Continual Reinforcement Learning by Merging Models

Bachelor thesis by Aleksandar Tatalović
Date of submission: October 30, 2023

1. Review: Dr. Martin Mundt
2. Review: Quentin Delfosse
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Technical University of Darmstadt

Artificial Intelligence and Machine Learning Lab

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

Hiermit erkläre ich, Aleksandar Tatalović, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

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

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

Darmstadt, October 30, 2023

A. Tatalović

A. Tatalović

Abstract

In deep learning, there are several model merging procedures that combine multiple models to create a new model that inherits the capabilities of each source model. In previous works, these procedures were exclusively used in deep supervised learning (SL), producing well-performing solutions. However, the usage of these procedures is not restricted to deep SL, as they can merge any models in deep learning that meet their criteria. We find that models in deep reinforcement learning (RL) meet these criteria. Recently developed merging procedures in SL accomplished to merge two models trained on disjoint datasets (that included the same classes) to create a new model that performed well on the combined dataset. Based on this accomplishment, we hypothesize that model merging in deep RL can enable modular continual RL, which means learning of a RL task by merging models trained on simpler sub-tasks created from the task. Here, we show that this hypothesis is incorrect. We assumed that merging similar features of RL models trained on sub-tasks of the same task would lead to the desired behavior of the merged solution in the original task. However, we found this is not necessarily the case. Not only the similarity of features is substantial, but also the meaning of their activations in the context of the sub-task. They can mean entirely different things in different sub-tasks and thus lead to different actions. This is not the case in deep SL, and the merging procedures do not deal with that issue when applied in deep SL. Our results demonstrate that model merging in our modular continual RL scenario is inherently more complex than merging models trained on disjoint datasets (that include the same classes) in deep SL. Even though we showed that our hypothesis is incorrect, we gained relevant insights about model merging in continual RL nonetheless. These insights can be researched in the future to determine what changes would have to be made to enable continual RL through the model merging procedures we considered or whether such changes can be made in the first place.

Contents

1 Introduction **5**

2 Background **6**

2.1 Continual Deep Reinforcement Learning . . . 6

2.1.1 Reinforcement Learning . . . 6

2.1.2 Deep Learning . . . 7

2.1.3 Continual Learning . . . 8

2.1.4 Continual Reinforcement Learning . . . 9

2.2 Proximal Policy Optimization . . . 10

3 From Merging Models in Supervised Learning to Continual Reinforcement Learning **12**

3.1 Related Work in Model Merging . . . 12

3.1.1 Weight Averaging . . . 13

3.1.2 Model Soups . . . 13

3.1.3 Fisher Merging . . . 14

3.1.4 Git Re-Basin . . . 15

3.1.5 Other Procedures . . . 19

3.2 Bridging the Gap to Continual Reinforcement Learning Scenarios . . . 20

3.3 Potential of Existing Merging Procedures . . . 22

3.4 Proposed Methodology . . . 26

4 Experiments **29**

4.1 Setup . . . 29

4.1.1 Environments . . . 29

4.1.2 Model Architecture and Learning Algorithm . . . 33

4.1.3 Model Training . . . 35

4.2 Results . . . 38

4.2.1 Starpilot . . . 38

4.2.2 Fruitbot . . . 41

5 Conclusion & Future Work **44**

1 Introduction

In recent years, several works have appeared that merged multiple deep learning models into one model that successfully combined their capabilities (Wortsman, Ilharco, Gadre, et al., 2022; Matena et al., 2021; Pasen et al., 2022). While the motivations for merging the models differed between these several works, they all focused on merging models in deep supervised learning specifically. The models were merged according to different merging procedures, and new procedures were introduced that improved upon previous procedures. All procedures require the models to meet specific criteria before merging them into a model that successfully combines their capabilities. We believe the procedures can enjoy similar success outside of deep supervised learning, provided the models they merge meet their criteria. We find that models in deep reinforcement learning meet these criteria, which indicates that the procedures could enjoy similar success in deep reinforcement learning.

Based on the recent accomplishments of newly introduced merging procedures (Ainsworth et al., 2022), we went one step further. We hypothesized that model merging procedures could be utilized in continual reinforcement learning scenarios, where a more complex RL task is learned by merging models trained on simpler sub-tasks created from the more complex task. If true, this is attractive to the field of continual reinforcement learning, as most current approaches consider a scenario where many complex tasks are learned sequentially and not a scenario where complex tasks are learned from related simpler tasks (Traoré et al., 2019; Atkinson et al., 2021; Kaplanis et al., 2019). This would add a simple way for continual reinforcement learning methods to incorporate modular learning of complex tasks, thus advancing the capabilities of the methods.

The newly introduced merging procedures that we based our hypothesis on merged models that were trained on disjoint datasets (that included the same classes) in supervised learning, and the resulting model outperformed both source models on the combined dataset, whereby the two disjoint datasets were created by splitting the combined dataset in the first place. We investigated the correctness of our hypothesis by crafting a similar scenario in reinforcement learning. We created two sub-tasks from one RL task. Then, we trained models on each sub-task and merged models that were trained on different sub-tasks.

This thesis presents a first attempt at combining model merging in deep learning with continual reinforcement learning and reports the insights gained. Moreover, it discusses what results can be expected of existing merging procedures.

The thesis is structured as follows. Chapter 2 provides the reader with background information pertinent to our work. It includes descriptions of relevant machine learning concepts. The chapter also explains how this information relates to later sections and why it proves pertinent. Chapter 3 introduces the reader to related works in model merging and explains individual merging procedures. It explains our underlying theories, which led us to hypothesize that model merging can be utilized in continual reinforcement learning scenarios. It also introduces our proposed methodology, which is meant to examine the correctness of our hypothesis. Chapter 4 discusses the results of our experiments and interprets them. Lastly, chapter 5 concludes our work by putting our findings into a broader perspective and discussing possible future works to gain further insights.

2 Background

2.1 Continual Deep Reinforcement Learning

In machine learning (ML), numerous learning paradigms have evolved (Emmert-Streib et al., 2022), and scientific works on ML often cannot be assigned to just one paradigm but lie at an intersection of multiple paradigms. Our work lies at an intersection of reinforcement learning, deep learning, and continual learning. This section clarifies what these individual paradigms entail and how they intersect. We additionally highlight the differences between reinforcement learning and supervised learning, as understanding these differences becomes crucial in the next chapter. We also elaborate on our understanding of ML concepts relevant to our work. This is necessary because many ML concepts are not clearly defined, and different interpretations can be found in scientific works.

2.1.1 Reinforcement Learning

We begin with a description of reinforcement learning (RL), based on the contents found in the book by Sutton et al. (2018). In RL, the learning algorithm learns by interacting with an environment through actions, which is why the learning algorithm is also called the agent. The agent must be able to observe the current state of the environment and take actions that can affect the state. We refer to the set of all possible actions the agent can take as the action space and to the set of all possible environment states as the state space. Actions are taken to achieve a goal, and a reward signal formalizes the goal. That means a reward is passed from the environment to the agent after every action taken, whereby the reward can be positive, negative, or zero. A reward of zero can be counted as no reward received. The reward signal is designed such that maximizing the cumulative sum of the received rewards, called the return, leads to achieving the goal. Thus, RL methods are designed to find optimal actions that maximize the expected return. The agent can take an action at each timestep. If t is the current timestep, we denote the rewards the agent receives after timestep t as $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$. The return can be defined differently, depending on the task the agent tries to solve. The task is for the agent to achieve the goal within a specific environment.

Suppose the agent-environment interaction can be split into subsequences, each having a clear beginning and end. In that case, the task to solve in the environment is called an episodic task, and the subsequences are called episodes (and the environment is called an episodic environment). Each episode ends in a terminal state. A video game consisting of distinct rounds the player must complete would be an episodic task. There exist other types of tasks in RL. However, we limit our description of RL to the episodic case, as we focus exclusively on episodic game-like environments in later chapters and use them in our experiments. In episodic tasks, the return G at timestep t for one episode is defined as the discounted sum of rewards:

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \tag{2.1}$$

where T denotes the final time step of the episode, and γ is the discounting factor within $[0, 1]$. The rationale behind the discounting factor is that rewards received in the future are worth less than rewards received in the present. The agent acts according to the policy it follows, which is a mapping from states to actions that can be either stochastic or deterministic. In the stochastic case, we denote the policy as $\pi(a|s)$. A policy is referred to as optimal policy if it maps each state to the best possible (optimal) action, meaning the action that maximizes the expected return. Two essential functions in RL are the state-value function and the action-value function. The state-value function under a policy π, denoted $V^\pi(s)$, receives a state s as input and maps it to the expected return when starting in s and following π afterward. Similarly, the action-value function under a policy π, denoted $Q^\pi(s, a)$, receives a state s and an action a as inputs and maps them to the expected return when taking action a in state s and following π afterward. Policies and value functions are often estimated by ML models (function approximators), and RL algorithms use the predictions of the value function's model to evaluate the policy model. The difference between the action-value and state-value function under a policy π is defined as the advantage function:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s). \tag{2.2}$$

This function helps to assess the advantage of choosing one action over others, as it represents the difference between the estimated value of taking a specific action in a given state and the expected value of following the current policy in that state.

We now discuss the differences between RL and supervised learning (SL). We derive the information on these differences from the book by (Bishop, 2007).
RL and SL are both considered to be traditional ML paradigms. Learning algorithms in SL typically train a model on a labeled dataset and aim to reduce the error between the model's predictions and the actual optimal outputs. Verifying whether the model's predictions are false or correct is straightforward because the optimal outputs are available in a labeled dataset. This process is entirely different from how agents learn in RL. The agent must determine the optimal actions (outputs) through trial and error. Moreover, it is not straightforward to determine whether one action is better than another, as rewards can be zero for many steps, no matter the action, and even if an action leads to an immediate positive reward, that does not ensure the action maximizes the return. In SL, a model output can only be correct or false (if the input comes from a labeled dataset). RL and SL are fundamentally different because they are trying to learn different tasks. There are no absolute definitions of a task in ML, so we clarify what we consider tasks. Common tasks in SL are classification tasks, where the goal is for the trained model to categorize input data into predefined classes. The dataset shapes a classification task. Different datasets can correspond to the same classification task when they categorize data points into the same classes, even if the datasets differ. Conversely, a single dataset can encompass multiple classification tasks. Common tasks in RL are episodic game-like tasks, which are profoundly disparate from classification tasks. Episodic RL tasks depend on the reward structure, the environment states, and the possible actions. Changing one of the three can constitute a new task, depending on the impact of the change. Of course, there are many other types of tasks in ML, but these are particularly relevant to our work. Performance in episodic RL tasks is typically measured in average episodic return. In classification tasks, performance is measured on a specific dataset used only for testing, referred to as the validation set. The test loss or accuracy of the model on the validation set are used to quantify performance.

2.1.2 Deep Learning

We now describe deep learning based on the contents found in the book by Prince (2023).
The previous section mentions that ML methods use models to approximate functions and generate predictions.

One type of ML model is deep neural networks (DNNs), which are neural networks with more than one hidden layer of units. At the time of writing, they are the most practical and capable ML models, making them widely used. Deep learning (DL) refers to optimizing a DNN's weights to solve a task in any ML scenario. This process is also referred to as training of the model. It is not a stand-alone paradigm because it mainly refers to the used model type. Whether DNNs are trained in SL or RL, both scenarios are considered DL, even though SL and RL are different learning paradigms. Deep RL was first introduced by Mnih, Kavukcuoglu, Silver, Graves, et al. (2013) and refers to the usage and training of DNNs in RL algorithms. No matter the scenario, DL algorithms involve some form of optimization problem usually solved by a stochastic gradient descent algorithm. This leads to different DL scenarios, like deep RL and deep SL, displaying similar characteristics during model training. That fact becomes important in section 3.2. DL and its properties are vital for our work, as the models we work with and those used in related work are solely DNNs. We use the terms neural network and DNN interchangeably.

2.1.3 Continual Learning

Classical ML algorithms, as described in the prior sections, optimize model weights in a training phase, after which learning ceases. If that model were to be trained on a new task afterward, using the same algorithm, the knowledge of the previously learned task would be overwritten and unutilized in the process. Chen et al. (2018) refer to this as isolated learning. On the other hand, continual learning (CL), an increasingly prominent area of modern machine learning (Khetarpal et al., 2020), encompasses a learning process in which learning never ceases and allows for learning multiple tasks sequentially without forgetting the previously learned tasks. The forgetting of previously learned tasks in ML is called catastrophic inference or catastrophic forgetting (McCloskey et al., 1989). Existing definitions of CL (Thrun, 1996; Chen et al., 2018; Mundt et al., 2023) are unspecific and abstract. In addition, there are no clear definitions of tasks in ML, and many related ML paradigms have evolved over the last years. Therefore, it is difficult to determine when precisely something counts as CL. The existence of multiple terms for CL in scientific literature, such as lifelong learning, continuous learning, and non-stationary learning, reflects this obscurity. Mundt et al. (2023) provide an insightful perspective on CL. According to them, a precise definition of CL could be counterproductive, as it may lead to an overly narrow view of the field and result in the loss of relevant concepts. Instead, CL should address diverse research inquiries and be considered a superset of related paradigms. Some related paradigms are transfer learning, multi-task learning, online learning, few-shot learning, curriculum learning, open world learning, meta learning, and federated learning. Transfer learning aims to transfer the knowledge of one model to another model (Zhuang et al., 2019) and is particularly relevant to our work, as many existing model merging procedures are utilized in transfer learning. Chen et al. (2018) propose that CL may require a systems approach, meaning that multiple learning algorithms and components must be combined to achieve the goals of CL. That being said, there is no clear definition of what the goals are in CL. From the unspecific and abstract definitions mentioned earlier, we summarize popular goals for a CL system:

- The system can sequentially learn different tasks.
- Previously learned knowledge is maintained and not forgotten (avoidance of catastrophic forgetting).
- Forward transfer: Past knowledge is utilized to accelerate learning a new task.
- Backward transfer: New knowledge is utilized to improve performance on old tasks.
- The system is capable of discovering new tasks.
- The system can learn on the job (learn online).

- The system must identify unseen, unknown data instances and reject those or use them for later learning.
- The system chooses which data to incorporate (active learning) and builds its curriculum.

Existing CL algorithms generally only try to achieve part of these goals.

2.1.4 Continual Reinforcement Learning

The isolated learning paradigm described earlier also applies to classical RL algorithms. Therefore, CL is researched in combination with RL, which is termed continual reinforcement learning (CRL). Abel et al. (2023) state: ”We tend to concentrate on agents that learn to solve problems, rather than agents that learn forever.” They phrase the goal of CRL to be the creation of forever learning RL agents, and they advertise the pursuit of RL agents who learn skills rather than the solution to one specific task. Interestingly, the first scientific works on CRL emerged roughly simultaneously as works on general CL (Thrun and Mitchell, 1995; Ring, 1997). There are no clear definitions of CRL, but multiple perspectives exist. The initial motivation for CRL was to solve the issue that RL agents require extensive experience to learn (meaning many interactions with the environment) (Thrun and Mitchell, 1995; Khetarpal et al., 2020). CRL is supposed to accelerate learning by incorporating previously acquired knowledge (forward transfer). It evolved into a separate field with its own goals and motivations. General CL work, as described in 2.1.3, tends to focus on SL scenarios (Ben-Iwhiwhu et al., 2022). RL's unique objectives and requirements explain CRL's development into a separate field. Popular goals for CRL agents are similar to the ones listed in 2.1.3, but there are some RL specific additions: The *ideal* CRL agent should learn skills (behaviors) while solving tasks. Those skills should be extendable in the future and serve as building blocks for more complex skills (hierarchical learning). It should adapt to changes and learn incrementally, meaning there is no fixed training set, and learning occurs at each time step.

Due to recent accomplishments in deep RL, many current CRL methods work in combination with deep RL. We refer to that combination as continual deep reinforcement learning. Most of these methods consider a task sequential setting and focus mainly on eliminating catastrophic forgetting. Original motivations for CRL, like combating the extensive required experience to learn and hierarchical learning of skills are less emphasized by current methods. Rolnick et al. (2018) developed CLEAR, which trains an agent on samples generated by the current policy while incorporating past experience generated by older policies. The agent is exposed to new tasks sequentially, and by incorporating past experience, the method keeps memory of the old tasks. CLEAR manages to reduce catastrophic forgetting significantly. Ben-Iwhiwhu et al. (2022) adjusted a policy function modeled by a DNN depending on the current task. They used one policy network to learn multiple tasks sequentially. They kept task specific configurations for the policy saved in a modulating mask, which is a structure that enables modification of neural networks based on certain conditions (Mallya et al., 2018; Wortsman, Ramanujan, et al., 2020). The configurations were applied depending on the task, which allowed the method to circumvent catastrophic forgetting. More CRL methods that combat catastrophic forgetting can be found in (Traoré et al., 2019; Atkinson et al., 2021; Kaplanis et al., 2018; Kaplanis et al., 2019). Mendez et al. (2022) proposed a different approach to CRL that can be considered to incorporate hierarchical learning. They explored functional compositionality for CRL. They decomposed RL tasks into subproblems where the output of one subproblem becomes the input of others. Their approach is based on modular neural networks, where each modular network contains some individual capabilities. Each subproblem is solved by learning the ideal combination of the modular networks and improving them afterward. This process is repeated iteratively. Their approach of decomposing RL tasks into subproblems and learning the original task in a modular way is similar to the approach we describe in our proposed methodology (3.4).

2.2 Proximal Policy Optimization

We now introduce the reader to the RL algorithm we use in our experiments to train models on episodic game-like tasks. Before we get into the specific method, we give context on RL methods in general. We describe RL methods based on the definitions and explanations in the books by Dong et al. (2020) and Sutton et al. (2018).

