Bachelor Thesis
Cognitive Computer Science
Faculty of Technology

Bielefeld University
Social Cognitive Systems
Prof. Dr.-Ing. Stefan Kopp

Cooking FuNny Dreams: Evaluating the FuN and DreamerV3 Algorithms in the Cooperative Cuisine Environment

Luka Finn Born

Supervisors: Florian Schröder
Annika Österdiekhoff

Bielefeld, November 2024

Contents

List of Figures

List of Tables

Abstract

In this thesis, various Reinforcement Learning techniques are applied to the Cooperative Cuisine Environment (Schröder and Heinrich, 2024), inspired by the game 'Overcooked' (Ghost Town Games Ltd., 2016). In this environment, the agent is tasked with cooking specific dishes in a kitchen setting. We both evaluated the FuN (Vezhnevets et al., 2017) and DreamerV3 (Hafner et al., 2024) algorithms while increasing the complexity of the environment along different dimensions, such as layout size or the complexity and count of the recipes. Regarding the layout complexity, we found that the DreamerV3 algorithm could generalize across different layout structures and sizes. We observed that both the DreamerV3 and FuN models were able to deal with a small number of recipes while struggling with an increasing amount. As for FuN, we examined two different methods for incorporating recipes, of which only one was successful in training an agent that completes the task. We were also able to observe structures in the results, which indicated an efficient usage of the hierarchical structure of the FuN architecture. When introducing additional time constraints to the dishes by adding the concept of orders to DreamerV3, however, it failed in the scope of our experiments. In general, we found that increasing the complexity of the environment usually required more detailed reward shaping. The source code of the experiments and the bachelor thesis itself, as well as videos and weights of trained agents, are available at https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba. The modifications to the environment, as well as the Gym wrapper we used can be found on the `rl-bachelor-thesis` branch of the Cooperative Cuisine Environment repository[1].

[1] https://gitlab.ub.uni-bielefeld.de/scs/cocosy/cooperative-cuisine/-/tree/rl-bachelor-thesis

1 Introduction

Reinforcement learning is a powerful machine learning paradigm where agents perform a series of actions in a dynamic environment with the goal of maximizing a reward with many applications, including healthcare, education, and transportation (Li, 2018; Shakya et al., 2023). This thesis explores the feasibility of using reinforcement learning to build an AI agent for the Cooperative Cuisine Environment (CCE) (Schröder and Heinrich, 2024), similar to the game 'Overcooked' (Ghost Town Games Ltd., 2016). In this environment, the agents are tasked with cooking recipes in a simplified kitchen setting. In the default setup of CCE, the built-in score counter is only increased when a submitting a recipe. Only using this indicator as a reward leads to very sparse rewards, making reinforcement learning a challenging task (Ding and Dong, 2020). Also, the high customizability of the environment allows for simple configuration of its complexity and for integrating custom rewards, making this environment a viable choice.

To accomplish this task, we first employ the DreamerV3 algorithm (Hafner et al., 2024), which was designed for mastering various environments with fixed hyperparameters and, therefore, is expected to work without any hyperparameter tuning. Using this model, we experiment with different reward configurations to train an agent that cooks tomato soup on a simple layout and examine the challenges this task provides. To explore the capabilities of this model, we then incrementally increase the complexity of the environment along different dimensions, such as forcing the model to generalize across kitchen layouts, more complex recipes that require individual subtasks, and introducing multiple recipes. Finally, we also introduce the concept of orders, which adds a time dependency to the recipes.

In addition to single-agent reinforcement learning, we briefly touched on Multi-Agent Reinforcement Learning (MARL) using the BenchMARL library (Bettini et al., 2024). Chapter C illustrates the results of the experiments involving MARL. However, due to technical difficulties and the current limitations of the BenchMARL library, we decided

not to investigate this topic further and instead explore the Feudal Network (FuN) (Vezhnevets et al., 2017) algorithm. More precisely, we examine the hierarchical FuN architecture and whether it is advantageous for this environment, especially when using multiple recipes. The incentive behind using a hierarchical architecture is that many recipes in the CCE require identical subtasks for which the same sub-policies can be invoked. With this intention, we explore multiple methods for integrating the concept of recipes into the FuN architecture.

This thesis mainly focuses on answering the following questions:

1. What modifications to the environment and rewards are necessary to train an agent which can serve a dish in a simple setup?
2. What challenges arise when increasing the complexity of the environment along different dimensions?
3. Is the FuN algorithm able to efficiently utilize its hierarchical structure in the context of CCE?

We organized this thesis into six chapters. Chapter 2 discusses prior work relevant to this thesis. Chapter 3 presents the CCE and theoretical background of the methods used. Next, Chapter 4 explains the details of implementation of the algorithms and the modifications to the environment. Chapters 5 and 6 cover the experiments conducted in this thesis and discuss the results. Finally, Chapter 7 concludes this thesis.

2 Related Work

Reinforcement learning has made major advancements in recent years. Especially in model-based reinforcement learning where an explicit world model is used which models the dynamics of the environment. Before, world models have not been accurate enough to match the performance of state-of-the-art model-free algorithms (Hafner et al., 2022). Hafner et al. (2020) introduced the Dreamer model where reinforcement learning is performed purely on trajectories imagined by the world model. In 2021 this concept was further refined with the DreamerV2 model which was able to achieve human-level performance on the Atari 200M benchmark (Hafner et al., 2022). Finally, Hafner et al. (2024) presented the third iteration of the Dreamer algorithm which was able to 'outperform specialized methods across over 150 diverse tasks, with a single configuration' (Hafner et al., 2024).

In the environment we are considering in this thesis, the agent has to complete tasks that can be divided into subtasks, like cutting a tomato or putting a patty on a pan. Hierarchical reinforcement learning methods follow a similar structure, consisting of a multi-level hierarchy of sub-problems or subtasks with different time horizons (Pateria et al., 2021). Vezhnevets et al. (2017) proposed the FuN algorithm, which uses a two-level hierarchy consisting of a manager and a worker. While previous methods like the options framework (Sutton et al., 1999) required explicitly provided sub-goals and 'pseudo-rewards', FuN learns these sub-goals by itself while still maintaining an explicit meaning of the goals (Vezhnevets et al., 2017).

