandbox for Open-Ended Reinforcement Learning Research". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*. Ed. by Joaquin Vanschoren and Sai-Kit Yeung. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Abstract-round1.html.

René Sass, Eddie Bergman, André Biedenkapp, Frank Hutter, and Marius Lindauer (2022). "DeepCAVE: An Interactive Analysis Tool for Automated Machine Learning". In: *CoRR* abs/2206.03493. DOI: 10.48550/ARXIV.2206.03493. arXiv: 2206.03493. URL: https://doi.org/10.48550/arXiv.2206.03493.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz (2015). "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel (2016). "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347. arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

Max Schwarzer, Johan Samir Obando-Ceron, Aaron C. Courville, Marc G. Bellemare, Rishabh Agarwal, and Pablo Samuel Castro (2023). "Bigger, Better, Faster: Human-level Atari with human-level efficiency". In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, pp. 30365–30380. URL: https://proceedings.mlr.press/v202/schwarzer23a.html.

Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty (2023). "Reinforcement learning algorithms: A brief survey". In: *Expert Syst. Appl.* 231, p. 120495. DOI: 10.1016/J.ESWA.2023.120495. URL: https://doi.org/10.1016/j.eswa.2023.120495.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587, pp. 484–489. DOI: 10.1038/NATURE16961. URL: https://doi.org/10.1038/nature16961.

Samarth Sinha, Homanga Bharadhwaj, Aravind Srinivas, and Animesh Garg (2020). "D2RL: Deep Dense Architectures in Reinforcement Learning". In: *CoRR* abs/2010.09163. arXiv: 2010.09163. URL: https://arxiv.org/abs/2010.09163.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, pp. 2960–2968. URL: https://proceedings.neurips.cc/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html.

Ghada Sokar, Rishabh Agarwal, Pablo Samuel Castro, and Utku Evci (2023). "The Dormant Neuron Phenomenon in Deep Reinforcement Learning". In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, pp. 32145–32168. URL: https://proceedings.mlr.press/v202/sokar23a.html.

Charles Spearman (1961). "The proof and measurement of association between two things." In.

Richard S Sutton and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Emanuel Todorov, Tom Erez, and Yuval Tassa (2012). "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*. IEEE, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109. URL: https://doi.org/10.1109/IROS.2012.6386109.

Tanja Tornede, Alexander Tornede, Lukas Fehring, Lukas Gehring, Helena Graf, Jonas Hanselle, Felix Mohr, and Marcel Wever (2023). "PyExperimenter: Easily distribute experiments and track results". In: *J. Open Source Softw.* 8.86, p. 5149. DOI: 10.21105/JOSS.05149. URL: https://doi.org/10.21105/joss.05149.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit Evciand Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Xingchen Wan, Cong Lu, Jack Parker-Holder, Philip J. Ball, Vu Nguyen, Binxin Ru, and Michael A. Osborne (2022). "Bayesian Generational Population-Based Training". In: *International Conference on Automated Machine Learning, AutoML 2022, 25-27 July 2022, Johns Hopkins University, Baltimore, MD, USA*. Ed. by Isabelle Guyon, Marius Lindauer, Mihaela van der Schaar, Frank Hutter, and Roman Garnett. Vol. 188. Proceedings of Machine Learning Research. PMLR, pp. 14/1–27. URL: https://proceedings.mlr.press/v188/wan22a.html.

Christopher John Cornish Hellaby Watkins (1989). "Learning from delayed rewards". PhD thesis. King's College, Cambridge United Kingdom.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen (2016). "Network Morphism". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 564–572. URL: http://proceedings.mlr.press/v48/wei16.html.

Wei Wen, Feng Yan, Yiran Chen, and Hai Li (2020). "AutoGrow: Automatic Layer Growing in Deep Convolutional Networks". In: *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*. Ed. by Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash. ACM, pp. 833–841. DOI: 10.1145/3394486.3403126. URL: https://doi.org/10.1145/3394486.3403126.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter (2023). "Neural Architecture Search: Insights from 1000 Papers". In: *CoRR* abs/2301.08727. DOI: 10.48550/ARXIV.2301.08727. arXiv: 2301.08727. URL: https://doi.org/10.48550/arXiv.2301.08727.

Lemeng Wu, Dilin Wang, and Qiang Liu (2019). "Splitting Steepest Descent for Growing Neural Architectures". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, pp. 10655–10665. URL: https://proceedings.neurips.cc/paper/2019/hash/3a01fc0853ebeba94fde4d1cc6fb842a-Abstract.html.

Barret Zoph and Quoc V. Le (2017). "Neural Architecture Search with Reinforcement Learning". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=r1Ue8Hcxg.

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.