RL algorithms are divided into several categories, the main distinction being model-based and model-free, which means algorithms that work with or learn a model of the environment and those that do not. Model-free algorithms are further divided into value-based and policy-based algorithms. Value-based algorithms aim to optimize a value function, usually the action-value function $Q^{\pi}(s,a)$, and then use the optimized result to approximate the optimal policy $\pi^* \approx \arg\max_{\pi} Q^{\pi}$. A notable value-based algorithm is Deep Q-Network (DQN), developed by Mnih, Kavukcuoglu, Silver, Rusu, et al. (2015), which uses two DNNs to approximate the Q-values of actions through experience. Policy-based algorithms, on the other hand, optimize the policy directly through iterative updates of a parameterized policy function $\pi_{\boldsymbol{\theta}}(a|s) = \pi(a|s,\boldsymbol{\theta})$, where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ is the policy function's parameter vector.

Policy-gradient methods form a subcategory of policy-based algorithms. These methods introduce a performance measure $L(\boldsymbol{\theta})$, also called the objective function, to evaluate the current policy $\pi_{\boldsymbol{\theta}}$. The gradient of the objective with respect to the weights $\boldsymbol{\theta}$ is termed the policy gradient ($\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$). Improving the policy is equivalent to maximizing the objective with respect to $\boldsymbol{\theta}$. Therefore, the weights are updated through iterative application of stochastic gradient ascent on an estimate $\hat{g}_k \approx \nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{\theta}_k)$ of the policy gradient:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \hat{g}_k, \tag{2.3}$$

where the learning rate α is used to control the update, and k denotes the current iteration. All policy gradient methods are based on this schema. Some policy gradient methods optimize a parameterized value function (typically the state-value function) alongside the parameterized policy function. This value function is incorporated into the objective and assists in evaluating the policy. Methods that follow this approach are called actor-critic methods, with the actor referring to the policy function and the critic to the value function.

Proximal policy optimization (PPO), a policy gradient actor-critic method developed by Schulman et al. (2017), is a popular RL algorithm we utilize in our work. It is easy to implement and requires less computational resources than other options because it does not store previously encountered states in memory during training. It also improves the shortcomings of other policy gradient methods.

PPO's objective function combines three terms:

$$L^{PPO}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \hat{\mathbb{E}}\left[L^{CLIP}(\boldsymbol{\theta}) - c_1 L^{VF}(\boldsymbol{\phi}) + c_2 S[\pi_{\boldsymbol{\theta}}](s_t)\right], \tag{2.4}$$

where t is one timestep, c_1 and c_2 are coefficients, and S denotes an entropy bonus that ensures exploration during training by increasing the uncertainty in the agent's predictions. Exploration in RL refers to the agent's strategy of actively seeking and trying out new actions to discover better policies and avoid getting stuck in a local optimum. PPO ensures exploration through the entropy bonus and its stochastic actor, which predicts probabilities of which actions are best to take rather than deterministically selecting actions. We first explain

$$L^{CLIP}(\boldsymbol{\theta}) = \hat{\mathbb{E}}\left[min(r_t(\boldsymbol{\theta})\hat{A}_{\boldsymbol{\phi}}^{\pi_{\boldsymbol{\theta}_{old}}}(s_t,a_t), clip(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)\hat{A}_{\boldsymbol{\phi}}^{\pi_{\boldsymbol{\theta}_{old}}}(s_t,a_t))\right], \tag{2.5}$$

and

$$r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t|s_t)}. \tag{2.6}$$

Here, $\boldsymbol{\theta}_{old}$ is the parameter vector *before* the update, and the goal is to maximize the policy ratio $r_t(\boldsymbol{\theta})$ multiplied by an estimate of the advantage function $\hat{A}_{\boldsymbol{\phi}}^{\pi_{\boldsymbol{\theta}_{old}}}$. L^{CLIP} maximizes both the regular product and a clipped version, which constraints the policy ratio within $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter that must be set beforehand. Then, the minimum of the regular and the clipped version is chosen as an update to avoid destructively large update sizes. The advantage function is estimated as in (Mnih, Badia, et al., 2016):

$$\hat{A}_{\boldsymbol{\phi}}^{\pi_{\boldsymbol{\theta}_{old}}}(s_t, a_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k \hat{V}_{\boldsymbol{\phi}}(s_{t+k}) - \hat{V}_{\boldsymbol{\phi}}(s_t), \tag{2.7}$$

where $\hat{V}_{\boldsymbol{\phi}}$ is a parameterized state-value function (the critic) that estimates $V^{\pi_{\boldsymbol{\theta}_{old}}}$. PPO updates $\hat{V}_{\boldsymbol{\phi}}$ every iteration by minimizing the mean-squared error between the estimated return and the actual return received after the policy has been run to collect samples:

$$L^{VF}(\boldsymbol{\phi}) = (\hat{V}_{\boldsymbol{\phi}}(s_t) - G_t)^2. \tag{2.8}$$

Maximizing L^{PPO} maximizes L^{CLIP} and minimizes L^{VF} at the same time due to L^{VF} being subtracted from L^{CLIP}. If DNNs model both $\pi_{\boldsymbol{\theta}}$ and $\hat{V}_{\boldsymbol{\phi}}$, they usually share the feature extraction layers, as they need to compute the same features. PPO runs the policy for T timesteps (another hyperparameter) in a single iteration of the algorithm. Then, the objective is maximized for every single timestep t within $[0, T]$, and finally, the average over all samples is used to update the weights.

3 From Merging Models in Supervised Learning to Continual Reinforcement Learning

This chapter's first section portrays our understanding of model merging based on related works. We also explain individual merging procedures and mention the scenarios in which these procedures were applied. In section 3.2, we present our hypotheses which let us combine model merging with CRL. Afterward, in section 3.3, we discuss the potential of individual merging procedures and in which scenarios we expect them to deliver which results. We conclude the chapter in section 3.4, constructing a methodology that serves to investigate our hypotheses and expectations.

3.1 Related Work in Model Merging

By model merging, we mean the merging of neural networks, which typically refers to combining multiple neural networks in some way to leverage their capabilities. Neural network merging is utilized in various subfields of ML, e.g. transfer learning, neural architecture search (X. Wang et al., 2020), and federated learning (Mensah et al., 2022). Depending on the subfield and purpose, the models are merged differently. We are interested in merging procedures that merge multiple models into one new model, transferring the capabilities of all source models to the resulting model as well as possible. We find such procedures primarily in scientific works on fine-tuning and feature matching (or neural alignment). Fine-tuning refers to the widespread process of taking a pre-trained base model and further training it on a dataset belonging to the target task (Howard et al., 2018; Min et al., 2017). It is considered to be a subfield of transfer learning. We present multiple existing procedures that vary in their complexity and fit together thematically. The covered procedures require the source models to have the same architecture and ensure that the resulting model also has that architecture. In the following subsections, we explain each procedure separately, going from least complex to most complex, and mention the scenarios in which they were utilized. We also mention two additional merging procedures in 3.1.5 that do not thematically fit with the other procedures. We do this to showcase other approaches to merging, but we do not consider them for our experiments, as we want to focus on one merging theme.

To explain the procedures, we consider all models to be L-layer multilayer perceptrons (MLP) (Bishop, 2007),

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{z}_{L+1}, \quad \boldsymbol{z}_{l+1} = \sigma(\boldsymbol{W}_l \boldsymbol{z}_l + \boldsymbol{b}_l), \quad \boldsymbol{z}_1 = \boldsymbol{x}, \tag{3.1}$$

where σ denotes an element-wise nonlinear activation function. We consider MLPs due to their simplicity, but the merging procedures can be applied to almost arbitrary model architectures.

3.1.1 Weight Averaging

The least complex procedure merges N neural networks by averaging their weights (when referring to weights we include the bias terms):

$$f_{merged}\left(\boldsymbol{x}; \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{\theta}_i\right). \tag{3.2}$$

Weight averaging is feasible as long as the models have the same architecture. It is possible to include model-specific weighting parameters λ_i, to control each model's significance in the averaging process. This changes the equation 3.2 to

$$f_{merged}\left(\boldsymbol{x}; \sum_{i=1}^{N}\lambda_i\boldsymbol{\theta}_i\right), \tag{3.3}$$

where $\lambda_i \geq 0$ and $\sum_i^N \lambda_i = 1$. If we consider all models equally important, we set $\lambda_i = \frac{1}{N}$, which is equal to weight averaging without weighting parameters. Merging procedures in the following subsections all involve weight averaging in some form (except the additional ones in 3.1.5).

Weight averaging was utilized in multiple scientific works. Wortsman, Ilharco, Gadre, et al. (2022) applied it to models trained in image and text classification tasks. They merged multiple fine-tuned models into one and showcased increased validation set accuracy compared to the typical approach of selecting the best-performing fine-tuned model. The models were fine-tuned from the same weight initialization and on the same datasets but with different hyperparameter configurations. Moreover, Wortsman, Ilharco, M. Li, et al. (2021) demonstrated that the robustness to domain shifts in fine-tuned models can be improved by computing a weighted average between the weights of the original pre-trained model and those of the fine-tuned model. The FedAvg algorithm used in federated learning (McMahan et al., 2017) averages the updated weights of individual worker models training on separate datasets to update the weights of a shared model. The algorithm requires the worker models to share the same initialization of weights and that their separate datasets belong to the same task.

3.1.2 Model Soups

Wortsman, Ilharco, Gadre, et al. (2022) approach of fine-tuning models from the same initialization but with different hyperparameter configurations revealed that a model with a disadvantageous hyperparameter configuration sometimes ended up with particularly low accuracy compared to the other models. In that case, the model with the low accuracy overly decreased the performance of the merged model. To circumvent this issue, they invented greedy soup and learned soup, two merging procedures that achieved better results than weight averaging. They refer to merged models as model soups, which explains the naming of the two procedures.

Greedy Soup

The idea behind greedy soup is simple: Sequentially add models to weight averaging and evaluate the averaged outcome each time after a model has been added. If the resulting model's performance decreases, undo the previous averaging and exclude that model afterward. Before merging, the N models are sorted in decreasing order based on performance (in the model soups paper, performance means validation set accuracy as they do supervised learning, but performance could be measured differently in other scenarios). Merging begins with the best-performing model, which ensures that the merged result never performs worse than the best-performing single model. The greedy soup procedure is displayed in algorithm 1.

Algorithm 1 Greedy Soup (Wortsman, Ilharco, Gadre, et al., 2022)

```
Input: Model weights $\{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_N\}$ // sorted in decreasing order of perf($\boldsymbol{\theta}_i$)
models_to_merge ← {}
// perf($\boldsymbol{\theta}_i$) computes the performance of $\boldsymbol{\theta}_i$ and avg($M$) averages the weights in set $M$
for $i = 1$ to $N$ do
    if perf(avg(models_to_merge ∪ $\{\boldsymbol{\theta}_i\}$)) ≥ perf(avg(models_to_merge)) then
        models_to_merge ← models_to_merge ∪ $\{\boldsymbol{\theta}_i\}$
    end if
end for
return avg(models_to_merge)
```

Learned Soup

Learned soup is a more advanced procedure that overcomes the sequential constraint from greedy soup. It requires a labeled validation set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$. It minimizes the following objective through gradient-based minibatch optimization:

$$\underset{\alpha\in\mathbb{R}^N,\beta\in\mathbb{R}}{\arg\min} \sum_{j=1}^{n} \mathcal{L}\left(\beta \cdot f\left(\boldsymbol{x}_j; \sum_{i=1}^{N} \alpha_i \boldsymbol{\theta}_i\right), y_j\right), \tag{3.4}$$

where $\mathcal{L}$ is some loss function. The coefficients α and β are learned in the process; hence, *learned* soup. Unlike the sequential greedy soup procedure, this procedure requires only a single pass through the validation set. The drawback is that learned soup requires all models to be loaded in memory at the same time. Learned soup can be modified to learn a separate α for each layer, which the creators call the ”by layer” variant.

3.1.3 Fisher Merging

Fisher merging, developed by Matena et al. (2021), is another procedure that improves upon weight averaging. Considering N models, Fisher merging optimizes

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \lambda_i \log p(\boldsymbol{\theta}|\boldsymbol{\theta}_i, \boldsymbol{F}_i), \tag{3.5}$$

which has the closed-form solution

$$\boldsymbol{\theta}^{*(j)} = \frac{\sum_{i=1}^{N} \lambda_i \boldsymbol{F}_i^{(j)} \boldsymbol{\theta}_i^{(j)}}{\sum_{i=1}^{N} \lambda_i \boldsymbol{F}_i^{(j)}}, \tag{3.6}$$

where $j = 1, \dots, |\boldsymbol{\theta}|$ and $p(\boldsymbol{\theta}|\boldsymbol{\theta}_i, \boldsymbol{F}_i)$ is the approximate isotropic Gaussian posterior distribution used for model i. $\boldsymbol{F}_i$ denotes the Fisher information matrix (FIM) (Soen et al., 2021) of $\boldsymbol{\theta}_i$. The λ_i correspond to the same weighting parameters used in equation 3.3. This procedure can be seen as computing a Fisher weighted average of the models' weights, i.e., a weighted average of the $\boldsymbol{\theta}_i$ where each individual $w \in \boldsymbol{\theta}_i$ is weighed according to it's Fisher information. This can naturally be seen as an improvement over weight averaging since an estimate of each weight's importance is incorporated. Storing the FIM in memory is usually impractical, except for the smallest models. This issue can be overcome by using the diagonal of the FIM (Kirkpatrick et al., 2016) instead of the actual FIM. The diagonal of the FIM can be estimated by

$$\hat{\boldsymbol{F}}_{\boldsymbol{\theta}} = \frac{1}{M} \sum_{i=1}^{M} \mathbb{E}_{y \sim p_{\boldsymbol{\theta}(y|\boldsymbol{x}_i)}} (\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y|\boldsymbol{x}_i))^2, \tag{3.7}$$

where $x_1, \ldots, x_M$ are drawn i.i.d. from the data that was used to train the model.

For a thorough derivation of the optimization problem in equation 3.5, we refer to the paper by Matena et al. (2021). To summarize it: They observed that averaging the weights of N models with the same architecture can be equated to finding the weights $\boldsymbol{\theta}$ that maximize the joint likelihood of the N models' posterior distributions. Usually, there is no access to a model's posterior distribution. They overcame this by approximating the posterior via an isotropic Gaussian approximation with a mean set to the model's weights. Using this approximation led to the optimization problem

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i}^{N} \log p(\boldsymbol{\theta}|\boldsymbol{\theta}_i, I), \tag{3.8}$$

where I is the identity matrix. The closed form solution is given by $\boldsymbol{\theta}^* = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{\theta}_i$, i.e. the average of the N models' weights. They explored using a Laplace approximation instead of an isotropic Gaussian approximation, which led to the optimization problem in equation 3.5 and ultimately to Fisher merging.

The creators of Fisher merging tested their procedure in comparison to weight averaging, copying the setups of Wortsman, Ilharco, Gadre, et al. (2022) and Wortsman, Ilharco, M. Li, et al. (2021). Fisher merging achieved better results than weight averaging in all cases.

3.1.4 Git Re-Basin

In their paper titled "Git Re-Basin: Merging Models Modulo Permutation Symmetries," which we refer to as "Git Re-Basin," Ainsworth et al. (2022) presented three matching procedures that align the weights of two independently trained models. They consider two models, A and B, which were trained from different random initializations with different data orders and potentially different hyperparameter configurations or datasets. The two models have the same architecture, and their weights are denoted as $\boldsymbol{\theta}_A$ and $\boldsymbol{\theta}_B$, respectively. The Git Re-Basin matching procedures aim to find a functionality-preserving permutation π which teleports $\boldsymbol{\theta}_B$ into the same loss basin as $\boldsymbol{\theta}_A$, such that when linearly interpolating between $\boldsymbol{\theta}_A$ and $\pi(\boldsymbol{\theta}_B)$, all intermediate models enjoy performance similar to $\boldsymbol{\theta}_A$ and $\boldsymbol{\theta}_B$. This process is visualized in figure 1, a replica we created based on a figure in the Git Re-Basin paper. Linearly interpolating between the weights of two models equals computing a weighted average of their weights where the weighting depends on an interpolation parameter α. This corresponds to the weighted weight averaging process defined in equation 3.3. The λ_1 and λ_2 $(N = 2)$ are defined by α in the following way: $\lambda_1 = \alpha$, $\lambda_2 = 1 - \alpha$. That means Git Re-Basin merges models A and B as follows: First, it permutes the weights of model B to align them with the weights of model A. Then, it computes a weighted average of model B's permuted weights and model A's regular weights.

The authors of Git Re-Basin investigated yet unexplained characteristics exhibited by solutions of stochastic gradient descent (SGD) algorithms in optimization problems of deep learning, such as two independently trained models with different random initializations inevitably achieving nearly identical performance with almost indistinguishable loss curves.

They suspected these unexplained characteristics to indicate the presence of yet undiscovered consistencies in the training dynamics of neural networks, resulting in independent training runs displaying similar characteristics. Their suspicions and a conjecture suggesting the possible existence of a π as depicted in figure 1 led them to develop the matching procedures. That conjecture is based on the two phenomena called mode connectivity and permutation symmetries of neural networks.