Because of their domain-agnostic performance and seemingly fitting structure respectively, both DreamerV3 and FuN are promising methods for creating an artificial agent for the CCE. The CCE is an environment inspired by the game 'Overcooked' (Ghost Town Games Ltd., 2016). Overcooked is a game with 3D graphics where players are tasked with serving ordered meals cooperatively in a top-down view of a kitchen environment. This game supports various kitchen layouts and recipes. Compared to Overcooked, the

environment we use has simplified 2D graphics. Overcooked-AI (Carroll et al., 2020) is a similar environment where two agents are tasked with cooking soups. In contrast to the environment used in this thesis, soup is the only dish supported in Overcooked-AI. The environment we consider also supports cooking various other recipes like burgers or salads with varying complexity. Carroll et al. (2020) used the Overcooked-AI environment to evaluate the performance of population-based training and self-play. Wang et al. (2020) proposed 'Bayesian Delegation' which is a decentralized multi-agent learning algorithm which uses Bayesian inference to 'infer the hidden intentions of others by inverse planning' (Wang et al., 2020). This method was evaluated in the gym-cooking environment which is similar to CCE. However, both of these works evaluated a multi-agent setting, while in this thesis, we focus on single-agent reinforcement learning.

3 Background and Methodology

In this chapter we present a broad overview of the methods used in this thesis and the environment that is used for the experiments. Section 3.1 covers the environment, Section 3.2 explains the basic principles of reinforcement learning and touches on some basic reinforcement learning algorithms. Section 3.3 and Section 3.4 cover the architecture of the DreamerV3 and FuN algorithms.

3.1 Cooperative Cuisine Environment (CCE)

CCE (Schröder and Heinrich, 2024) is an environment inspired by the game 'Overcooked' (Ghost Town Games Ltd., 2016). In this environment, one or multiple agents are tasked with cooking dishes. The observation received by the agents is a top-down 2D image of the kitchen. In this kitchen, various cooking utensils are placed on a grid. The utensils include stoves, cutting boards, trashcans, various ingredients, and more. The agents can move in all four directions, carry items between grid cells, and interact with them (6 actions in total).

While the environment runs, the agents receive orders specifying which meals shall be cooked. These orders can expire after a certain amount of time. The environment can be played with either a single agent or multiple agents to encourage cooperation. Figure 3.1 shows a screenshot of a CCE session with one agent and a simple layout.

When comparing this environment to both gym-cooking (Wang et al., 2020) and Overcooked-AI (Carroll et al., 2020), the type of observation is very similar, consisting of a 2D top-down view of a kitchen with simple graphics. The Overcooked-AI environment only supports soups consisting of tomatoes and onions, which need to be combined using a special counter. In this environment, those ingredients do not require further processing, such as cutting, before being combined. Like CCE, this environment supports

orders that can be specified in a configuration file. Gym-cooking also supports only three ingredients and supports defining custom recipes by implementing custom Python classes. It is possible to define multiple recipes that can be cooked at the same time. Additionally, only one way of combining ingredients is supported, which is different to CCE where special cooking equipment such as pots or pans can be required for combining ingredients. CCE additionally allows for the configuration of arbitrary recipes and for adding custom ingredients and cooking equipment using YAML configuration files, enabling a wide range of recipes with different complexities without having to modify the source code. Additionally, CCE supports effects like fire which, if not extinguished, spreads across the environment and 'burns' the ingredients. In all three environments, the layouts are configurable using external files.

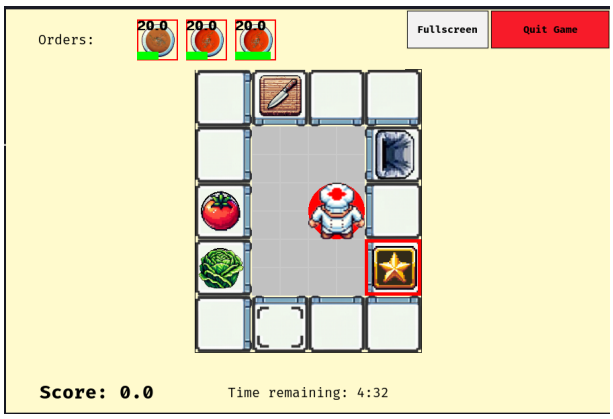


Figure 3.1: Screenshot of the Cooperative Cuisine Environment.
This figure showcases an example screenshot of the Cooperative Cuisine Environment. In the middle, one can see the 2D top-down view of the kitchen with a cutting board, a tomato and salad dispenser on the right of the layout, a cutting board at the top left, a trashcan in the top right, in the bottom left a counter for submitting finished orders and an empty plate dispenser in the bottom left.
In the top left of the screen, the current orders are shown, with the green bar indicating how long this order is active and the number indicating how much the score will be increased when this order is submitted. The current score and remaining time are shown at the bottom of the screen.

3.2 Reinforcement Learning

In this thesis, we analyze the ability of different reinforcement learning algorithms to learn how to solve various tasks in CCE. Reinforcement learning is a field of Machine Learning where an agent acts in an environment while trying to achieve a goal. Usually, this is modeled as a Markov Decision Process (MDP) or Partially Observable Markov Decision Process (POMDP). An MDP is defined by a five tuple (S, A, P_a, R_a) where S is the set of states the environment can adopt, A is the set of actions the agent can perform, $P_a(S_{t+1} = s'|S_t = s)$ is the probability of the environment transitioning from state s to s' when performing action a and $R_a(s, s')$ is the immediate reward after transitioning from state s to s' by performing action a (Uther, 2017). A POMDP additionally consists of a set of observations $\mathcal{O}$ and an observation function $Z(a, s, o)$ (Poupart, 2017).

The agent aims to maximize the expected cumulative reward (the return). The return can be defined with a finite horizon (e.g. 3.1) or with an infinite horizon with an additional discount $\gamma \leq 1$ (e.g. 3.2). A small γ near 0 would prioritize rewards in the near future, while a γ near 1 would put the same importance on all future rewards (Ding, Huang, et al., 2020).

$$R_1(\tau) = \sum_{t=0}^{T} r_t \tag{3.1}$$

$$R_2(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{3.2}$$

Reinforcement learning algorithms can be divided into either model-based or model-free. Model-based algorithms utilize an additional world model which learns to approximate the transition probabilities of the environment. On the other hand, model-free algorithms do not use such a model but instead learn a policy directly from the environment. Using a separate model that approximates the dynamics of the environment has the advantage that the agent does not necessarily have to perform the actions in the real environment but instead can use this world model for reinforcement learning, which can be beneficial when dealing with real-world environments or environments that require much computing power. This principle is used by the DreamerV3 model which is described in more detail in Section 3.3. However, one problem of model-based reinforcement learning is that the agent is limited by the accuracy of the world model. If

the world model is not able to model some dynamics of the environment, the agent will not encounter these in the training process (H. Zhang et al., 2020).

3.2.1 Policy Gradient Methods

In practice, we use a policy π_θ with a set of parameters θ that we aim to learn. Assuming a finite horizon, according to the policy gradient theorem, we can get an approximation of the gradient of the objective function $J(\theta) = \mathbb{E}_\pi[R(\tau)]$ by (Williams, 1992; J. Zhang et al., 2020):

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau) \sum_{t=0}^{T-1} \nabla \log \pi_\theta(s_t, a_t)] \tag{3.3}$$

We can further reduce variance and increase stability by adding a baseline term to the gradient (Ding, Huang, et al., 2020). Using this approximation, we can now perform Gradient Ascent to optimize the objective:

$$\theta_{t+1} = \theta_t + \alpha(R(\tau) - b(s_t))\nabla \log \pi_\theta(s_t, a_t) \tag{3.4}$$

Where $b(s_t)$ is the state dependent baseline and α is the learning rate.

3.2.1.1 Actor-Critic Methods

Actor-critic methods are a special kind of policy gradient methods, consisting of two parts: First, we have the actor, which acts as a policy and predicts actions, aiming to maximize the value predicted by the critic. The critic approximates the value function and evaluates the actions selected by the actor (Konda and Tsitsiklis, 1999).

In Advantage Actor-Critic (A2C), we incorporate the value function as a baseline (Mnih et al., 2016):

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\pi_\theta(s_t, a_t)\mathcal{A}(s, a)],$$

where $\mathcal{A}(s, a) = Q(s_t, a_t) - V(s_t) = R_a(s_t, s_{t+1}) + \gamma \cdot Q(s_{t+1}, a_{t+1}) - V(s_t)$ is the advantage function which describes how good action s_t is in state s_t in comparison to the value of the current state.

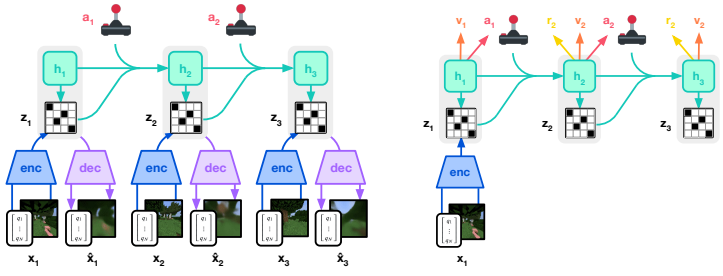
A commonly used family of policy gradient, actor-critic style methods is Proximal Policy Optimization (PPO) (Schulman et al., 2017). This algorithm, among other things, clips the policy update in the objective function to remove the incentive of large policy updates. In the training of PPO, in each iteration of the algorithm, $N \cdot T$ steps in the environment are first collected by N parallel agents. Afterward, the loss is optimized on these $N \cdot T$ timesteps using minibatch SGD or Adam (Schulman et al., 2017).

3.3 DreamerV3

DreamerV3, proposed by Hafner et al. (2024), is the third generation of the Dreamer Algorithm. This algorithm is model-based and uses actor-critic style training.

The world model is a Recurrent State-Space Model (RSSM). The RSSM consists of an encoder that encodes the observations into a stochastic representation, a recurrent sequence model, and a 'Dynamics predictor', which predicts the latent state generated by the encoder from the hidden state of the recurrent neural network. In addition to this, for the training process, a 'Reward predictor' is used to predict the immediate reward; a 'Continue predictor' predicts whether the environment will terminate after the current step, and a decoder aims to reconstruct the original input image.

In the training process of DreamerV3, the learning of the actor and critic networks is performed purely on trajectories predicted by the world model, given an initial observation of the environment. The world model is trained from a dataset of experience collected during training. This dataset contains sequences of images, actions, rewards, and discount factors. Figure 3.2 shows an outline of the architecture and training process.



(a) World Model Learning (b) Actor Critic Learning

Figure 3.2: Illustration of the DreamerV3 model. Taken from Hafner et al. (2024)

DreamerV3 employs multiple strategies to ensure good performance across various domains with fixed hyperparameters. First, a combination of free bits (Kingma et al., 2016) and a small representation loss is utilized when training the autoencoder. Hafner et al. (2024) found that this solves the dilemma of images of 3D environments containing much unnecessary information, while in 2D environments, single pixels may be relevant for solving the task.

Another measure used to cope with various kinds of environments is the normalization of the returns to be approximately in the interval $[0, 1]$, which allows using a fixed entropy scale across different domains. In addition to this, for reconstructing inputs and predicting rewards and returns, the networks learn a version of the targets transformed by the symlog function, which, due to its logarithmic form, compresses very large or small numbers while approximating the identity function near the origin where the values are already small (Hafner et al., 2024).

One aspect of the DreamerV3 model it is being advertised with in the paper is that it was the first algorithm to successfully collect diamonds in Minecraft from scratch without any imitation learning or prior knowledge. To train the model for Minecraft, they used a version with around 200×10^6 parameters and trained it for about 100×10^6 steps. At the end of the training process, the agent was able to successfully find diamonds in about 0.4% of the episodes. To achieve this goal in Minecraft, the agent has to collect various resources like wood, coal, and iron and combine them to create (/ 'craft') tools with which it can destroy (/'mine') diamonds. (Hafner et al., 2024).

This process can be compared to cooking items in the CCE. Here, the agent also needs to combine various items in a specific order to 'cook' the meal which shall be served. However, in CCE, it is much easier for the agent to obtain the required items for cooking as they are clearly visible in the observation the agents receive and do not need to be searched for. Additionally, in CCE, we are only dealing with a 2D environment and a top-down view of the scene, while in Minecraft, on the other hand, the agent must navigate in a complex 3D environment and, in many cases, has to search for specific items like iron, which have to be uncovered by destroying obscuring blocks. Therefore, we assume that this algorithm would fit the CCE scenario.