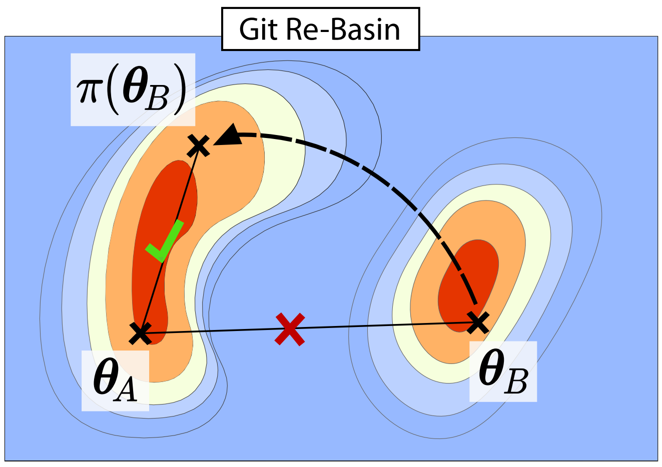


Figure 1: Ainsworth et al. (2022):"**Git Re-Basin merges models by teleporting solutions into a single basin.** $\boldsymbol{\theta}_B$ is permuted into functionally-equivalent $\pi(\boldsymbol{\theta}_B)$ so that it lies in the same basin as $\boldsymbol{\theta}_A$." (the figure is a replica of a figure in (Ainsworth et al., 2022))

Mode connectivity is the phenomenon that loss basins of independently trained models with different random initializations can be connected by simple curves in weight space, over which model performance is nearly constant (Garipov et al., 2018; Draxler et al., 2018). The term mode in this context refers to a region within the loss landscape, specifically a local minimum or a basin of attraction (loss basin). The curves connecting different modes in the weight space do not necessarily have to be perfectly straight lines; they can exhibit some complexity. Nonetheless, the essential concept is that there are paths in weight space along which the model's performance remains stable. Suppose the curves connecting the modes are perfectly straight lines. In that case, one can linearly interpolate between the models' weights with essentially no increase in loss at any point, Frankle et al. (2019) define that as linear mode connectivity (LMC). The Git Re-Basin procedures aim to establish LMC between models. Entezari et al. (2021) conjectured that differently initialized models trained on the same dataset using some variant of SGD could be permuted such that they become linearly mode connected.

Conjecture 1. *Permutation invariance, informal (Entezari et al., 2021): Most SGD solutions belong to a set S whose elements can be permuted in such a way that there is no barrier on the linear interpolation between any two permuted elements in S.*

A "barrier" in conjecture 1 is referring to a loss (or error) barrier, as defined in (Frankle et al., 2019). Consider two models M_1 and M_2 and the perfectly linear path, denoted as p, that connects these two models in the weight space. Informally, a loss barrier is a metric that quantifies the maximum increase in loss along the path p. If the loss barrier ≈ 0, it implies that linearly interpolating between the two models' weights does not lead to a performance decrease (=LMC).

Permutation symmetries of neural networks, noted by Hecht-Nielsen (1990), can be described as follows: Reordering the output weights of any layer in a neural network will preserve network functionality, as long as the input weights of the next layer are reordered accordingly. For a formal explanation, consider an L-Layer MLP as defined in equation 3.1 and a permutation matrix $\boldsymbol{P} \in S_d$ (S_d denotes the set of all $d \times d$ permutations). We can permute the output features of any intermediate layer l:

$$\boldsymbol{P}\boldsymbol{z}_{l+1} = \boldsymbol{P}\sigma(\boldsymbol{W}_l\boldsymbol{z}_l + \boldsymbol{b}_l) = \sigma(\boldsymbol{P}\boldsymbol{W}_l\boldsymbol{z}_l + \boldsymbol{P}\boldsymbol{b}_l). \tag{3.9}$$

From $\boldsymbol{z}_{l+1} = \boldsymbol{P}^T\boldsymbol{P}\boldsymbol{z}_{l+1}$ and equation 3.9 it follows that

$$\boldsymbol{z}_{l+1} = \boldsymbol{P}^T\sigma(\boldsymbol{P}\boldsymbol{W}_l\boldsymbol{z}_l + \boldsymbol{P}\boldsymbol{b}_l), \tag{3.10}$$

which means as long as we reorder the input weights of layer $l+1$ according to $\boldsymbol{P}^T$, we can permute the output features of layer l according to $\boldsymbol{P}$ while preserving functionality. For clarification, consider model weights $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ to be identical with the exception of

$$\boldsymbol{W}_l' = \boldsymbol{P}\boldsymbol{W}_l, \quad \boldsymbol{b}_l' = \boldsymbol{P}\boldsymbol{b}_l, \quad \boldsymbol{W}_{l+1}' = \boldsymbol{W}_{l+1}\boldsymbol{P}^T. \tag{3.11}$$

Then the two models $f(\boldsymbol{x};\boldsymbol{\theta})$ and $f(\boldsymbol{x};\boldsymbol{\theta}')$ are functionally equivalent for all inputs $\boldsymbol{x}$. Based on that, Ainsworth et al. (2022) drew the following implication: "This implies that for any trained weights $\boldsymbol{\theta}$, there is an entire equivalence class of functionally equivalent weight assignments, not just one such $\boldsymbol{\theta}$, and convergence to any one specific element of this equivalence class, as opposed to any others, is determined only by random seed."

The failure to find a π such that LMC holds, does not mean that none exists because the number of possible permutations is enormous. According to the authors of Git Re-Basin, conjecture 1 is realistically impossible to disprove for any wide model architecture. They demonstrated that LMC was not present early during the training phase, meaning that LMC is an emergent property of training algorithms and not model architectures. They showcased that solutions of SGD algorithms are implicitly biased to admitting LMC, while other training algorithms are not.

The authors matched neurons of neural networks trained on the same image classification datasets, whereby the models were trained from different random initializations with different random batch orders. In many cases, their procedures successfully established LMC, significantly improving accuracy when linearly interpolating between model weights after permuting compared to interpolation without permuting model weights. Accuracy was improved even in cases where LMC was not fully established. They also matched models trained on disjoint datasets. The disjoint datasets were created by splitting one original dataset. Both disjoint datasets still included samples of all classes previously included in the original dataset. They linearly interpolated between the weights of the permuted source model and the weights of the other source model. They found intermediate models to outperform both on the original dataset in terms of test loss (but not in terms of accuracy). However, the intermediate models were outperformed by a model trained on the original dataset. Other merging procedures that use other neural alignment algorithms and then average weights can be found in (Singh et al., 2019; H. Wang et al., 2020).

We explain the Git Re-Basin matching procedures using two MLPs, A and B, without bias terms. However, they work for almost arbitrary model architectures, including bias terms (as long as the models have the same architecture).

Activation Matching

Inspired by Y. Li et al. (2016) previous work, which showed that models must learn similar features to perform the same task successfully, this procedure matches the learned features of models by performing regression between their activations. More precisely, it constrains ordinary least squares between the activations of model A and the activations of a permuted model B to solutions in the set of permutation matrices S_d. The d-dimensional activations for the l'th layer of models A and B for all n training data points are denoted as $\boldsymbol{Z}^{(A)}, \boldsymbol{Z}^{(B)} \in \mathbb{R}^{d\times n}$ respectively. The procedure solves the optimization problem

$$\boldsymbol{P}_l = \underset{\boldsymbol{P}\in S_d}{\arg\min} \sum_{i=1}^{n} \|\boldsymbol{Z}_{:,i}^{(A)} - \boldsymbol{P}\boldsymbol{Z}_{:,i}^{(B)}\|^2 = \underset{\boldsymbol{P}\in S_d}{\arg\max}\, \langle \boldsymbol{P}, \boldsymbol{Z}^{(A)}(\boldsymbol{Z}^{(B)})^T\rangle_F, \tag{3.12}$$

where $\langle \boldsymbol{K}, \boldsymbol{H}\rangle_F = \sum_{i,j} K_{i,j}H_{i,j}$ denotes the Frobenius inner product between real-valued matrices $\boldsymbol{K}$ and $\boldsymbol{H}$. Note that a training dataset is required to compute the activations. Equation 3.12 constitutes a linear assignment problem (LAP), which can be solved efficiently by existing algorithms (Hahn et al., 1998; Jonker et al., 1987; Crouse, 2016). After the LAP is solved, the computed per-layer permutations $\boldsymbol{P}_l$ are applied to the weights of model B

$$\boldsymbol{W}'_l = \boldsymbol{P}_l \boldsymbol{W}_l^{(B)} \boldsymbol{P}_{l-1}^T, \quad \boldsymbol{b}'_l = \boldsymbol{P}_l \boldsymbol{b}_l^{(B)}, \tag{3.13}$$

producing weights $\boldsymbol{\theta}'_B$ with activations that align as closely possible with those of $\boldsymbol{\theta}_A$. Computing $\boldsymbol{Z}^{(A)}$ and $\boldsymbol{Z}^{(B)}$ requires a single pass over the training dataset, resulting in a computationally relatively efficient procedure.

Weight Matching

Each row of the l layer weights W_l corresponds to a single feature, and if two such rows were equal, they would compute the same feature. Following that idea, this procedure aims to match the learned features of the models by permuting the weights of model B such that $\left[\boldsymbol{W}_l^{(A)}\right]_{i,:} \approx \left[\boldsymbol{W}_l^{(B)}\right]_{j,:}$ for every layer. To find such a permutation π of $\boldsymbol{\theta}_B$, ordinary least squares regression is performed between vectorized versions of $\boldsymbol{\theta}_A$ and $\pi(\boldsymbol{\theta}_B)$:

$$\underset{\pi}{\arg\min} \, \|\text{vec}(\boldsymbol{\theta}_A) - \text{vec}(\pi(\boldsymbol{\theta}_B))\|^2 = \underset{\pi}{\arg\max} \, \text{vec}(\boldsymbol{\theta}_A) \cdot \text{vec}(\pi(\boldsymbol{\theta}_B)). \tag{3.14}$$

Re-expressing this in terms of full weights results in

$$\underset{\pi=\{\boldsymbol{P}_i\}}{\arg\max} \, \langle \boldsymbol{W}_1^{(A)}, \boldsymbol{P}_1 \boldsymbol{W}_1^{(B)} \rangle_F + \langle \boldsymbol{W}_2^{(A)}, \boldsymbol{P}_2 \boldsymbol{W}_2^{(B)} \boldsymbol{P}_1^T \rangle_F + \cdots + \langle \boldsymbol{W}_L^{(A)}, \boldsymbol{W}_L^{(B)} \boldsymbol{P}_{L-1}^T \rangle_F. \tag{3.15}$$

This is another matching problem, but unlike the case in activation matching, this is not a LAP, and the authors of Git Re-Basin assert it to be NP-hard. To solve the matching problem, they proposed an approximation of the optimization problem, which transforms it into a LAP by focusing on a single $\boldsymbol{P}_l$ while holding the others fixed:

$$\begin{aligned} &\underset{\boldsymbol{P}_l}{\arg\max} \, \langle \boldsymbol{W}_l^{(A)}, \boldsymbol{P}_l \boldsymbol{W}_l^{(B)} \boldsymbol{P}_{l-1}^T \rangle_F + \langle \boldsymbol{W}_{l+1}^{(A)}, \boldsymbol{P}_{l+1} \boldsymbol{W}_{l+1}^{(B)} \boldsymbol{P}_l^T \rangle_F \\ = &\underset{\boldsymbol{P}_l}{\arg\max} \, \langle \boldsymbol{P}_l, \boldsymbol{W}_l^{(A)} \boldsymbol{P}_{l-1} (\boldsymbol{W}_l^{(B)})^T + (\boldsymbol{W}_{l+1}^{(A)})^T \boldsymbol{P}_{l+1} \boldsymbol{W}_{l+1}^{(B)} \rangle_F \end{aligned} \tag{3.16}$$

To solve this LAP, $\boldsymbol{P}_l$ is selected to maximize the optimization problem in equation 3.16 layer by layer. This is repeated until convergence, resulting in the best possible $\boldsymbol{P}_l$ for every layer. The authors of Git Re-Basin summarize this in algorithm 2. Once the algorithm terminates, the permutations $\boldsymbol{P}_l$ are applied to the weights

Algorithm 2 PermutationCoordinateDescent (Ainsworth et al., 2022)

```
Input: Model weights $\theta_A = \{W_1^{(A)}, \ldots, W_l^{(A)}\}$ and $\theta_B = \{W_1^{(B)}, \ldots, W_l^{(B)}\}$ /* also works with biases, but for ease of presentation biases are not included */
Result: A permutation $\pi = \{P_1, \ldots, P_{L-1}\}$ of $\theta_B$ such that $\text{vec}(\theta_A) \cdot \text{vec}(\pi(\theta_B))$ is approximately maximized.
Initialize: $P_1 \leftarrow I, \ldots, P_{L-1} \leftarrow I$
repeat
    for $l \in \text{RandomPermutation}(1, \ldots, L-1)$ do
        $P_l \leftarrow \text{SolveLAP}\left(W_l^{(A)} P_{l-1} (W_l^{(B)})^T + (W_{l+1}^{(A)})^T P_{l+1} W_{l+1}^{(B)}\right)$
    end for
until convergence for all $l$
```

of B as in equation 3.13. Furthermore, algorithm 2 always terminates, is fast, and does not require a labeled training dataset, in contrast to the activation matching procedure. The authors of Git Re-Basin also presented an extended version of the algorithm capable of handling any number of models.

Learning Permutations with a Straight-through Estimator

The third Git Re-Basin procedure learns the ideal permutation of weights $\pi(\boldsymbol{\theta}_B)$ implicitly by optimizing an approximation of $\pi(\boldsymbol{\theta}_B)$, denoted as $\tilde{\boldsymbol{\theta}}_B$. Let $\mathcal{L}$ be some loss function; the goal is to optimize

$$\min_{\tilde{\boldsymbol{\theta}}_B} \mathcal{L}\left(\frac{1}{2}\left(\boldsymbol{\theta}_A + \text{proj}\left(\tilde{\boldsymbol{\theta}}_B\right)\right)\right), \quad \text{proj}(\boldsymbol{\theta}) \triangleq \arg\max_{\pi} \text{ vec}(\boldsymbol{\theta}) \cdot \text{vec}(\pi(\boldsymbol{\theta}_B)). \tag{3.17}$$

Solving this optimization problem requires a labeled dataset for computing the loss. Now, we run into the issue that the optimization problem involves non-differentiable projection operations, which is problematic in the backward pass when computing the gradient of the loss, as gradient-based optimization algorithms typically require smooth differentiability. However, the problem can be overcome by a straight-through estimator (Bengio et al., 2013; Kusupati et al., 2021; Rastegari et al., 2016): In the forward pass, the projection in the optimization problem 3.17 is computed as usual, projecting $\tilde{\boldsymbol{\theta}}_B$ to the nearest realizable $\pi(\boldsymbol{\theta}_B)$. In the backward pass, instead of using the result of the projection operation proj$(\tilde{\boldsymbol{\theta}}_B)$, the gradient is computed using $\tilde{\boldsymbol{\theta}}_B$. This enables the computation of usable gradients while evaluating the loss without breaking the projection constraint. The projection proj$(\tilde{\boldsymbol{\theta}}_B)$ can be solved by algorithm 2. The creators found that initializing $\tilde{\boldsymbol{\theta}}_B = \boldsymbol{\theta}_A$ leads to best results in practice. Summarizing the procedure: The currently nearest realizable $\pi(\boldsymbol{\theta}_B)$ is approximated by solving proj$(\tilde{\boldsymbol{\theta}}_B)$. Then, the loss of the midpoint between $\boldsymbol{\theta}_A$ and $\pi(\boldsymbol{\theta}_B)$ is computed. Afterward the gradient of the computed loss is evaluated using $\tilde{\boldsymbol{\theta}}_B$ instead of $\pi(\boldsymbol{\theta}_B)$. Finally, the weights are updated through gradient descent. This process is repeated until convergence.

3.1.5 Other Procedures

The two procedures we present here will broaden the reader's perspective on how models may be merged.

Concatenate and Prune

Pasen et al. (2022) proposed a procedure for merging two models with the same architecture. Their procedure concatenates the two models layerwise and then learns the importance of the units in the concatenated model. After computing the importance, half of the units are pruned according to their importance, such that the resulting model has the same size and architecture as the original models. The pruned units are those with the least importance. They refer to the two original models as teachers and the concatenated model before pruning as big students. The procedure is displayed in algorithm 3. Steps 2.-4. correspond to the merging procedure. The importance of the big student's neurons can be learned in various ways, e.g. as in (Molchanov, Mallya, et al., 2019). Unlike previous procedures, concatenating and pruning does not involve weight averaging or any form of altering the model weights.

The motivation behind the procedure's creation was to find better features for network training and to improve the accuracy/size tradeoff in DL (which refers to the fact that models with larger sizes tend to achieve better accuracy, but bigger networks are more costly in terms of memory and computation). Other methods like channel pruning (Voita et al., 2019; Molchanov, Tyree, et al., 2016) and knowledge distillation (Hinton

Algorithm 3 ConcatenateAndPrune (Pasen et al., 2022)

```
Input: Models $A$ and $B$ with weights $\boldsymbol{\theta}_A$ and $\boldsymbol{\theta}_B$
1. Training of teachers $A$ and $B$
2. Concatenation of teachers into a big student
3. Learning of importance of big student's neurons
4. Compression of big student by pruning half of the neurons according to importance
5. Fine-tuning of the student
return Fine-tuned student
```

et al., 2015) aim to achieve the same goals, but they require the costly training of a bigger network first. The concatenate and prune procedure avoids the potentially too costly training of a bigger network. In their experiments, Pasen et al. (2022) merged models trained on the same image classification datasets but starting from different random initializations. They showed that the merged model achieved better results than the two original models (it successfully learned better features).

Ensemble

Unlike the previous procedures, a N model ensemble is not merging the N models' weights. An ensemble merges the predictions of the N models in some way (Ganaie et al., 2021; Dietterich, 2000). The simplest approach to merging predictions is again, to average them:

$$\frac{1}{N}\sum_{i=N}^{N} f(\boldsymbol{x}, \boldsymbol{\theta}_i). \tag{3.18}$$

This procedure is usually applied to models trained on the same task to improve performance, especially generalization. It does not meet our criteria of merging multiple models into one model. However, it should be mentioned because many related works in model merging use model ensembles as a baseline for their methods. It also broadens the perspective on different ways to merge models.