3.4 FuNs

FuNs are another architecture for reinforcement learning proposed by Vezhnevets et al. (2017). This architecture uses a hierarchical approach consisting of two modules: the manager and the worker. The manager aims to predict goal vectors while the worker tries to predict actions that achieve these goals. The idea behind this hierarchical structure is that the worker learns sub-policies for small action sequences like cutting a tomato while the manager learns a more high-level policy that orchestrates these sub-policies (Vezhnevets et al., 2017). In the context of CCE, this could be especially useful in settings where the agent has to learn to cook multiple recipes, as there are many similarities between different recipes. For example, tomato soup and onion soup require almost identical steps, with only the ingredients being different.

The architecture consists of a shared perceptual module used by the manager and worker, which consists of two convolutional layers and one fully connected layer. The manager then converts this shared perceptual state into its own latent state s_t using another fully connected layer.

The worker uses a standard LSTM (as proposed in Hochreiter and Schmidhuber, 1997) in combination with the goal g_t predicted by the manager to produce the final action distribution. To achieve this, the worker first converts the goal into a much smaller embedding space created by pooling the last c goals of the manager and applying a linear projection ϕ to the pooled embedding. This embedding is then combined with the result of the LSTM using a matrix-vector product, which is then passed into the $\mathrm{softmax}$ function to produce the final action distribution. The linear projection does not use any biases to ensure that the only constant value created by the projection is zero, which prevents the worker from 'ignoring' the goals of the manager, as the goal always influences the policy (Vezhnevets et al., 2017).

The manager uses a dilated LSTM (dLSTM) (defined in 3.5) to predict the goals.

$$\hat{h}_t^{t \mod r}, g_t = \mathrm{LSTM}(s_t, \hat{h}_{t-1}^{t \mod r}; \theta^{\mathrm{LSTM}}) \tag{3.5}$$

A dLSTM is a variation of an LSTM where the hidden state is divided into r substates (/ 'cores'), $h = \{\hat{h}^i | i = 1, \ldots, r\}$. In each step, only one of these cores is updated. This separation allows for longer preservation of memory while still updating the complete

hidden state in each step (Vezhnevets et al., 2017). Figure 3.3 shows a graphical representation of the architecture of the model.

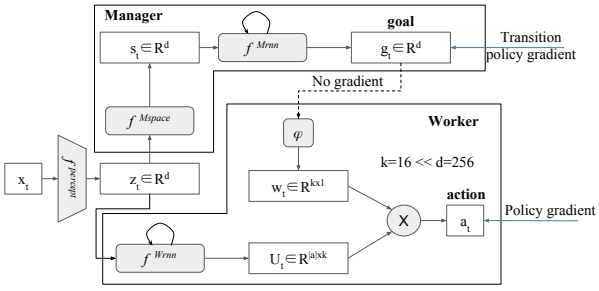


Figure 3.3: The schematic illustration of FuN. Taken from Vezhnevets et al. (2017)

3.4.1 Training

The FuN architecture is not trained end-to-end, as no gradient information flows between the worker and manager. Instead, the worker receives an intrinsic reward defined by 3.6.

$$r_t^I = \frac{1}{c}\sum_{i=1}^{c} d_{\cos}(s_t - s_{t-i}, g_{t-i}) \tag{3.6}$$

where $d_{\cos}$ is the cosine similarity and c is a horizon which defines the temporal resolution of the manager (Vezhnevets et al., 2017).

This enforces the semantic meaning of goals being directional vectors in the latent state space of the manager, indicating in which direction this state should change (Vezhnevets et al., 2017). The purpose of this is that, according to Vezhnevets et al. (2017), it is more feasible for the worker to cause directional shifts in the latent space rather than moving the state to arbitrary new locations. The directional instead of absolute nature of these goals also allows for 'structural generalization' (Vezhnevets et al., 2017), where the same goal vector can invoke sub-policies that are beneficial in various positions in the latent space of the manager.

For the manager and worker, different discount factors γ can be used, allowing for (e.g.) more greedy behavior of the worker while the manager considers long-term rewards. More details on the training process and architecture can be found in Vezhnevets et al. (2017).

4 Implementation

In this chapter, we touch on details of the implementations of the algorithms used in this thesis, as well as the custom implementations made for the experiments. Section 4.1 covers the changes we made to the environment to make it compatible with the different experimental setups, and in Section 4.2, we illustrate the implementation details of the individual algorithms.

4.1 Changes to environment

4.1.1 Discrete Movement

In the default setup, the environment uses a continuous movement system where agents can smoothly move between cells. In settings with multiple agents, they can also push each other around. These push physics require the calculation of distance and collision matrices, which we found to be accountable for a significant part of the time spent calculating the environment steps. However, since this environment does not necessarily require fast reflexes, a much lower temporal resolution than the default a regular player would use suffices for reinforcement learning.

With the temporal resolution chosen for the experiments in this thesis, the agent moves exactly the side length of one cell per step. Therefore, we opted to update the movement system so that the agents are fixed to the grid of the environment, which means that executing the action of going into a particular direction once is equivalent to moving into this direction until the center of the following grid cell is reached. When using the environment in a MARL setting, the agents can no longer push each other away but are prevented from moving to a cell if there is another agent or obstacle. On a system with Ubuntu 24.04.1 and an AMD Ryzen 7 7800X3D, this movement system improved the

average time spent on calculating the movement from around $1.6 \times 10^{-4}s$ to around $5.6 \times 10^{-5}s$.

4.1.2 Gym Wrapper

To allow for reinforcement learning in CCE, a wrapper around the CCE environment implementing the interface of Gym (version $\leq 0.25.2$, Brockman et al. (2016)) environments was created. In this implementation, the reward of the environment is influenced by updating the built-in score when the agent should get a nonzero reward. The reward after each step is the difference between the score before and after the step.

To enable different environment variations during the same run such as multiple recipes or layouts, the gym environment takes an `EnvironmentGenerator` as an argument. These generators produce `EnvironmentPreset` objects, which specify the available items, the layout, and the environment config. We implemented two different implementations of the `EnvironmentGenerator`: the `SinglePresetSelector`, which always selects a specific preset, and the `RandomPresetSelector`, which randomly chooses one of the provided presets.

4.1.2.1 Random Layout Generation

If the selected preset does not contain a layout, a random one will be generated. To achieve this, an empty layout with walls on the edges is created. Then, walls are iteratively added at random locations. To ensure that the agent is still able to reach every empty field, we build a graph from the layout, with the nodes being the empty fields where adjacent fields are connected by edges. Then, we check that the graph is still connected and only add the wall at the selected location if this is the case. Finally, we replace some of the walls still adjacent to empty fields with the counters necessary for cooking the recipe to ensure that the agent can still reach every counter.

4.1.3 Custom Hooks Callbacks for Reward shaping

CCE supports adding custom hook callbacks, which get triggered when certain events occur, e.g., when the agent serves a dish. These hook callbacks can be configured in

the config file of the environment. The following hook callbacks were added to allow for more in-depth reward shaping. The `PutMealOnPlateHookCallback` callback allows for rewarding putting a specific item on a plate. Moving this item from one plate to another does not get rewarded. The `DropOffOnEquipmentHookCallback` callback rewards dropping off a specific item with a provided name on cooking equipment, like a pot or stove, with a name that is provided in the config file. In addition to that, we also implemented the `PunishTrashcanUsageHookCallback` to punish the usage of the trashcan. If the item being put into the trashcan is waste, this will not be punished. Finally, to allow for adaptive rewards depending on the current order, we implemented the `OrderSpecificHookCallback`, which forwards the hook event to a child `HookCallbackClass` if a recipe with a provided name is currently being ordered.

4.1.4 Order Generation and Representation

In experiments involving multiple recipes, an alternative implementation for the generation process of the orders was used, which only creates orders with one specific recipe and ensures that there is an active order at all times.

In the representation we used, each recipe r_i has a fixed ID i. If enabled in the environment parameters, the agent receives a second observation $o \in \mathbb{R}^n$ with $o_i = 1$ and $o_{j \neq i} = 0$, where n is the total number of recipes.

In setups where orders exist, the observation o which the agent receives in addition to the rendered image of the environment is given by:

$$o_j = \begin{cases} 1, & \text{if } r_j \text{ is ordered} \\ 0, & \text{else} \end{cases} \tag{4.1}$$

In this representation, however, the timing of the orders is not considered.

4.2 Algorithms

To perform the experiments, we used the implementations of the algorithms depicted in this section. For the DreamerV3 experiments, we opted for the implementation by

Hafner et al. (2024). This implementation is written in Python and uses the TensorFlow (Martín Abadi et al., 2015) framework. We modified the training process in this thesis to allow for more in-depth experiments.

For the experiments with FuN, we used the PyTorch (Paszke et al., 2019) implementation provided by Weitkamp (2020). This implementation differs from the original paper (Vezhnevets et al., 2017) in a few aspects. First, A2C n-step learning is used instead of A3C (Mnih et al., 2016), which was used for the experiments in the original paper (Vezhnevets et al., 2017). Also, the learning rate is not annealed here (Weitkamp, 2020). The perceptual module additionally implements an MLP architecture as an alternative to the architecture described in Vezhnevets et al., 2017, which consists of two fully connected layers, each with 64 hidden units. Each layer is also followed by a ReLU activation function (Weitkamp, 2020).

5 Experiments

In this chapter, we cover the various experiments we conducted and their results. Section 5.1 covers the hardware setup used for running these experiments. Sections 5.2 and 5.3 cover the experiments involving the DreamerV3 and FuN algorithms, respectively.

5.1 Hardware setup

The experiments in this thesis were executed on the CITEC GPU Cluster of the University of Bielefeld. All experiments were performed on nodes with either an NVIDIA A40 with 48 GB of VRAM, an NVIDIA TESLA P-100 with 16 GB of VRAM, or an NVIDIA GTX 1080ti with 11 GB of VRAM. The runs for the DreamerV3 model were mainly performed on nodes with an NVIDIA A40, while for the runs involving FuNs, we mostly used nodes with an NVIDIA GTX 1080ti, as this is a much smaller model that does not need as many resources. Due to inconsistencies and slightly different available resources for each run, it is not possible to make an in-depth comparison of the training times.

5.2 DreamerV3

Regarding the DreamerV3 algorithm, we started with a simple 4 by 4 layout with a trashcan, cutting board, pot, salad tomatoes, and a serving window. The agent is tasked with cooking tomato soup and submitting it to the serving window. In each step, the agent receives a 64 by 64 pixel RGB image as an observation. Figure A.1 shows an example of such an observation. The environment terminates automatically after 300 steps, independent of what the agent does.

5.2.1 Basic Reward Shaping

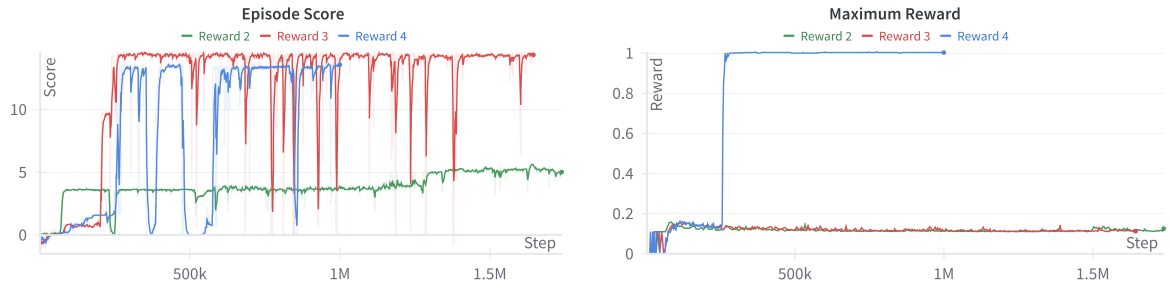
In our experiments, we tried multiple different reward configurations. We used the layout displayed in Figure A.1 for all experiments in this section.