3.2 Bridging the Gap to Continual Reinforcement Learning Scenarios

The model merging procedures from the previous section were applied exclusively in deep SL, specifically to models trained on image and text classification datasets. Considering the procedures' theoretical background and prerequisites, we suppose that they can achieve comparable results in other DL scenarios, including deep RL. The prerequisites focus on model architecture, model initialization, and optimization algorithms used during training. These properties are not specific to deep SL but DL in general. The only SL-specific requirement of some procedures is a labeled validation set used to evaluate model accuracy or test loss. However, a comparable labeled dataset could be crafted for RL scenarios, or other RL-specific performance measurements could be used instead. The unexplained characteristics exhibited by solutions of SGD algorithms in DL mentioned in the Git Re-Basin paper appear in all kinds of DL scenarios that incorporate SGD algorithms, not just deep SL scenarios (Stooke et al., 2018; Ohnishi et al., 2019). That means, if yet undiscovered consistencies in the training dynamics of neural networks triggered by SGD algorithms do exist, as suggested by the authors of Git Re-Basin, we think those consistencies should also exist in deep RL, provided that SGD algorithms are used during training (which is the case in most deep RL methods). If those consistencies enable merging procedures to work in deep SL, we suspect those consistencies also enable the procedures to work in deep RL. It is clear that specifics to deep RL methods and RL tasks alter the training dynamics compared to deep SL, but it is unclear to what extent they do so and whether that will prevent fruitful application of the merging procedures to deep RL models.

Why do we wonder whether model merging can lead to well-performing solutions in deep RL? The obvious answer is that if it does, it can be utilized to improve model performance similarly to how it is utilized in deep SL. However, we propose another aspect that makes model merging in deep RL uniquely intriguing.
In 3.1.4, we mention that in the Git Re-Basin paper, a classification dataset was split into two disjoint datasets. Both new datasets still contained samples of all classes previously included in the original dataset. Then, two differently initialized models were trained on the disjoint datasets, respectively (one model per dataset). Afterward, the models were merged, and the resulting merged model achieved lower test loss than the two unmerged models on the original dataset. We hypothesize that developing a similar methodology using episodic RL environments instead of classification datasets is possible, allowing Git Re-Basin procedures (and maybe other merging procedures) to be used in CRL scenarios.

Based on the findings by Y. Li et al. (2016), which showed that models performing the same task must learn similar features (this is also mentioned in 3.1.4), and that models trained from different initializations learn similar features in different places, we claim that the merging procedures in 3.1 are highly dependent on the learned features of the models they are trying to merge. Averaging weights (which all of the procedures do in some way) that belong to entirely different features can never lead to viable results if the merged model is intended to inherit the capabilities of the source models. The prerequisites of the procedures ensure that the source models learn enough similar features at the same places for the procedures to achieve decent results, e.g. by requiring them to be trained from the same initialization. The Git Re-Basin procedures are the only ones that provide a viable approach to merging models trained from different initializations and with different data batch orders because they align the models' features by permuting the weights of one model accordingly before averaging their weights. Their disjoint dataset methodology works because they keep samples of all classes in the two disjoint datasets after splitting. Models trained on those disjoint datasets learn similar features because they have to solve the same task of recognizing the same classes.

Episodic RL environments that are at least somewhat complex typically require the agent to learn multiple skills to maximize the return. New environments could be created so that each of them corresponds to a subset of the skills required to solve the task in the original environment. That subset should differ for each new environment while as much similarity as possible is kept. Splitting an environment in that way requires at least the change of the reward structure and, most likely, the change of possible environment states (the state space), which is why we consider the new environments in this setting to entail distinct tasks. In contrast, splitting a labeled dataset into two disjoint datasets that include the same classes can hardly be considered as creating two new tasks. We refer to the task in the original environment as a super-task and to the tasks in the new environments created by splitting the super-task as sub-tasks of the super-task. Suppose many similarities remain shared between the sub-tasks (e.g. state properties and action space), similar to how all classes remained in both disjoint datasets of the Git Re-Basin methodology. In that case, we think models trained on the sub-tasks should learn similar features, and all these similar features should be useful for performing the super-task. Especially the state properties should remain the same. If states are images, which is the case in most game-like RL environments, such properties would be visual properties.

From the previous paragraphs, we can extract our main hypothesis, which builds upon other hypotheses, for example that model merging will enjoy similar success in deep RL, given the same prerequisites:

Hypothesis. *Using a Git Re-Basin procedure to merge distinct models that solve different episodic RL sub-tasks of the same super-task will result in a model that performs well on the super-task.*

We posit that creating an agent that performs well on a super-task by merging models trained on sub-tasks of the super-task achieves some popular goals of CRL mentioned in 2.1.4:

- Learned skills are building blocks for more complex skills (hierarchical learning).
- Accelerated learning of more complex tasks through prior learning of simpler tasks (even though it is unclear whether learning sub-tasks will be faster than learning the super-task).

Even though we explain that two sub-tasks created from one super-task are distinct tasks, these tasks are not unrelated, and we do not think that current model merging procedures allow for merging such to achieve a well-performing result (learned features are entirely different). We think of merging models trained on sub-tasks to solve a super-task as continually learning one super-task in a modular fashion. This is similar to the modular CRL method we mention in 2.1.4 (Mendez et al., 2022) but quite different from most other current CRL methods, which focus on catastrophic forgetting. Assuming our hypothesis is correct, we suggest that existing model merging procedures could be combined with current CRL methods to enhance CRL agents with more capabilities. For example, model merging could be used to continually learn different super-tasks by merging models trained on their sub-tasks (only merging models trained on sub-tasks belonging to the same super-task). This would result in multiple merged models that can solve different super-tasks (each merged model corresponds to one super-task). Afterward, an existing CRL method from the ones mentioned in 2.1.4 could be used to learn the super-tasks sequentially, since merging can probably not be used to merge super-tasks, as they are most likely entirely unrelated tasks.

3.3 Potential of Existing Merging Procedures

Our hypothesis states that the Git Re-Basin procedures can produce viable solutions in the described super-task/sub-tasks scenario, but what about the other procedures? By the super-task/sub-tasks scenario, we mean the merging of models, which were each trained on different sub-tasks of the same super-task, with the goal of creating a merged model that performs well on the super-task.

Greedy soup generally only applies to merging scenarios where the source models are trained on the same task. It is senseless to predict its potential for the super-task/sub-tasks scenario.

When it comes to merging models trained from different initializations in the super-task/sub-tasks scenario, we do not think any non-Git Re-Basin procedure can produce well-performing solutions because even if the models learn many similar features (which is not ensured, but we hypothesize that they do), they do not learn them at the same places. However, there could be minor differences between the solutions of the non-Git Re-Basin procedures. Weight averaging should always perform terribly and worse than any other procedure because the other procedures are better versions of weight averaging. Learned soup and Fisher merging could slightly improve upon weight averaging in the lucky case that some similar features are learned at the same places by pure chance. Moreover, learned soup could weigh one model very heavily in the averaging process, which could be a trick to avoid the terrible outcome of unweighted averaging. If that is the case, it can greatly outperform Fisher merging, but that would be due to the trick and not through constructive merging. The trick only applies if one sub-task is sufficient to achieve decent returns in the super-task.

When it comes to merging models trained from the same initialization in the super-task/sub-tasks scenario, there is some potential for the non-Git Re-Basin procedures to produce well-performing solutions. It depends very much on the similarity of features required to solve the sub-tasks. If the sub-tasks require the agents to learn very similar features as we hypothesize they do, they should mostly do so in similar places due to the same weight initialization. However, in deep RL, we expect the generated episodes during training to influence the learned features. Suppose the different models do not encounter similar states at the beginning of training. In that case, similar features may be learned in different places, even though the models are trained from the same initialization. The reward structures of the different sub-tasks may also influence the learned features. Due to that, we expect the Git Re-Basin procedures to outperform the others because of feature alignment. We expect the solutions of the other procedures to show some learning of the super-task nonetheless, at least more so than in the different initialization cases. We expect weight averaging to be naturally outperformed by learned soup and Fisher merging. We expect Fisher merging to outperform learned soup as it learns the importance of each weight. Learned soup can only use the same weight for every model weight of one model, which allows Fisher merging to have more control over the individual weights. In an unexpected scenario where the source models trained on the sub-tasks learn similar features at mostly the same places, the solutions of Fisher merging and learned soup might outperform those of the Git Re-Basin procedures because no feature alignment is required. They use better weighting parameters in the averaging process (assuming the weighting parameters of the Git Re-Basin procedures are set beforehand, and not every possible weight is interpolated).

It is important to consider that learned soup and Fisher merging require a labeled dataset to merge models. Learned soup uses it to estimate a loss and learn optimal parameters, and Fisher merging uses it to estimate the FIM. While it is possible to create a labeled RL dataset manually or save the training data of a deep RL training run, it is very memory-intensive and impractical, making the procedures unattractive for merging in deep RL.

Between the three Git Re-Basin procedures, there should only be minor differences, if any. That was also the case in the Git Re-Basin paper, where the procedures achieved very similar results. Activation matching and learning permutations with a straight-through estimator have the drawback that they are not easily applied in deep RL, as they both use a labeled dataset. Weight matching provides a more practical approach. There is no technical difference between applying it in deep SL and deep RL scenarios.

Table 3.1 summarizes our thoughts on the potential of each procedure in the super-task/sub-tasks scenario and includes short summaries of the individual procedures.

Table 3.1: Our thoughts on what performance can be expected from solutions of each merging procedure in the super-task/sub-tasks scenario. The thoughts are separated based on whether the source models are trained from the same weight initialization or different weight initializations.

Procedure	**Summary**	**Same Initializations**	**Different Initializations**
Weight Averaging	Averages the weights of the source models. (we only consider weight averaging where both source models are weighed equally)	It could perform well unless the source models do not learn similar features at the same places or learn very few similar features in general. Then it will perform terribly.	It should perform terribly because the source models learn similar features (if any) at different places. Weight averaging cannot overcome that.
Greedy Soup	Sequentially adds source models to weight averaging and evaluates the result afterward. Undoes merging and discards the source model if performance decreases.	This procedure only makes sense when merging source models that are trained on the same task to increase performance. Thus, it is not suited for the super-task/sub-tasks scenario.	This procedure only makes sense when merging source models that are trained on the same task to increase performance. Thus, it is not suited for the super-task/sub-tasks scenario.
Learned Soup	Learns an ideal weighted average of the source models' weights by minimizing a loss w.r.t. the weighting parameter.	It should always perform better than weight averaging. If the source models do not learn similar features at the same places, it could weight one model significantly in the averaging process to avoid terrible performance (we call this the learned soup trick).	It should slightly outperform weight averaging but still perform terribly for the same reasons why weight averaging should perform terribly. It could overcome the terrible performance by weighting one model significantly in the averaging process (trick).
Fisher Merging	Computes a weighted average of the source models' weights, where weighting is done according to the Fisher information of each model weight.	It should perform better than weight averaging and learned soup, except if there are no similar features at the same places. Then, it will perform terribly and worse than learned soup (because of the learned soup trick).	It should perform terribly (for the same reasons as weight averaging) but slightly better than weight averaging and worse than learned soup due to the learned soup trick.

Git Re-Basin: Weight Matching	Obtains an optimal permutation by minimizing the distance in weight space between one model and a permuted version of the other model. Then, it computes a weighted average of the permuted model's weights and the other model's weights.	It should outperform any previous procedure because we expect not all similar features to be learned at the same places. In the unexpected case that almost all similar features are learned at the same places, using better weighting parameters, which Fisher merging and learned soup do, would be better than this procedure, as there is no need to align features.	It should perform worse than with same initializations but significantly better than weight averaging, Fisher merging and learned soup. In the unexpected case that entirely different features are learned by the source models, it should perform just as poorly as the previous procedures. This is because there would be no features to align, but then no procedure could produce good results.
Git Re-Basin: Activation Matching	Learns an optimal permutation by performing OLS regression between the models' activations. Then, it computes a weighted average of the permuted model's weights and the other model's weights.	This procedure is challenging to apply in RL, because it requires a labeled training dataset. It should perform similarly to Weight Matching.	This procedure is challenging to apply in RL, because it requires a labeled training dataset. It should perform similarly to Weight Matching.
Git Re-Basin: Learning Permutations with a Straight-through Estimator	Minimizes the loss between one model's weights and an approximation of the other model's permuted weights, thus learning an optimal permutation. Then, it computes a weighted average of the permuted model's weights and the other model's weights.	This procedure is challenging to apply in RL, because it requires a labeled training dataset. It should perform similarly to Weight Matching.	This procedure is challenging to apply in RL, because it requires a labeled training dataset. It should perform similarly to Weight Matching.

3.4 Proposed Methodology

To investigate the correctness of our hypothesis and the potential of existing merging procedures, we propose a methodology that is based on the idea of decomposing an episodic RL super-task into two sub-tasks and merging models trained on the sub-tasks.

We select an episodic game-like RL environment in which we can identify a set of three skills that the agent needs to develop to (directly or indirectly) maximize the returns. No other skills must be required for return maximization, only those three. For that, the environment is required to display some degree of complexity. It may not be possible to identify individual skills in too simple environments. We denote the set of the three skills as superset $= \{S1, S2, S3\}$. Now we split this set into two subsets: subset1 $= \{S1, S2\}$ and subset2 $= \{S1, S3\}$. Based on the subsets, we create two new environments from the original environment, such that one new environment requires subset1 for return maximization and the other requires subset2. As in the sections, we refer to the task in the original environment as super-task and to the tasks in the new environments as sub-tasks of the super-task. Note that we narrow our focus to one individual skill per sub-task. We try not to create an overly complex scenario. We intend to keep as many similarities between the sub-tasks as possible to maximize the similarity in the learned features of two distinct agents, each trained on one sub-task, respectively.

Once the super-task and the corresponding sub-tasks are created, we train five policy models on each sub-task. The policy models return action probabilities based on the environment states they receive as inputs. Thus, the policy model is synonymous with the agent, and we use these terms interchangeably from now on. We use a seed for every model's weight initialization and a seed for level generation during the training run. We purposefully use environments that provide this option. This improves the reproducibility of our experiments. In this case, a level refers to one episode inside the environment. Using the same seed for level generation between two different sub-tasks will not result in more similar levels, as they belong to distinct environments. Whereas using the same seed for level generation in training runs on the same task would generate the same levels (and the order of them) in both training runs. We use different combinations of seeds for each model, whereby that combination is an element out of $\{0, 1, 2, 3, 4\} \times \{0, 1, 2, 3, 4\}$. The five models trained on the first sub-task have the seed combinations $\{00, 11, 22, 33, 44\}$, and the five models trained on the second sub-task have the seed combinations $\{01, 12, 23, 34, 40\}$. The first number corresponds to the weight initialization seed, and the second corresponds to the level generation seed used during training. Now, we perform two rounds of merging between the sub-task models. In the first round, we merge models from the same weight initialization, and in the second round, we merge models trained from different weight initializations. This gives us the following merge combinations (the $+$ signals merging and not addition):

Round 1: $00 + 01, \quad 11 + 12, \quad 22 + 23, \quad 33 + 34, \quad 44 + 40$

Round 2: $00 + 12, \quad 11 + 23, \quad 22 + 34, \quad 33 + 40, \quad 44 + 01$

This results in ten merged models where the source models were trained on distinct sub-tasks belonging to the same super-task. The goal is for these models to perform reasonably well on the super-task. Defining a performance threshold for what can be considered reasonably well in this scenario is difficult, e.g. a minimum average return. Instead, we evaluate performance by comparing the merged models' performance on the super-task to the performance of models trained directly on the super-task. The closer the merged models' performance is to the models' performance trained directly on the super-task, the better.

We let every merged model play the super-task for 5000 episodes, using the level generation seed $\{5\}$. Similarly, we train five models on the super-task, using seed combinations $\{00, 11, 22, 33, 44\}$, and then let each model play the super-task for 5000 episodes using the same level generation seed $\{5\}$. For each merging round, we compute the mean average return across the five merged models. We also compute the mean average return across the five models trained directly on the super-task, which we call the upper bound (UB). We call it upper bound because we do not expect the merged models to outperform the models trained directly on the super-task. Then, we compare these three mean average return values. We do the same for the standard deviation of the mean return (std). This will let us compare the performances of merging round one, round two, and UB. We also investigate individual merge performances to get an insight into the consistency of the merging. We use the same architecture for every model and the same learning algorithm for every training run. After every training run, we confirm that the model successfully learned the intended task and especially the skills required for the task. When multiple skills are to be learned for return maximization, it is not ensured that the model learns them, even if it achieves decent returns. Many nuances can lead to an agent finding tricks to achieve good returns while not solving the intended task or learning all the intended skills. Generally, if there are multiple skills to be learned, the agent could learn a subset of those and still achieve good returns. We ensure this is not the case for any model by the judgment of a human oracle (watching the model play the game after training) and by designing the super-task and sub-tasks accordingly.

We intend to repeat the described merging process more than once, using different procedures. Based on our thoughts on the potential of each merging procedure (3.3), we select two of them to try in our experiments. Those are weight averaging and "Git Re-Basin: Weight Matching". Our hypothesis focuses on the Git Re-Basin procedures, and they are the only procedures that align features between models, so we deem it necessary to try at least one of them. In the previous section, we mentioned that we expect all Git Re-Basin procedures to produce similarly performing solutions but that two require a labeled dataset. We do not see the need to create a labeled dataset if we expect the same results. Thus, we use weight matching and none of the other two.