As a baseline to show the importance of reward shaping in CCE, we performed runs where the agent would only get a reward when it successfully served a finished meal ('Reward 1' in Table A.2). With the layout used in this configuration, the agent has to take at least 33 actions to cook and serve a tomato soup, as described in Section A.1.1. Considering there are six available actions, there are $6^{33} \approx 4.78 \times 10^{25}$ possible action sequences, of which only a few lead to the agent successfully cooking a recipe. This order of magnitude indicates that we would have to run the training for a very large number of steps until a reward is achieved. With this configuration, as expected, the agent was not able to achieve any reward in around 10×10^6 steps in the environment. Figure A.2a shows the episode score of this run.

To increase the chance of the agent successfully submitting a meal by random chance, we also conducted an experiment where the agent would only perform random actions until a reward was first encountered. However, as we can see in the episode score, depicted in Figure A.2b, this led to the same results. This effect highlights that we clearly need to modify the reward further using domain-specific knowledge.

Consequently, we integrated rewards for intermediate steps such as cutting an item, putting an item on cooking equipment (e.g. a pot), and serving the finished meal. When introducing these rewards, we found that we also had to introduce punishments, e.g., for using the trashcan. Figure 5.1 shows the results of the following runs with the different reward configurations from Table A.2. Reward configuration 'Reward 2' (Table A.2) only contains these positive rewards. We can see that the agent managed to achieve non-zero rewards. However, in the maximum reward (Figure 5.1b), we can see that the agent never managed to achieve a reward of 1, which is the reward for serving the finished recipe. In Video D.1.1.1, we can see the agent's behavior which has learned to cut tomatoes and throw them into the trashcan to achieve a reward. Besides the punishment for trashcan usage, we also found that when giving the agent a reward for putting a cut tomato on cooking equipment, we also need to add a punishment for taking the cut tomato out of the pot. Figure 5.1 shows a run with reward configuration 'Reward 3' demonstrating this necessity. Here, we can also see in the maximum reward

that the agent did not manage to complete the task. Video D.1.1.2 shows the agent cutting a tomato and afterward repeatedly adding and removing the cut tomato from the pot. Finally, when adding a punishment to prevent this loophole, we arrived at reward configuration 'Reward 4' (Table A.2). Looking at the maximum reward, we can see that the agent managed to serve tomato soup for the first time at around $265{,}500$ steps.



(a) Episode score

(b) Maximum reward each algorithm iteration

Figure 5.1: Results of the runs with reward configurations from Table A.2

5.2.2 Recipe Complexity

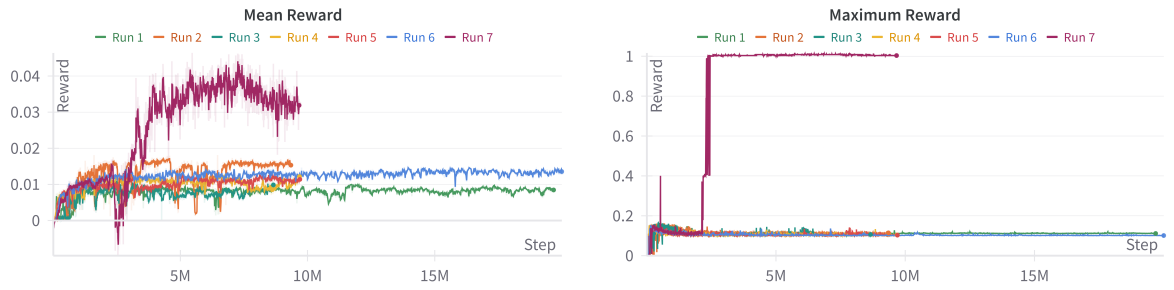
To explore the performance of the DreamerV3 model in environments with more complex recipes, we also conducted experiments where the agent is tasked to serve a burger. While tomato soup only has one ingredient, which must be cut and put into a pot, burgers require salad, tomatoes, a bun, and meat. These ingredients themselves require multiple steps to be created. The salad, tomatoes, and meat need to be cut on the cutting board. In addition to that, the cut meat (now called `RawPatty`) needs to be put into a pan. In the default configuration, if the meat is in the pan for too long, it can burn and eventually cause fire, which spreads and destroys other items. This means that the agent has to first complete multiple independent subtasks to be able to combine the ingredients to get the finished burger. For all the experiments in this section, we used the layout shown in Figure A.3. Due to the higher complexity and sparse rewards in the environment, we hypothesize that more in-depth reward shaping is required to cook this recipe.

First, we started with a simple reward setup similar to the final reward setup for tomatoes (Table A.2). Additionally, we added rewards for putting items on a plate, as the agent needs to put different ingredients on a plate to assemble the burger. We also added

punishments for burning an item and starting a fire. For all the experiments regarding the burger, we used the model size configuration from Hafner et al. (2024) with around 50×10^6 parameters.

We found that the agent would often find loopholes for receiving rewards and would not advance any further once these were discovered. For example, we noticed that setting the punishment for throwing items into the trash too low would lead to the agent partially cooking the recipe and then throwing away the intermediate results (see Video D.1.3.1). For instance, in a run with a trashcan punishment of -0.1, the agent learned to repeatedly cut a single item and throw it into the trash as the total reward for cutting this item is greater than 0.1. Another loophole the agent found is to partially cut a tomato and then put it back into the tomato dispenser to reset the cutting progress (see Video D.1.3.2).

Finally, after removing the ability for meat to burn and more in-depth tweaking of the rewards involving putting a higher weight on certain steps required to produce the patty and punishing putting an item on an empty counter, the agent managed to learn to cook and serve the burger. Video D.1.3.7 shows the final agent playing an episode and successfully serving a burger. The exact reward configurations for the individual runs are illustrated in Table A.4. The results of these runs can be seen in Figure 5.2. Table A.3 also contains additional information about each run.



(a) Mean reward

(b) Maximum reward

Figure 5.2: The maximum and mean of each collected training batch for the runs involving burgers. Since the agent received a reward of 1 for serving a burger in all reward configurations, one can easily tell that this only occurred in the last run.

5.2.3 Recipe count

Now that we have experimented with the performance of the model on a single recipe, we then introduced multiple recipes the same agent has to learn in parallel. In this setup, we randomly chose a recipe from a fixed pool of recipes for each episode. Each recipe is given a unique ID, which is passed to the agent as a second observation in the form of a one-hot encoded vector. For each recipe, a different reward configuration is used. In all experiments in this section, we used the model size configuration from Hafner et al. (2024) with approximately 50×10^6 parameters.

We explored multiple setups with different numbers of recipes. First, only tomato soup and salad. Both of these recipes differ in the process required to cook them while also having common steps like cutting a tomato. For episodes where tomato soup was selected as the recipe, we used the reward configuration 'Reward 4' from Table A.2. For the other episodes with salad, we instead used the reward configuration shown in Table A.5. With these configurations, the agent learned to serve both recipes reliably. Figure 5.3 shows the episode score of this run.

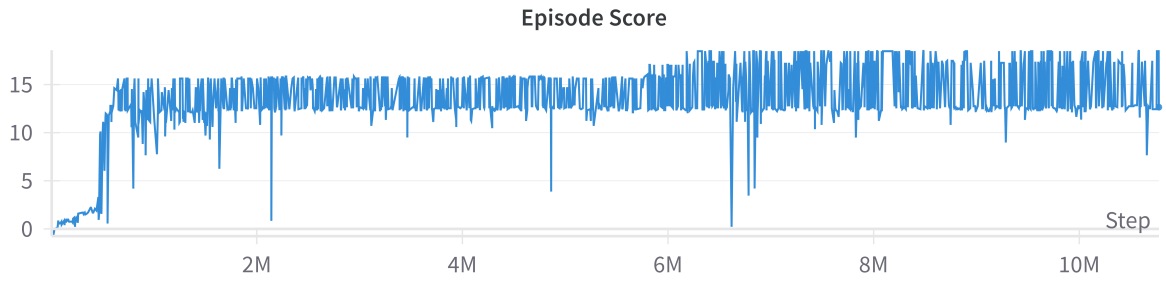


Figure 5.3: Episode score of run with 2 recipes during the training process.

Then we moved on to a setup with four recipes: Tomato Soup, Onion Soup, Salad, and Chips. For the experiments with four recipes, we extended the Hook Callbacks to allow for rewarding more specific events. For example, the agent will now not be rewarded for putting a cut tomato into a pot when the current recipe is onion soup. In the run, using the reward configuration in Table A.6, the agent learned to always cook onion soup and was, therefore, unable to cook the appropriate recipe in $3/4$ of the runs. After modifying the rewards to those in Table A.7, where the agent gets punished for trying to serve the wrong recipe, the agent learned to cook 3 of the 4 recipes. Figure 5.4 shows the

evolution of the reward rate throughout the training for each recipe. Figure A.4 displays the fraction of episodes in which the agent managed to serve the correct recipe in an evaluation setting.

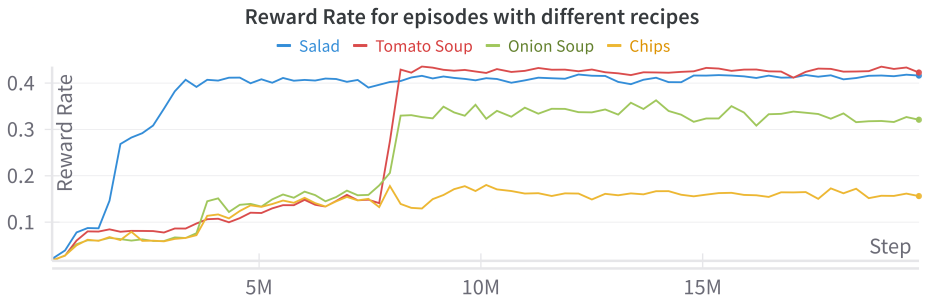


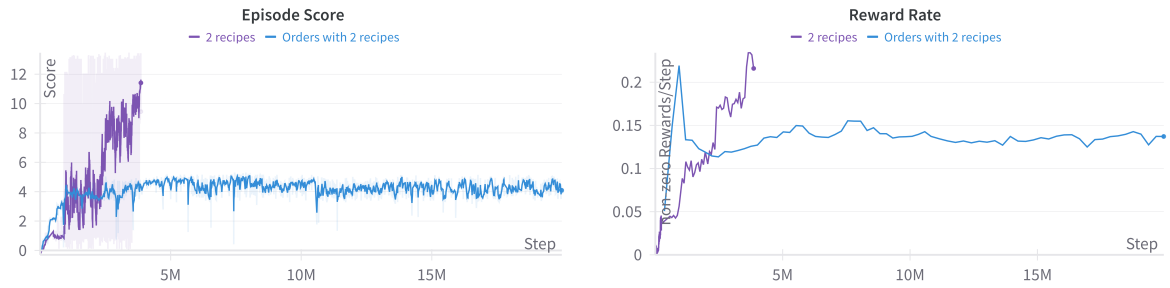
Figure 5.4: Reward rate of episodes with certain recipes during the training when punishing attempting to serve a wrong meal. Note that since we used different reward setups for the different recipes this is not 100% indicative of the actual progress, though this can give a rough tendency.

5.2.4 Orders

Finally, we also introduced the concept of orders. Previously, we considered only a single recipe for each episode. Now, the agent receives orders during the runtime of the environment that have to be completed in a certain amount of time.

We first examined the performance with two recipes in a setting similar to the first setting in Section 5.2.3, with the only difference in the rewards being that `OrderSpecificHookCallbacks` are used to only give rewards for the specific recipe when it is ordered. In this setup, two orders always exist, with each recipe having the same probability of being chosen for every order. Each order o is alive for t_o steps where t_o is sampled from a uniform distribution with the interval $[100, 140)$.

However, with this configuration, the agent did not learn how to cook the appropriate recipe. Looking at the videos (Video D.1.4.3), the agent always learned to cook tomato soup and then serve it when a tomato soup order was available. Figure 5.5 shows the results of this run compared to the run with two recipes described in Section 5.2.3, which indicates the effect where the reward rate is much lower compared to a run with two possible recipes without orders, meaning that rewards occurred rarer.



(a) Episode reward

(b) Reward rate

Figure 5.5: Episode reward and reward rate of a run with orders of two possible recipes compared to a run with two recipes without orders.

5.2.5 Layout complexity

To test the generalization capabilities of DreamerV3, we continued by increasing the complexity of the layout. We used the final reward configuration from Section 5.2.1 for all experiments in this section. First, we introduced multiple layouts from which a random one was chosen each episode. Next, we tested the performance on randomly generated layouts of a fixed size of 5 by 5 and 7 by 7 individually. The method used for generating these layouts is illustrated in Section 4.1.2.1. Since we already saw in Section 5.2.1 that randomly finding an action sequence that leads to a reward is very rare since the amount of possible action sequences grows exponentially, we assume that a layout of 7 by 7 may be too large as this larger size implies longer paths for getting from one counter to another while also increasing the probability of obstacles blocking the way which the agent has to circumnavigate.