We try weight averaging over Fisher merging or learned soup for several reasons. What we are mainly interested in is uncovering the correctness of our hypothesis and suspicions. For that, we do not need to get the best possible merging outcome. The most important thing is that we can draw insightful conclusions concerning our hypothesis from our experiments. Based on our thoughts in the previous section, each of the three procedures will lead us to the same conclusions in the same situation. That is because they all average weights without aligning features; it is just that Fisher merging and learned soup do so in a better way. However, Fisher merging and learned soup require a labeled dataset, and it is not worth investing the resources in creating such if trying the procedures will not help us gain more insight compared to only trying weight averaging. The performance differences between solutions of the three procedures can be explored in future work, but we are interested in something else. Moreover, trying weight averaging will give us insights into what we can expect from solutions of Fisher merging and learned soup. Figure 2 shows a visual presentation of our proposed methodology.

In addition to the one super-task we consider, we select a second episodic RL environment and repeat the process explained in the last paragraphs. Examining our hypothesis in two environments allows us to draw more meaningful conclusions. We select two super-tasks with varying complexity. The more complex super-task requires us to modify the state space, the action space, and the reward structure to create sub-tasks. The other, less complex super-task requires us only to modify the state space and the reward structure. This way, we can better assess how much impact the individual components of an RL task (state space, action space, and reward structure) have on the agent's learned features.

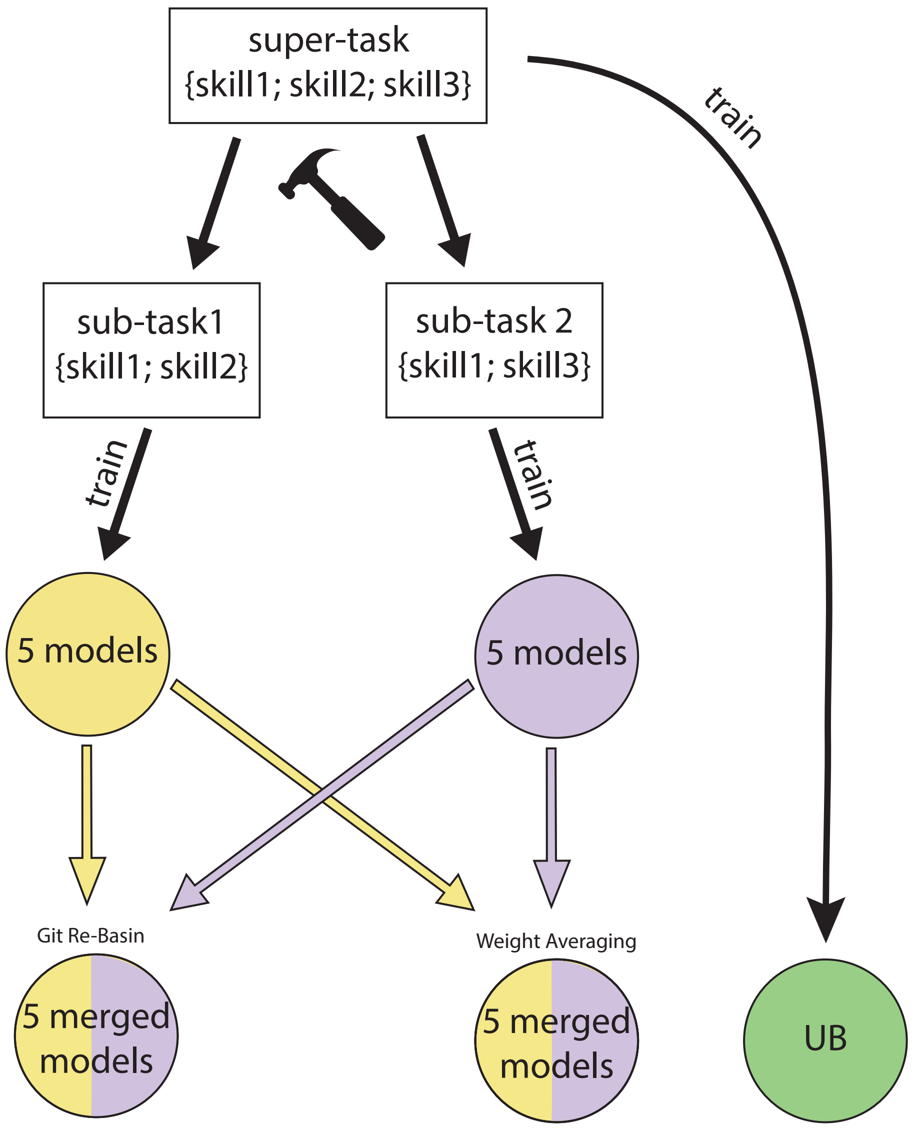


Figure 2: Visual presentation of our proposed methodology.

4 Experiments

Having established our methodology, we can now conduct experiments. We first present our setup, and then discuss the results.

4.1 Setup

4.1.1 Environments

We begin with our environment choice. Cobbe et al. (2019) created a suite of 16 episodic game-like environments for RL research called Procgen Benchmark. The suite offers a diverse range of procedurally generated games and allows for simple modification. This simple modification and tweaking of the environments is what we require to create super-tasks and sub-tasks that meet our criteria (see 3.4). Similar suites exist, such as ALE (Bellemare et al., 2012) and OpenAI Gym (Brockman et al., 2016). However, Procgen Benchmark's code is more easily modified. Moreover, the suite provides the option to specify a seed for procedural level generation, allowing us to adjust training and testing runs as we desire. Procedurally generated games are games in which various aspects of the game world, such as the game's levels, maps, characters, items, or even gameplay elements, are generated algorithmically rather than being manually designed by game developers.

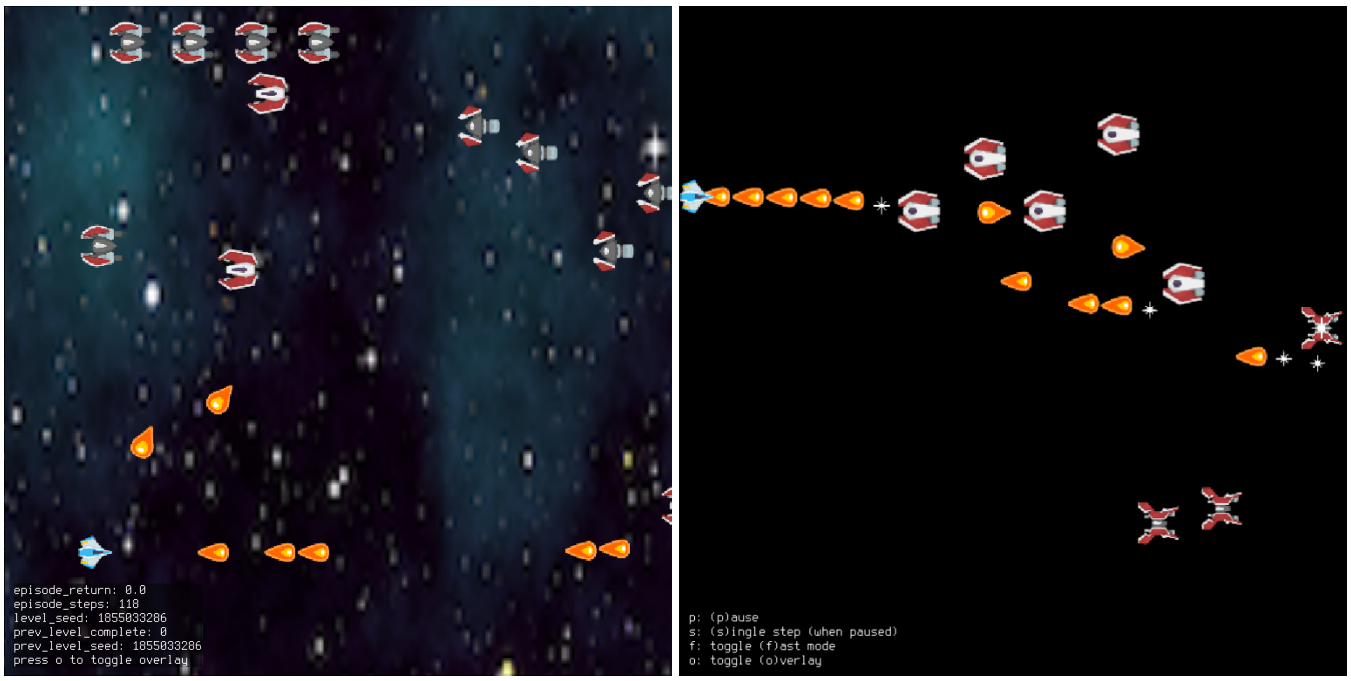
Not every Procgen environment is suited to our intentions. Some are overly simple, so we cannot identify three skills that must be learned to maximize the return. Others are too complex, and learning them requires a larger model architecture than the one we are using. However, there are enough options that meet our criteria.

Before creating the sub-tasks, we tweak the super-tasks first. We are interested in the merging of models in deep RL and want to uncover the correctness of our hypothesis from 3.2. To do so, we remove unnecessary influences as best as we can. By unnecessary influences, we mean anything that is unrelated to the skills we want the models to learn but complicates the environment. We are not interested in creating the new best RL agent. We remove backgrounds in all environments, which leaves us with completely black backgrounds for all games. Usually, Procgen games include multiple themes for each entity, which means they can look differently each time they spawn. We limit the number of possible themes. By entity, we refer to any structure existing in the environment besides the player. Different difficulty settings can be selected for every environment in Procgen Benchmark. Those are easy, hard, and intense. We consider only easy difficulty settings.

Every environment in Procgen Benchmark has an action space of size 15, meaning the player can give 15 different action inputs to the environment. Usually, a subset of these actions are no-operation actions, meaning they may be given as input, but they do not have any consequence on the environment state. That is to keep consistency between environments. We do not break that rule. Whenever we alter actions, we alter their consequences on the environment state. If we want to remove an action, we make it a no-operation action. This is also necessary to keep model architectures the same for merging.

For our more complex super-task, where more aspects of the environment need to be changed to create sub-tasks, we select the game called Starpilot. For our simpler super-task, we select the game called Fruitbot.

Starpilot



```
episode_return: 0.0
episode_steps: 118
level_seed: 1855033286
prev_level_complete: 0
prev_level_seed: 1855033286
press o to toggle overlay
```

```
p: (p)ause
s: (s)ingle step (when paused)
f: toggle (f)ast mode
o: toggle (o)verlay
```

Figure 3: **Left:** The original Starpilot game before we modify it (easy settings). **Right:** Our super-task. The Starpilot game after we remove backgrounds and limit enemy starship themes. Player starship is blue, enemy starships are red.

Starpilot is a simple side-scrolling shooter game. An image of a possible game state can be seen in figure 3. The player controls a starship and must survive until he reaches the level end, which is always 200 in-game units away in every level. Due to the nature of scrolling games (the background moves the level towards the end, no matter what the player does), the level end is reached inevitably in a relatively short time frame (≈30s) unless the player's starship is destroyed on his journey. The level end is signaled by a giant rocket, which can be seen in figure 4. Upon touching the rocket with his starship, the player successfully completes the level, and a new level begins. During his voyage, the player encounters enemy starships that fire projectiles directly targeted at him. If an enemy projectile hits the player's starship, the level ends immediately, and a new level begins. If the player's starship touches an enemy starship, the level ends immediately, and a new level begins. There are different types of enemy starships. Some are fast flyers that move quickly, and others are regular flyers that do not move as quickly. Both types of flyers can spawn having the same theme. They can also spawn with different themes each time they spawn. This makes it impossible to identify different enemy types based on their appearance. Enemy flyers spawn in groups. The maximum group size in which enemies can spawn is size 5, meaning whenever a group of enemy flyers spawns in the environment, it consists of 1 to 5 flyers. The player can move his starship in every direction, including diagonally. In addition to that, he can fire projectiles similar to those of the enemies. However, the player can only fire projectiles in a straight line, whereas enemies can fire projectiles upward, backward, downward, and diagonally (in every possible direction). The player can destroy enemy starships by hitting them with his projectiles. However, he needs to hit enemies multiple times to destroy them. Enemies only need to hit the player once to make the level end. The player's starship, the enemy starships, their projectiles, and the rocket are all entities that exist in the environment. That is if the environment is used with the easy setting. The hard and intense settings introduce more entities.

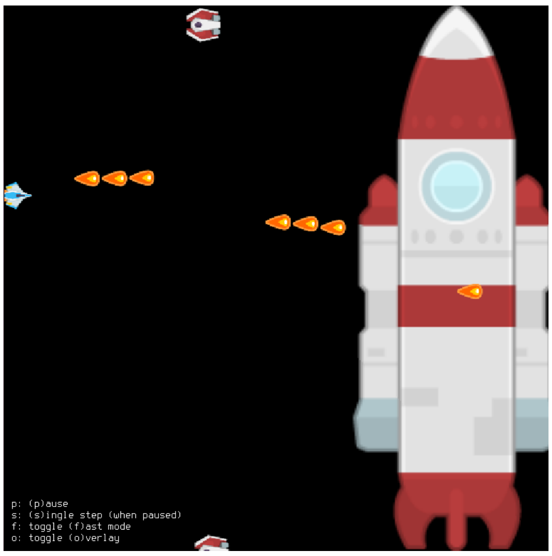


Figure 4: The giant rocket signaling a Starpilot level end.

The player can receive positive rewards by destroying enemy starships and reaching the level end. A reward of 1 is gained for destroying one enemy starship, and a reward of 10 is gained when reaching the level end. The agent receives no negative rewards. To maximize the return, the player must live long enough to reach the level end and shoot as many enemies as possible on his voyage. Reaching the level end, i.e. touching the giant rocket, is achieved by dodging enemy projectiles and dodging enemy starships. We identify a set of three skills required to maximize the return: superset={shoot enemy starships; dodge enemy projectiles; dodge enemy starships}. We split this set of skills into two subsets that serve as the basis for the creation of our sub-tasks: subset1={dodge enemy projectiles; dodge enemy starships} and subset2={shoot enemy starships; dodge enemy starships}. The skills individual to the subsets are "shoot enemy starships" and "dodge enemy projectiles". If we create two sub-tasks based on these skills, we want the main task to equally emphasize both skills "shoot enemy starships" and "dodge enemy projectiles". We find that this is not the case in the original Starpilot game. With the current reward structure and the amount of enemies that can spawn, the player is encouraged to shoot more than to dodge. This is also because shooting enemies makes up for the ability to dodge projectiles (and enemy starships) since the enemies can be destroyed before they shoot the player.

That is why we modify Starpilot in the following way to create our super-task: To encourage the agent to learn dodging equally well as shooting, we reward the agent with a survival bonus of 0.01 after every step the player survives. Moreover, we remove backgrounds and limit the possible themes for enemy starships to two themes. We also limit the theme for the giant rocket to one theme. The comparison between the original game and our super-task can be seen in figure 3.

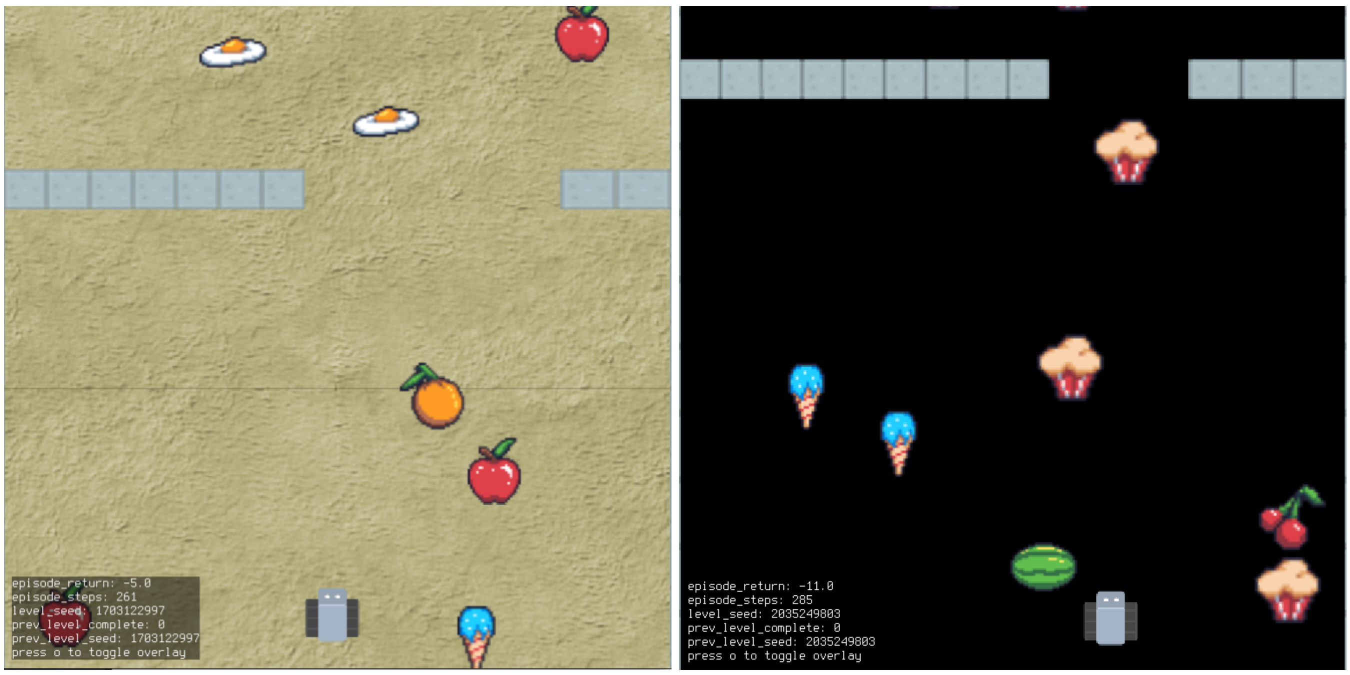
From this super-task, we create the sub-tasks "Starpilot_dodge", which corresponds to subset1, and "Starpilot_shoot", which corresponds to subset2.

Sub-task 1: Starpilot_dodge
To create Starpilot_dodge, we transform the player's shoot action into a no-operation action, essentially removing the action. This leaves the player no other option than dodging to reach the level end. This simple modification changes the action space and the state space (since states cannot include the player's projectiles anymore). In addition, we have to increase the survival bonus from 0.01 to 0.1 per step taken. This must be done because otherwise, the sub-task is not learnable for our RL agents. After all, enemy starships are no reward sources anymore. The agents need more observable rewards to learn the game. Thus, we increase the survival bonus. After applying the modifications, our human oracle confirms that agents trained on the sub-task learn the desired skills. Now, this sub-task is ready for experiments.

Sub-task 2: Starpilot_shoot
To create Starpilot_shoot, we remove the enemy starships' abilities to shoot projectiles. Furthermore, we

remove the survival bonus in this sub-task, and we reduce the maximum enemy spawn group size to 3. We do this so that RL agents trained on the sub-task do not learn to spam shoot all the time but instead aim and move to the enemy starships' height to shoot them intentionally. If we did not do that, agents trained on the sub-task would not learn to shoot enemies on purpose effectively. After applying the modifications, our human oracle confirms that agents trained on the sub-task learn the desired skills. Now, we can train agents on this sub-task, which is ready for experiments.

Fruitbot



```
episode_return: -5.0
episode_steps: 261
level_seed: 1703122997
prev_level_complete: 0
prev_level_seed: 1703122997
press o to toggle overlay
```

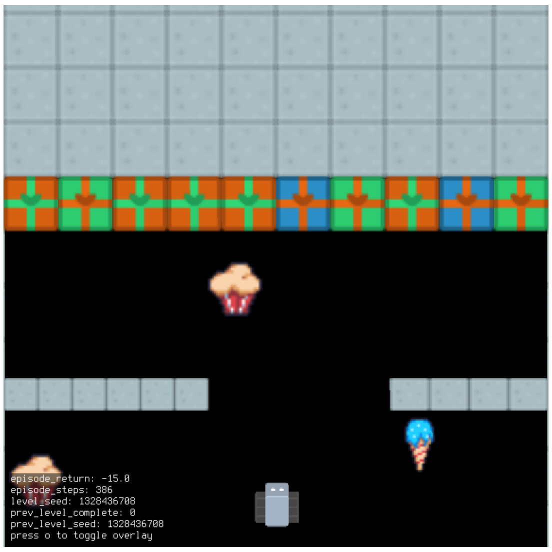
```
episode_return: -11.0
episode_steps: 285
level_seed: 2035249803
prev_level_complete: 0
prev_level_seed: 2035249803
press o to toggle overlay
```

Figure 5: **Left:** The original Fruitbot game before we modify it (easy settings). **Right:** Our super-task. The Fruitbot game after we remove backgrounds, limit fruits to watermelons and cherries and limit treats to popcorn and ice cream. The player controls the little robot and can move it to the left and the right.

Fruitbot is also a scrolling game, but this time, the scrolling is upward instead of sideward. An image of a possible game state can be seen in figure 5. The player controls a robot and is supposed to pick up as many fruits as possible with his robot until he reaches the level end. The player can only move the robot sideways and neither up nor down. The level end is signaled by a line of several presents, as shown in figure 6. Upon touching the presents with his robot, the player successfully completes the level, and a new level begins. On his voyage, the player does not only encounter fruits but also treats, which he must avoid. Unlike in Starpilot, the entities in Fruitbot other than the player cannot move or perform any actions. The fruits and treats spawn randomly inside the world. There are several types of fruits and several types of treats. In one level, the amount of fruits and treats that will spawn is determined at the time of level generation. The minimum number of both fruits and treats is 10, respectively, and the maximum number is 20. They spawn in groups of 2 over the level duration. Upon touching a fruit or a treat, the fruit or treat will disappear. In addition, several walls prevent the player from moving further unless the player navigates the robot through small openings in the walls. If the player touches a wall with his robot, the level ends immediately, and a new level begins.

The level does not end if the player touches a treat with his robot.

The player can receive positive rewards by touching fruits with his robot. He receives a reward of 1 for every touched fruit. He also receives a positive reward of 10 for reaching the level end. If he touches a treat, the player receives a negative reward of -4. To maximize the return, the player must pick up as many fruits as possible, avoid as many treats as possible, and navigate the robot through the openings inside the walls. We identify a set of three skills required to maximize the return: superset={pick up fruits; avoid treats; navigate through openings}. We split this set of skills into two subsets that serve as the basis for creating our sub-tasks: subset1={avoid treats; navigate through the openings} and subset2={pick up fruits, navigate through the openings}.



```
episode_return: -15.0
episode_steps: 386
level_seed: 1328436708
prev_level_complete: 0
prev_level_seed: 1328436708
press o to toggle overlay
```

Figure 6: The line of presents signaling a Fruitbot level end.

To create our super-task, we remove the background from the original game and limit the types of fruits to watermelons and cherries and the types of treats to ice cream and popcorn. We keep the reward structure and the spawn number of fruits and treats of the original game.

From this super-task, we create the sub-tasks ”Fruitbot_treats”, which corresponds to subset1, and ”Fruitbot_fruits”, which corresponds to subset2.

Sub-task 1: Fruitbot_treats
To create Fruitbot_treats, we remove the fruits from the game. Then, we reduce the negative rewards for touching a treat to -3, and we introduce a survival bonus of 0.05 per step, similar to what we do in Starpilot. We must do this to make the sub-task learnable for our RL agents. Moreover, we increase the total number of treats that can spawn in one level. We increase the minimum number to 20 and the maximum to 44. This lets an RL agent observe more (negative) rewards and increases the learnability of the sub-task. After applying the modifications, our human oracle confirms that agents trained on the sub-task learn the desired skills. Now, this sub-task is ready for experiments.

Sub-task 2: Fruitbot_fruits
To create Fruitbot_fruits, we remove the treats from the game. Then, we increase the total number of fruits that can spawn in one level. We increase the minimum number to 40 and the maximum to 40. We do this for the same reason we do it in Fruitbot_treats: to increase the learnability of the sub-task by letting agents observe more rewards. After applying the modifications, our human oracle confirms that agents trained on the sub-task learn the desired skills. Now, this sub-task is ready for experiments.

4.1.2 Model Architecture and Learning Algorithm

We train our agents with PPO (2.2) and use the same model architecture for the actor network and the critic network, except for the respective last layer output dimensions. The model architecture consists of three convolutional layers followed by two fully connected linear layers and rectified linear units between every layer. It is the same architecture as the one introduced by Mnih, Kavukcuoglu, Silver, Rusu, et al. (2015), referred to as NatureCNN. This is a small architecture, but it can learn our super-tasks and sub-tasks. Note that we only use the easy environment settings from Procgen Benchmark. The model is not capable of learning Procgen games with hard or intense difficulty settings, but since we prefer using the easy environments, the model architecture suites perfectly. We could use a bigger architecture, but we see no advantage. Again, we investigate our hypotheses on model merging in deep RL, and we do not try to develop the best-performing RL agent. As long as we can ensure the model learns all the desired skills, that is all we need. In fact, using a smaller model architecture instead of a bigger one saves a lot of computational costs and significantly speeds up the experiments. The model architecture for the actor network is displayed in figure 7.

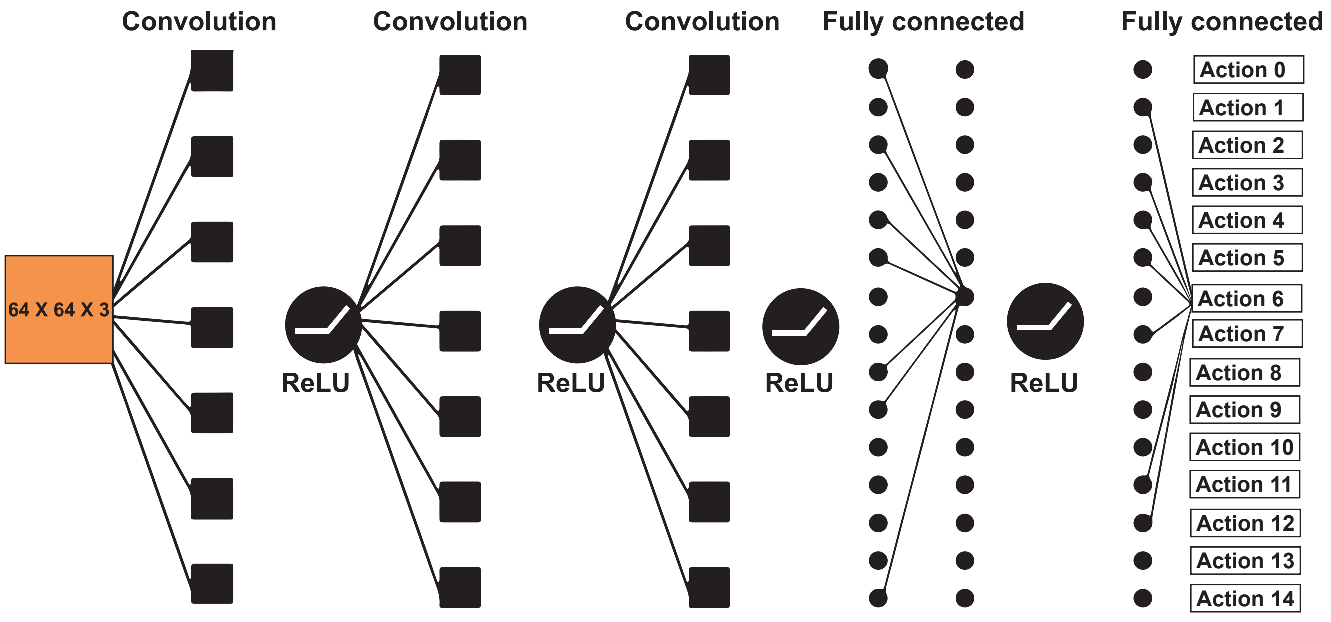


Figure 7: The model architecture we use consists of three convolutional layers and two linear layers. The output in this figure is specific to the actor network and resembles the action space. The input is a $64{\times}64{\times}3$ RGB image received from a Procgen game.

The network inputs consist of $64{\times}64{\times}3$ RGB images that represent the environment states of the Procgen environments. We do not perform any preprocessing on the images. They are forwarded to the models as they are received from the Procgen environments. We find that preprocessing (e.g. stacking frames or grayscaling the images) does not lead to any performance changes in the models. The first convolutional layer has the output dimension 32, kernel size 8 and stride 4. The second convolutional layer has the output dimension 64, kernel size 4 and stride 2. The last convolutional layer has the output dimension 64, kernel size 3 and stride 1. The output of the last convolutional layer is flattened before the first fully connected layer receives it as input. No padding is used by any convolutional layer. The first fully connected layer has the output dimension 512. The output dimension of the last linear layer differs between the actor and the critic. The actor network's last linear layer has the output dimension 15, corresponding to the 15 possible actions the player can perform in every Procgen environment. The output with the largest value is predicted to lead to the most return. The critic network's last linear layer has the output dimension 1, corresponding to the estimated value of the input state.

During training, we use vectorized environments, running 64 instances of one environment simultaneously. That means the agent plays 64 game instances at the same time. This significantly speeds up the training process. Also, using multiple environments in parallel can help stabilize training. It makes it less likely for training to get stuck in suboptimal solutions. Additionally, it aids in exploration by exposing the agent to a broader range of experiences. Using 64 environments in parallel means the batch dimension of the model input is 64.

We use the PPO implementation from (Raffin et al., 2021), which allows for seamless integration of Procgen environments. The used optimization algorithm is Adam (Kingma et al., 2015), an SGD optimization algorithm. With Adam, we use no weight decay, and the betas are $\beta_1 = 0.9$, $\beta_2 = 0.999$. Table 4.1 lists the PPO hyperparameters we use.

PPO Hyperparameters	
Learning rate	.0005
Discount factor	.999
# Time steps for each environment per update	256
# Epoch	3
Minibatch size	2048
Clip range actor	.2
Clip range critic	None
Entropy coefficient for loss calculation	.01
Value function coefficient for loss calculation	.5
GAE_lambda*	.95
# Workers	1
# Environments per worker	64
Normalize advantage	Yes
Max value for gradient clipping	.5

Table 4.1: The hyperparameters we use in combination with the PPO implementation in (Raffin et al., 2021). *GAE_lambda refers to the factor for the trade-off of bias vs variance for the Generalized Advantage Estimator.

4.1.3 Model Training

In our research inquiry, models are a means to an end, just like our environments. We use them to investigate something else: model merging in deep RL, specifically, our hypothesis. That is why model training and preparation of them belong in our setup section. Before we merge models, it is relevant to know whether they can even perform their sub-task; this is what we confirm in this section.

We merge the actor networks resulting from PPO and do not merge the critic networks, even though this may be interesting in other model merging scenarios (we discuss this in the next chapter 5). From now on, when we use the word model, we refer to the actor networks resulting from PPO training runs.

In all cases, we train each model for 25 million steps, as every agent learns all the desired skills with that amount of training experience. We use seeds for model initialization and environment procedural generation, as explained in our methodology section (3.4). To recap: We always use elements from $\{0, 1, 2, 3, 4\} \times \{0, 1, 2, 3, 4\}$, where the first number corresponds to the model initialization seed and the second to the environment seed. We do not set a limit to the number of levels that can be generated during one training run, which means the agents can encounter an infinite amount of different procedurally generated episodes. We do this so the agents generalize well and do not memorize specific levels without learning the game. We name the models by the following schema: *taskname+SeedCombination*, whereby we shorten Starpilot to SP and Fruitbot to FB.

Figure 8 shows that our models trained on the Starpilot sub-tasks learn the games successfully during training. It reports the rolling mean of the past 100 episodes (levels) at each timestep, whereby the curves are smoothed using a Gaussian filter with sigma=600. Training is relatively consistent between all models trained on the same sub-task. SP_dodge11 and SP_shoot23 have gotten unadvantageous seed combinations, but they learn their sub-tasks nonetheless. The Starpilot_shoot training runs, in particular, exhibit dissimilar learning curves in the middle of training, but they converge to similar solutions at the end of training. The mean returns are very different between sub-tasks, but this is normal because they are different tasks, and the reward structures are different. The tables 4.2 and 4.3 show the testing results of each model when running them for 5000 episodes on their respective sub-task, using a level generation seed that was not used during training by any model. The models with the unadvantageous seeds achieve the worst results. However, all models consistently achieve good returns and generalize well, considering the seed for level generation during training is different from training level seeds. The models are ready for merging.

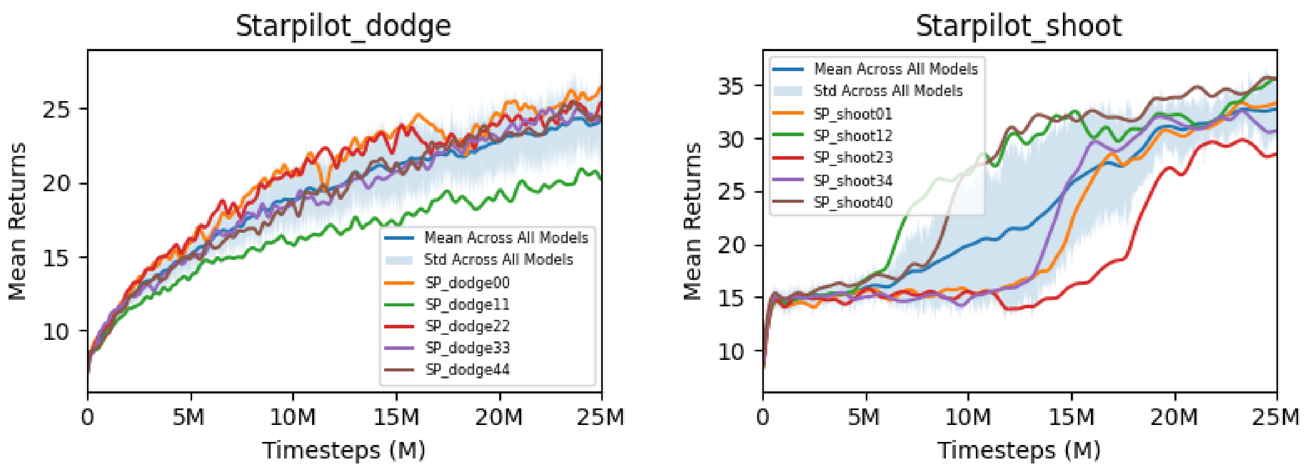


Figure 8: Every model trained on a Starpilot sub-task achieves good returns at the end of training, and learning happens successfully. All models trained on the same sub-task reach similar mean returns at the end of training, although there are visible differences between the best-performing model and the worst-performing model.

Model	Mean	Std
SP_dodge00	26.16	11.27
SP_dodge11	20.54	10.73
SP_dodge22	25.02	11.38
SP_dodge33	24.67	11.10
SP_dodge44	24.65	11.30
Overall	24.21	11.32

Table 4.2: The models trained on Starpilot_dodge learn the game and generalize well. The table shows mean return and std over 5000 testing episodes for each model and across all models (Overall), whereby each model was tested on the Starpilot_dodge game using seed 5 for level generation.

Model	Mean	Std
SP_shoot01	33.33	8.18
SP_shoot12	33.16	8.66
SP_shoot23	29.43	8.26
SP_shoot34	32.35	10.12
SP_shoot40	35.93	8.26
Overall	32.84	8.97

Table 4.3: The models trained on Starpilot_shoot learn the game and generalize well. The table shows mean return and std over 5000 testing episodes for each model and across all models (Overall), whereby each model was tested on the Starpilot_shoot game using seed 5 for level generation.

Similarly to figure 8, figure 9 shows that our models trained on the Fruitbot sub-tasks learn the games successfully during training. The training runs are very similar, and the models for both sub-tasks achieve almost identical mean returns at the end of training, suggesting that learning the task is easy for the models and that they achieve close to maximum returns. This is not the case for the models trained on the Starpilot sub-tasks, which is expected, considering the Fruitbot sub-tasks are simpler games. The tables 4.4 and 4.5 show the testing results of each model when running the models for 5000 episodes on their respective sub-task. All models consistently achieve good returns, and there are no significant differences between them. This means they generalize well. They also all consistently achieve the maximum episode lengths, which we do not visualize here, but we observe this in the testing data, which enforces our suspicion that they achieve close to maximum returns. The models are ready for merging.

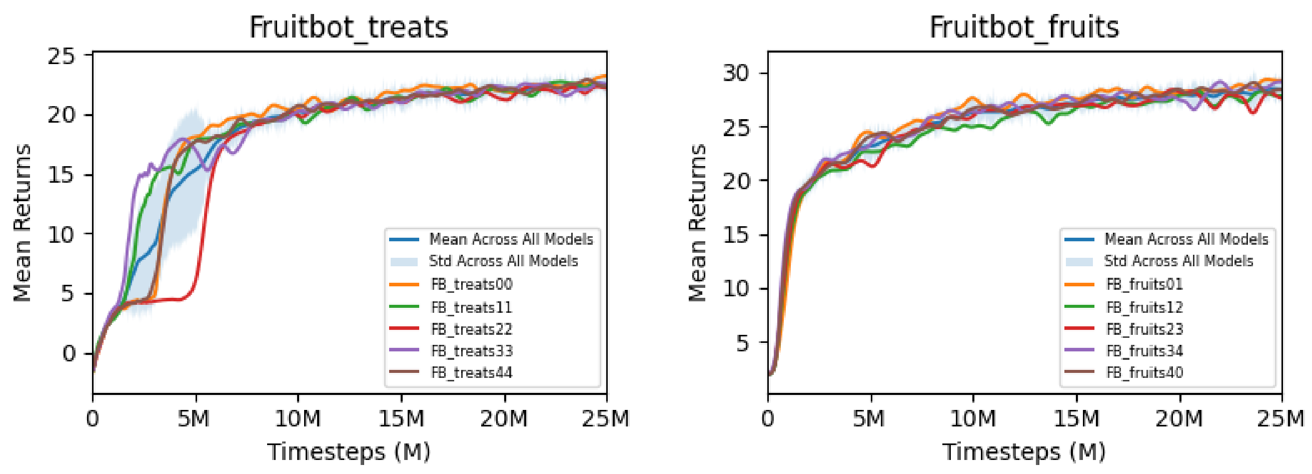


Figure 9: Every model trained on a Fruitbot sub-task achieves good returns at the end of training, and learning happens successfully. All models trained on the same sub-task reach very similar mean returns at the end of training.

Model	Mean	Std
FB_treats00	23.17	6.52
FB_treats11	22.73	7.26
FB_treats22	22.43	7.56
FB_treats33	22.58	7.65
FB_treats44	22.52	7.42
Overall	22.68	7.30

Table 4.4: The models trained on Fruitbot_treats learn the game and generalize well. The table shows mean return and std over 5000 episodes for each model and across all models (Overall), whereby each model was tested on the Fruitbot_treats game using seed 5 for level generation.

Model	Mean	Std
FB_fruits01	30.02	7.85
FB_fruits12	28.35	7.16
FB_fruits23	27.58	8.38
FB_fruits34	28.98	7.98
FB_fruits40	28.68	7.76
Overall	28.72	7.88

Table 4.5: The models trained on Fruitbot_fruits learn the game and generalize well. The table shows mean return and std over 5000 episodes for each model and across all models (Overall), whereby each model was tested on the Fruitbot_fruits game using seed 5 for level generation.

Finally, figure 10 reports the training curves for our upper bounds. We again report the rolling mean returns of the past 100 episodes. The training is stable across all five seeds, with little standard deviation. All models achieve similar returns at the end of training. The upper bounds' test performance is reported in the results section and is used to evaluate the merged models' performance.

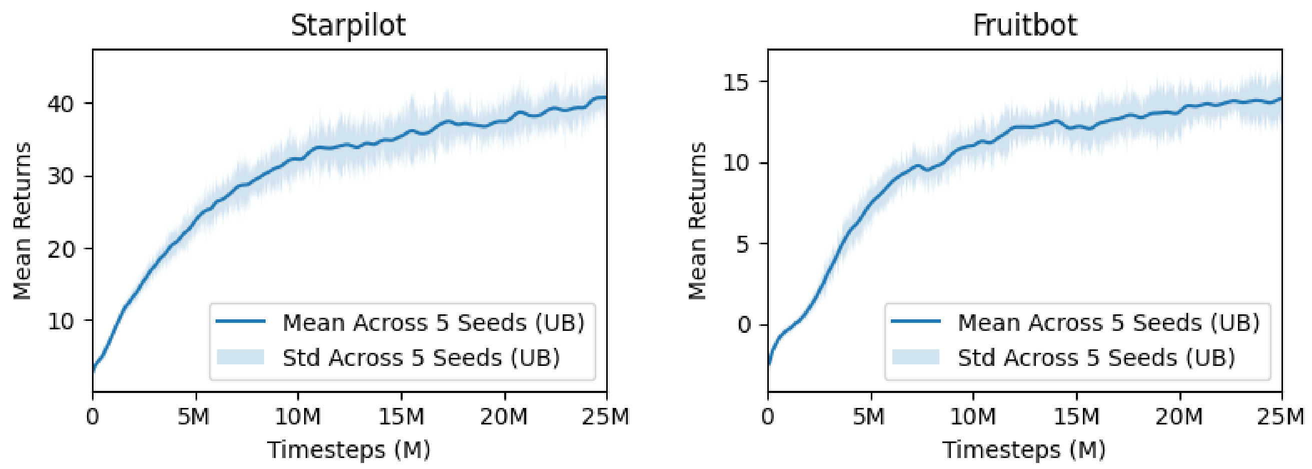


Figure 10: The upper bounds for both super-tasks. Training is consistent for both super-tasks, and all models learn their respective game.

4.2 Results

We report merging results for each super-task separately. In addition to comparing the merged models to the UBs, we also compare them to a random agent, which will let us make better conclusions in the case that the merging performs very poorly. We present extra metrics when they add more insight. We use three different alphas for Git Re-Basin merging and try out permuting both source models. Remember that the alpha determines the individual weighting of each model in the averaging process. We use alphas 0.25, 0.5 and 0.75. An alpha value of 0.25 weights the permuted model (model B) more heavily, and an alpha value of 0.75 weights the unpermuted model more heavily. Using these alpha values will uncover whether weighting one sub-task more heavily leads to better or worse results, which in turn will give us more insights.

4.2.1 Starpilot

Tables 4.6 and 4.7 show the mean returns and stds of the Git Re-Basin solutions when letting them play the super-task (Starpilot) for 5000 episodes. One row corresponds to the results of one merging round, which means five merges of differently seeded source models using the same procedure, as explained in section 3.4. We do not find significant differences between the individual merges of one merging round, so we do not report the individual results. Merging seems to produce consistent solutions. The merged solutions perform significantly worse than the UB, and they perform similarly to the random agent, almost achieving no returns. Table 4.8 shows the mean returns and stds of the weight averaging solutions when letting them play the super-task for 5000 episodes, and they also perform terribly, whether the source models were trained from the same initialization or not. At first sight, it becomes evident that our hypothesis is wrong. However, some

Git Re-Basin		
Model A=SP_dodge, Model B=SP_shoot		
Model	Mean	Std
UB	33.33	8.18
$\alpha = 0.5$; same init	3.03	2.74
$\alpha = 0.25$; same init	3.24	2.90
$\alpha = 0.75$; same init	3.47	2.76
$\alpha = 0.5$; diff init	2.05	1.94
$\alpha = 0.25$; diff init	1.93	1.72
$\alpha = 0.75$; diff init	2.85	2.41
Random Agent	2.31	2.18

Table 4.6: Git Re-Basin produces terrible solutions when merging source models trained on different Starpilot sub-tasks, achieving mostly worse returns than solutions by weight averaging. This table reports results where Model A is always the one trained on the SP_dodge sub-task. ”same init” corresponds to merging round 1, and means that the source models had the same initialization. ”diff init” corresponds to merging round 2.

Git Re-Basin		
Model A=SP_shoot, Model B=SP_dodge		
Model	Mean	Std
UB	33.33	8.18
$\alpha = 0.5$; same init	3.03	2.74
$\alpha = 0.25$; same init	3.47	2.76
$\alpha = 0.75$; same init	3.24	2.90
$\alpha = 0.5$; diff init	2.11	1.77
$\alpha = 0.25$; diff init	3.10	2.34
$\alpha = 0.75$; diff init	1.71	1.68
Random Agent	2.31	2.18

Table 4.7: Git Re-Basin produces terrible solutions when merging source models trained on different Starpilot sub-tasks, achieving mostly worse returns than solutions by weight averaging. This table reports results where Model A is always the one trained on the SP_shoot sub-task. ”same init” corresponds to merging round 1, and means that the source models had the same initialization. ”diff init” corresponds to merging round 2.

highly interesting subtleties in the results let us gain relevant insights into our suspicions, theories, and other hypotheses that built up our main hypothesis.

Git Re-Basin merging with alpha=0.5 performs worse than weight averaging when the source models were trained from the same initialization and when they were trained from different initializations. This is unexpected. We expected Git Re-Basin to merge more similar features and that it would thus lead to better results. Now the opposite happened. Does this mean Git Re-Basin merged less similar features than weight averaging? This is unreasonable since it does the same thing as weight averaging except that it aligns features before averaging. Hence, it is impossible for Git Re-Basin to merge less similar features than weight averaging. We have an idea why the results are worse for Git Re-Basin, which does not suggest Git Re-Basin merges less similar features than weight averaging.

Weight Averaging		
Model	Mean	Std
UB	33.33	8.18
Same Init	3.05	2.78
Diff Init	2.53	2.21
Random Agent	2.31	2.18

Table 4.8: Weight averaging produces terrible solutions when merging source models trained on different Starpilot sub-tasks. ”same init” corresponds to merging round 1, and means that the source models had the same initialization. ”diff init” corresponds to merging round 2.

A model trained on Starpilot_shoot learns to move to the same height as the enemy starships to shoot them, whereas a model trained on Starpilot_dodge learns to move to a different height than the enemy starships so as not to touch them. Now, let us assume both models learn similar features that make them recognize enemy starships when activated. One model moves towards the enemy starships when recognizing them, and the other model moves away when recognizing the enemy starships. Merging these two models and these features would result in a model that can recognize enemy starships but does not know what to do when recognizing them. It would probably neither dodge nor move to a position to shoot but perform a random action. While the exact features may be merged, they lead to opposing actions in the source models, which means the merged model cannot be expected to behave as desired. We assumed that merging the same features automatically leads to the desired model output, but this may not be the case. That would explain why Git Re-Basin solutions perform worse than weight averaging solutions. It may be better to merge unrelated features than to merge features that lead to opposing actions. That also supports our idea that similar features are learned between models trained on different sub-tasks. However, that does not support our hypothesis that merging similar features will automatically lead to a solution that can solve the super-task. Models trained directly on the super-task probably learn different behaviors for the same feature, depending on other features. If the enemies are very close to the agent, this may result in the agent moving away, and if they are far away, the agent moves to the same height to shoot them. Models trained on Starpilot_dodge do not learn these different behaviors because they are not needed to solve their task. That means for our merging scenario, not only the similarity of features is relevant, but also the model outputs they lead to in the source models. Features that lead to specific actions in models trained on the super-task can lead to different actions in models trained on sub-tasks of that super-task.

Now, are there any features the models should learn in both sub-tasks that lead to the same actions? We think that the answer is almost none. The skill that is shared between the sub-tasks is ”dodge enemy starships”, and we just explained how models trained on Starpilot_shoot learn to shoot enemy starships for the most part and probably only to avoid them when it is not possible to shoot them in time, whereas models trained on Starpilot_dodge learn to avoid them in every scenario. Moreover, models trained on Starpilot_shoot cannot encounter enemy projectiles during training, so they cannot learn any features of them. So if a model trained on Starpilot_dodge learns features to recognize enemy projectiles, there are no appropriate features to merge them with, resulting in a loss of the ability to dodge enemy projectiles. How about the skill to shoot enemy starships? The models trained on Starpilot_dodge cannot shoot, but they can give the shooting action input, just that it is a no-operation action for them. This means they probably learn to perform this action in some situations because no-operation actions can be useful. Thus, models trained on different Starpilot sub-tasks learn to shoot in entirely different situations because the models trained on Starpilot_dodge ”think” that shooting is to do nothing. Merging the models then cannot be expected to result in a model that knows when to shoot and when not to. Thus, no skills are left for the merged models to learn properly.

However, the fact that merges of source models with the same initialization consistently perform better than those of source models with different initializations, indicates that there are a few similar features learned between the different source models that lead to the same actions. This confirms our theory that models trained from the same initialization and on different sub-tasks learn more similar features than models trained from different initializations and on different sub-tasks. This is also supported by the fact that Git Re-Basin merges of source models with the same initialization have the same results, no matter what model was permuted, whereas the Git Re-Basin merges of source models with different initializations have different results, depending on which model was permuted. It suggests that Git Re-Basin can clearly find similar features between models trained from the same initialization on different sub-tasks and cannot do so when the models have a different initialization, and matches features that are less similar.

Furthermore, we observe that Git Re-Basin solutions that weighted one of the source models more in the averaging process produced the best results. This aligns with our previous statements of source models trained on different sub-tasks learning the same features but performing different actions based on them. Weighting one model more in the averaging process increases the probability of performing one of the two actions rather than not knowing what to do and performing an utterly wrong action.

The merged models that weighted the source model trained on Starpilot_dodge more produced better results than the merged models that weighted the source models trained on Starpilot_shoot more. We think that this is due to the bias in the sub-tasks. We already mentioned that models trained on Starpilot_dodge learn to perform the shoot action nonetheless and that models trained on Starpilot_shoot cannot learn features of enemy projectiles. For these reasons, we think it is reasonable to assume that models trained on Starpilot_dodge are better at performing the super-task Starpilot before merging than the models trained on Starpilot_shoot.

We investigate this suspicion by letting the models trained on the sub-tasks play Starpilot for 1000 episodes. The results are in table 4.9. The models trained on Starpilot_dodge achieve slightly higher returns, suggesting our suspicion is correct.

Model	Mean	Std
SP_dodge	7.36	6.26
SP_shoot	6.69	5.78

Table 4.9: Models trained on SP_dodge perform better on the super-task than the models trained on SP_shoot. The table reports mean returns and stds over 1000 episodes across 5 different seeds after letting the models play the Starpilot game.

Table 4.9 also tells us that the models trained on the sub-tasks are better at performing the super-task than the merged models, which supports our idea that models before merging learn valuable skills for solving the super-task but lose those skills due to similar features being matched that lead to opposing actions in the source models, ultimately resulting in a loss of those skills.

4.2.2 Fruitbot

Git Re-Basin		
Model A=FB_treats, Model B=FB_fruits		
Model	Mean	Std
UB	13.99	7.12
$\alpha = 0.5$; same init	-2.70	4.08
$\alpha = 0.25$; same init	-3.83	5.13
$\alpha = 0.75$; same init	-1.41	5.51
$\alpha = 0.5$; diff init	-2.74	4.18
$\alpha = 0.25$; diff init	-5.00	6.06
$\alpha = 0.75$; diff init	0.10	6.13
Random Agent	-2.64	3.19

Table 4.10: Git Re-Basin produces terrible solutions when merging source models trained on different Fruitbot sub-tasks, some solutions performing worse than a random agent. This table reports results where Model A is always the one trained on the FB_treats sub-task. ”same init” corresponds to merging round 1, and means that the source models had the same initialization. ”diff init” corresponds to merging round 2.

Git Re-Basin		
Model A=FB_fruits, Model B=FB_treats		
Model	Mean	Std
UB	13.99	7.12
$\alpha = 0.5$; same init	-2.70	4.08
$\alpha = 0.25$; same init	-1.41	5.51
$\alpha = 0.75$; same init	-3.83	5.13
$\alpha = 0.5$; diff init	-2.74	4.19
$\alpha = 0.25$; diff init	-0.43	5.73
$\alpha = 0.75$; diff init	-4.94	6.21
Random Agent	-2.64	3.19

Table 4.11: Git Re-Basin produces terrible solutions when merging source models trained on different Fruitbot sub-tasks, some solutions performing worse than a random agent. This table reports results where Model A is always the one trained on the FB_fruits sub-task. ”same init” corresponds to merging round 1, and means that the source models had the same initialization. ”diff init” corresponds to merging round 2.

Tables 4.10 and 4.11 show the mean returns and stds of the Git Re-Basin solutions when letting them play the super-task (Fruitbot) for 5000 episodes. One row corresponds to the results of one merging round, which means five merges of differently seeded source models using the same procedure, as explained in section 3.4. We do not find significant differences between the individual merges of one merging round, so we do not report the individual results. Merging seems to produce consistent solutions. The merged solutions perform significantly worse than the UB, similar to how the merged solutions performed compared to the UB in Starpilot. However, this time, there are apparent differences in the performances of the merged solutions and the random agent. Sometimes, the merged solutions perform worse, sometimes better than the random agent. The differences in the mean returns between merging rounds are greater than the ones in Starpilot. However, that mainly applies to the Git Re-Basin solutions that used alpha=0.25 or alpha=0.75. The Git Re-Basin merges that used alpha=0.5 are performing very similarly to the random agent and the weight averaging solutions, which can be seen in table 4.12. Once again, it becomes evident at first sight that our hypothesis is incorrect. Nevertheless, once again, there are highly interesting subtleties in the results that give relevant insights.

Git Re-Basin merges that weighted the models trained on Fruitbot_treats more heavily, achieved much better returns than the ones that weighted the models trained on Fruitbot_fruits more heavily. In addition to that, the merged models, which weighted the models trained on Fruitbot_fruits more heavily, performed worse than the random agent. Considering these results, the suspicion arises that models trained on Fruitbot_fruits mistake treats for fruits and that models trained on Fruitbot_treats mistake fruits for treats.

We investigate this suspicion by letting the models trained on one sub-task play the other sub-task for 1000 episodes. The results confirm our suspicion. The models trained on Fruitbot_treats receive a mean return of 12.83 when playing the Fruitbot_fruits game, and the mean episode length is 410.11. The maximum episode length in Fruitbot is 420, which means the models almost always reach the level end, which gives them +10 rewards. They pick up almost no fruits since the total mean return is only 12.83. Similarly, the models trained on Fruitbot_fruits receive a mean return of -28.51 when playing the Fruitbot_treats game, and the mean episode length is 352.12. This is clear evidence that they pick up many treats on purpose on their way to the level end. There is no chance that they accidentally pick up that many treats.

Weight Averaging		
Model	Mean	Std
UB	13.99	7.12
Same	-2.66	4.01
Diff Init	-2.51	3.81
Random Agent	-2.64	3.19

Table 4.12: Weight averaging produces terrible solutions when merging source models trained on different Fruitbot sub-tasks. ”same init” corresponds to merging round 1, and means that the source models had the same initialization. ”diff init” corresponds to merging round 2.

This means that when models are trained on the Fruitbot game, they learn the differences between fruits and treats, and the learned features capture specific details of the respective food types. However, once one food type is gone, which is the case in both sub-tasks, the models learn more generic object features. This makes sense since more specific features are not needed if there is only one type of food in the game.

Now, a similar situation arises to the one with the Starpilot sub-tasks. The models trained on the sub-tasks learn similar features to recognize foods, but one model learns to avoid the foods, and the other model learns to touch them. The activations of the same features in the source models thus again lead to different actions, and the merged solutions of Git Re-Basin that use alpha=0.5 and those of weight averaging learn to behave like a random agent when encountering foods. Once one model is weighted more heavily, however, the merged model learns to either avoid objects more or to pick them up more, resulting in merged models that weigh Fruitbot_treats source models more heavily to achieve the most returns. Since there is the same amount of fruits and treats in one level, and treats give more negative rewards than fruits give positive rewards, an agent that picks up all food can never outperform an agent that avoids all food.

We observe again that the Git Re-Basin merges of models trained from the same initialization produce the exact solutions, no matter which model was permuted, and the Git Re-Basin merges of models trained from different initializations display the opposite. This adds to the legitimacy of the insights mentioned in the previous section.

All models trained on the sub-tasks learn to navigate the little robot through the openings in the walls and reach the level end almost every time. In contrast to the previous examples of picking up fruits or treats and avoiding enemy starships or moving towards them, recognizing walls and openings should lead to the same actions in models trained on different Fruitbot sub-tasks. Hence, based on our hypothesis and our interpretations in the previous paragraphs, nothing should stop the merged models from inheriting the skill of navigating the robot through openings in the walls. Unfortunately, the low returns suggest otherwise. We further investigate this by creating a new Fruitbot game, where only walls and the level end exist, and all foods have been removed. We call this game Fruitbot_empty. Now, we let one of the Git Re-Basin solutions (alpha=0.5) play this game for 1000 episodes. We would expect it to achieve close to mean episode lengths of 420. However, it achieves mean episode lengths of 87.33. The random agent achieves mean episode lengths of 82.58. This means that our merged model did not inherit the skill of navigating the robot through openings in the walls, even though the source models almost perfected that skill.

There are two possibilities: Either the source models did not learn similar features to navigate the robot through openings in the walls, or they did learn similar features to do so, but they were somehow destroyed during merging. Assuming that the source models did not learn similar features to perform the skill, there is one explanation why that may be. Maybe the RL-specific training dynamics cause models to learn dissimilar features to perform the same skills, even if the models are trained from the same initialization. These RL-specific training dynamics are exploration and non-deterministic model outputs during training. We explained in section 2.2 that exploration means that the learning algorithm chooses actions that go against the model's predictions during training to avoid getting stuck in a local optimum. Because of that, the agent learns non-deterministically, and different training runs with the exact prerequisites may lead to the learning of different features. Also, the difference in encountered states of two such training runs could lead to models learning dissimilar features. However, we merged across multiple seeds, and the merged models performed consistently poorly, and no merged model learned to navigate through openings in walls. This suggests that this is not an unlucky event but happens consistently, and we believe that explanation to be unrealistic.

Assuming the source models did learn similar features, but they were lost during merging, there are several explanations for why that may have happened. Wrong features could be averaged, but that should be avoided when applying Git Re-Basin merging. We have another explanation that fits with our other findings and interpretations. In DNNs, the set of features activated to lead to a specific model output can be pretty complex, and the same features are activated in very different situations. Suppose two models are merged where the same activations of the same features lead to opposite actions (avoid starships, move to starships). In that case, this may mess with all of the other feature activation and model output interplays, causing the agent to perform wrong actions in all situations, not just where the activations of the same features lead to opposing actions. Either way, it is difficult to explain this finding, and more research needs to be undertaken to uncover the reasoning behind all of the results.

5 Conclusion & Future Work

Our findings show that our main hypothesis is incorrect, and the considered merging procedures do not enable model merging for modular CRL. However, they also provide valuable insights into why such modular merging is not leading to good solutions and in which cases it could do so.

Merging models, each trained on a different RL sub-task of the same super-task, cannot be equated with merging models, each trained on a different classification dataset, whereby the datasets are disjoint but contain the same classes. The former is inherently more complex, and the model merging procedures we considered cannot be expected to produce well-performing solutions for the super-task in such scenarios.

Labeled datasets that contain the same classes expose models trained on them to the same features. In contrast, a sub-task may not include an entity that is included in other sub-tasks of the same super-task. Thus, models trained on that sub-task very likely do not learn the features required to recognize the missing entity, but models trained on the other sub-tasks do learn these features. Creating sub-tasks from one super-task in RL is more similar to creating two classification datasets from one classification dataset and *not* keeping the same classes in both resulting datasets.

Moreover, while we did find evidence that deep RL models do learn similar features when solving different sub-tasks of the same super-task, we were naive to assume that merging these features would in and of itself mean that the resulting model would behave as desired. That is because activating the same features in models trained on different sub-tasks can result in opposing actions. In classification tasks, this is much less likely, as recognizing a class is the task itself, while recognizing something in an RL environment (like an entity) is only a small part of solving the task. The agent must decide what action to perform based on what he recognizes, which depends on the context of whatever he recognizes within the environment. This context can be vastly different, even in sub-tasks of the same super-task. Hence, it is insufficient to ensure that models trained on different sub-tasks learn the same features. It must be ensured that the activations of the same features entail the same meaning in the context of the respective sub-tasks. We suspect the interplays between activations and model outputs are particularly important for model merging in deep RL and that this is not the case in deep SL.

Ultimately, we can say that merging of models trained on RL sub-tasks to create a model that can perform their super-task is not feasible with the considered merging procedures, except maybe for very particular situations where the agents trained on different sub-tasks are exposed to the same features and where the activations of these features entail the same meaning in the context of the respective task. However, it is arguable whether this can still be considered a CRL scenario.

This is where future work could investigate further. Other super-tasks can be sought, which allow for creating sub-tasks where the activations of the same features entail the same meanings in the respective sub-tasks. Then, our suspicions can be confirmed or disproven. It may also be helpful to step back from merging models in CRL and first investigate the merging of models trained on the same task in deep RL. This could reveal that merging models with the considered procedures in deep RL can generally not be expected to produce well-performing solutions, or it could prove that it can be and that model merging in CRL is feasible with the considered procedures given suitable modifications.

Other procedures than Git Re-Basin that align neurons should be tried in similar scenarios to verify whether our findings are not specific to the used merging procedures. Such procedures can be found in (Y. Li et al., 2016; Singh et al., 2019). There are also new merging procedures that were developed very recently, which are merging models trained on entirely different tasks, even in SL. We did not consider these procedures because they do not fit thematically with the other procedures. They can be found in (Yadav et al., 2023; Stoica et al., 2023).

Model merging could also be used in very different RL scenarios from the ones considered up to now. We mentioned that not only the actor networks resulting from different PPO training runs could be merged, but also the critics. Doing so may be useful when training is resumed after merging and could improve training outcomes. This is similar to the fine-tuning scenarios mentioned in section 3.1.

Bibliography

Abel, David et al. (2023). “A Definition of Continual Reinforcement Learning.” In: *CoRR*.

Ainsworth, Samuel K., Jonathan Hayase, and Siddhartha S. Srinivasa (2022). “Git Re-Basin: Merging Models modulo Permutation Symmetries.” In: *CoRR*.

Atkinson, Craig et al. (2021). “Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting.” In: *Neurocomputing*.

Bellemare, Marc G. et al. (2012). “The Arcade Learning Environment: An Evaluation Platform for General Agents.” In: *CoRR*.

Ben-Iwhiwhu, Eseoghene et al. (2022). “Lifelong Reinforcement Learning with Modulating Masks.” In: *CoRR*.

Bengio, Yoshua, Nicholas Léonard, and Aaron C. Courville (2013). “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation.” In: *CoRR*.

Bishop, Christopher M. (2007). *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer.

Brockman, Greg et al. (2016). “OpenAI Gym.” In: *CoRR*.

Chen, Zhiyuan and Bing Liu (2018). *Lifelong Machine Learning, Second Edition*. Morgan & Claypool Publishers.

Cobbe, Karl et al. (2019). “Leveraging Procedural Generation to Benchmark Reinforcement Learning.” In: *CoRR*.

Crouse, David Frederic (2016). “On implementing 2D rectangular assignment algorithms.” In: *IEEE Trans. Aerosp. Electron. Syst.*

Dietterich, Thomas G. (2000). “Ensemble Methods in Machine Learning.” In: Springer.

Dong, Hao et al. (2020). *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. Ed. by Hao Dong, Zihan Ding, and Shanghang Zhang. Springer Nature.

Draxler, Felix et al. (2018). “Essentially No Barriers in Neural Network Energy Landscape.” In: *CoRR*.

Emmert-Streib, Frank and Matthias Dehmer (2022). “Taxonomy of machine learning paradigms: A data-centric perspective.” In: *WIREs Data Mining Knowl. Discov.*

Entezari, Rahim et al. (2021). “The Role of Permutation Invariance in Linear Mode Connectivity of Neural Networks.” In: *CoRR*.

Frankle, Jonathan et al. (2019). “Linear Mode Connectivity and the Lottery Ticket Hypothesis.” In: *CoRR*.

Ganaie, Mudasir Ahmad et al. (2021). “Ensemble deep learning: A review.” In: *CoRR*.

Garipov, Timur et al. (2018). “Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs.” In: *CoRR*.

Hahn, Peter, Thomas Grant, and Nat Hall (1998). “A branch-and-bound algorithm for the quadratic assignment problem based on the Hungarian method.” In: *Eur. J. Oper. Res.*

Hecht-Nielsen, Robert (1990). “On the algebraic structure of feedforward network weight spaces.” In: *Advanced Neural Computers*. Elsevier, pp. 129–135.

Hinton, Geoffrey E., Oriol Vinyals, and Jeffrey Dean (2015). “Distilling the Knowledge in a Neural Network.” In: *CoRR*.

Howard, Jeremy and Sebastian Ruder (2018). “Universal Language Model Fine-tuning for Text Classification.” In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics.

Jonker, Roy and A. Volgenant (1987). “A shortest augmenting path algorithm for dense and sparse linear assignment problems.” In: *Computing*.

Kaplanis, Christos, Murray Shanahan, and Claudia Clopath (2018). “Continual Reinforcement Learning with Complex Synapses.” In: *CoRR*.

– (2019). “Policy Consolidation for Continual Reinforcement Learning.” In: *CoRR*.

Khetarpal, Khimya et al. (2020). “Towards Continual Reinforcement Learning: A Review and Perspectives.” In: *CoRR*.

Kingma, Diederik P. and Jimmy Ba (2015). “Adam: A Method for Stochastic Optimization.” In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kirkpatrick, James et al. (2016). “Overcoming catastrophic forgetting in neural networks.” In: *CoRR*.

Kusupati, Aditya et al. (2021). “LLC: Accurate, Multi-purpose Learnt Low-dimensional Binary Codes.” In: *CoRR*.

Li, Yixuan et al. (2016). “Convergent Learning: Do different neural networks learn the same representations?” In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Mallya, Arun, Dillon Davis, and Svetlana Lazebnik (2018). “Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights.” In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part IV*. Springer.

Matena, Michael and Colin Raffel (2021). “Merging Models with Fisher-Weighted Averaging.” In: *CoRR*.

McCloskey, Michael and Neal J. Cohen (1989). “Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem.” In: *Psychology of Learning and Motivation*.

McMahan, Brendan et al. (2017). “Communication-Efficient Learning of Deep Networks from Decentralized Data.” In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*. Proceedings of Machine Learning Research. PMLR.

Mendez, Jorge A., Harm van Seijen, and Eric Eaton (2022). “Modular Lifelong Reinforcement Learning via Neural Composition.” In: *CoRR*.

Mensah, Daniel Opoku et al. (2022). “eFedDNN: Ensemble based Federated Deep Neural Networks for Trajectory Mode Inference.” In: *CoRR*.

Min, Sewon, Min Joon Seo, and Hannaneh Hajishirzi (2017). “Question Answering through Transfer Learning from Large Fine-grained Supervision Data.” In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*. Association for Computational Linguistics.

Mnih, Volodymyr, Adrià Puigdomènech Badia, et al. (2016). “Asynchronous Methods for Deep Reinforcement Learning.” In: *CoRR*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, et al. (2013). “Playing Atari with Deep Reinforcement Learning.” In: *CoRR*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, et al. (2015). “Human-level control through deep reinforcement learning.” In: *Nat.*

Molchanov, Pavlo, Arun Mallya, et al. (2019). “Importance Estimation for Neural Network Pruning.” In: *CoRR*.

Molchanov, Pavlo, Stephen Tyree, et al. (2016). “Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning.” In: *CoRR*.

Mundt, Martin et al. (2023). “A wholistic view of continual learning with deep neural networks: Forgotten lessons and the bridge to active and open world learning.” In: *Neural Networks*.

Ohnishi, Shota et al. (2019). “Constrained Deep Q-Learning Gradually Approaching Ordinary Q-Learning.” In: *Frontiers Neurorobotics*.

Pasen, Martin and Vladimír Boza (2022). “Merging of neural networks.” In: *CoRR*.

Prince, Simon J.D. (2023). *Understanding Deep Learning*. MIT Press.
Raffin, Antonin et al. (2021). “Stable-Baselines3: Reliable Reinforcement Learning Implementations.” In: *J. Mach. Learn. Res.*
Rastegari, Mohammad et al. (2016). “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks.” In: *CoRR*.
Ring, Mark B. (1997). “CHILD: A First Step Towards Continual Learning.” In: *Mach. Learn.*
Rolnick, David et al. (2018). “Experience Replay for Continual Learning.” In: *CoRR*.
Schulman, John et al. (2017). “Proximal Policy Optimization Algorithms.” In: *CoRR*.
Singh, Sidak Pal and Martin Jaggi (2019). “Model Fusion via Optimal Transport.” In: *CoRR*.
Soen, Alexander and Ke Sun (2021). “On the Variance of the Fisher Information for Deep Learning.” In: *CoRR*.
Stoica, George et al. (2023). “ZipIt! Merging Models from Different Tasks without Training.” In: *CoRR*.
Stooke, Adam and Pieter Abbeel (2018). “Accelerated Methods for Deep Reinforcement Learning.” In: *CoRR*.
Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press.
Thrun, Sebastian (1996). *Explanation-Based Neural Network Learning. A Lifelong Learning Approach*. Springer New York, NY.
Thrun, Sebastian and Tom M. Mitchell (1995). “Lifelong robot learning.” In: *Robotics Auton. Syst.*
Traoré, René et al. (2019). “DisCoRL: Continual Reinforcement Learning via Policy Distillation.” In: *CoRR*.
Voita, Elena et al. (2019). “Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned.” In: *CoRR*.
Wang, Hongyi et al. (2020). “Federated Learning with Matched Averaging.” In: *CoRR*.
Wang, Xiaoxing et al. (2020). “MergeNAS: Merge Operations into One for Differentiable Architecture Search.” In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. ijcai.org.
Wortsman, Mitchell, Gabriel Ilharco, Samir Yitzhak Gadre, et al. (2022). “Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time.” In: *CoRR*.
Wortsman, Mitchell, Gabriel Ilharco, Mike Li, et al. (2021). “Robust fine-tuning of zero-shot models.” In: *CoRR*.
Wortsman, Mitchell, Vivek Ramanujan, et al. (2020). “Supermasks in Superposition.” In: *CoRR*.
Yadav, Prateek et al. (2023). “Resolving Interference When Merging Models.” In: *CoRR*.
Zhuang, Fuzhen et al. (2019). “A Comprehensive Survey on Transfer Learning.” In: *CoRR*.