Additionally, to test how well the model is able to generalize across different layout sizes, we experimented with using randomly generated layouts with random sizes from 4 by 4 to 7 by 7. In these experiments we evaluate the performance of using layouts with padding compared to layouts without padding. Padding means adding wall blocks around the layout to ensure that all objects have the same scale regardless of the size of the layout. The incentive behind using this padding is to test how well the DreamerV3 model can cope with varying scaling of the objects in the environment and can transfer the knowledge from smaller to bigger scalings. An example of how this padding works is illustrated in Figure 5.6.

(a) Without layout padding (b) With layout padding

Figure 5.6: Example observation in a setup were layouts with the size of 4 by 4 to 7 by 7 can be generated with and without layout padding.

5.2.5.1 Results

Figure 5.7 shows the episode score of the different runs described in Section 5.2.5. In these results we can see that with more complexity, the algorithm takes longer to learn a policy to achieve an optimal reward. In the run with 7 by 7 random layouts, we can see that the agent did not manage to achieve a reward comparable to, e.g., the run with random 5 by 5 layouts. Indeed, if one takes a look at the maximum encountered reward in each episode, we can clearly see that this value never reached 1 in this run, meaning that the agent never managed to successfully serve a tomato soup.

Looking at the runs with both randomly generated layouts and randomly chosen sizes, we can see that the agent did learn to serve tomato soup in some runs. When looking at the results for individual layout sizes, we can see a clear difference between the run with and without layout padding, with the run using this padding having a much higher and more consistent success rate with layouts of a larger size. Figure 5.8 shows the fraction of episodes where a reward of 1 was encountered when following the policy without any exploration with the latest checkpoint of each run. In both cases, we ran this evaluation for $30{,}000$ steps for each size.

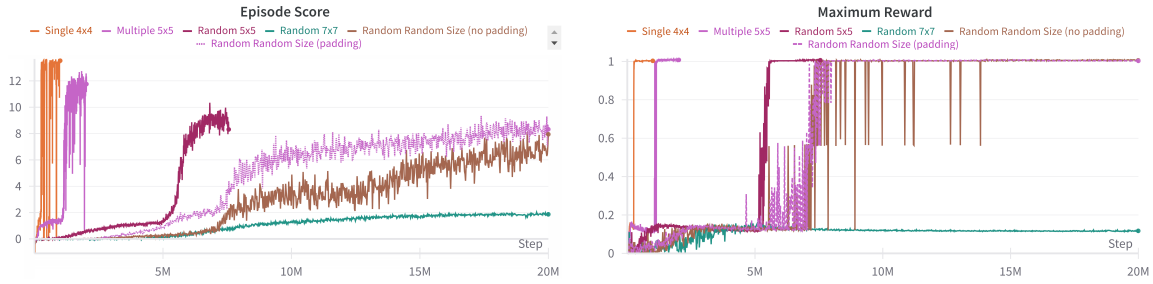


Figure 5.7: Rewards of runs with different layout complexity.

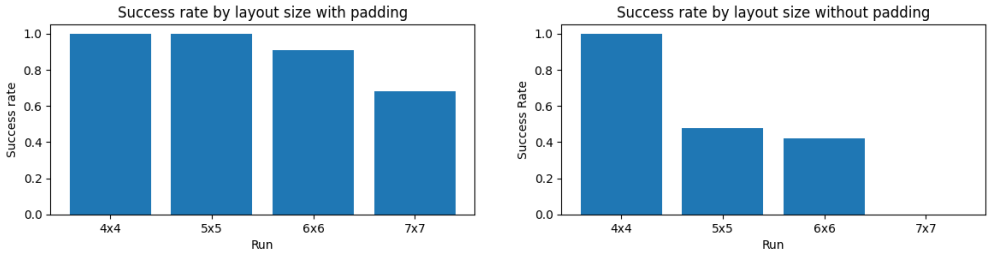
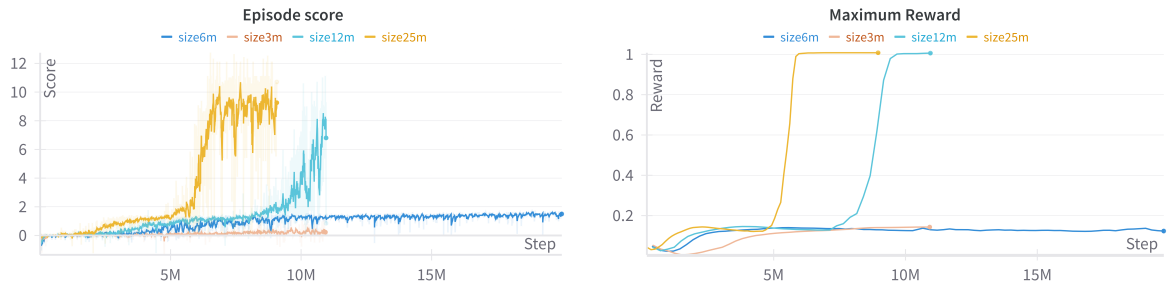


Figure 5.8: Fraction of successful episodes after $30{,}000$ steps in the environment for each size.

5.2.6 Model Complexity

Since more parameters usually imply longer training times and need more resources, another aspect we experimented with is the number of model parameters. Hafner et al. (2024) features multiple model size configurations with approximate counts of parameters. We conducted the experiment described in Section 5.2.5 using randomly generated layouts of size 5 by 5.

Figure 5.9 shows the result of running this experiment for size configurations with approximately 3×10^6, 6×10^6, 12×10^6 and 25×10^6 parameters. Due to hardware and time constraints, we terminated some of these runs early when the agent managed to complete the objective. Again, to check when the agent first managed to serve a tomato soup, we can look at the maximum reward as this action yields a reward of value 1, which is the highest the agent can receive in this configuration. From this maximum reward depicted in Figure 5.9b, we can see that only the runs 'size12', and 'size25', which correspond the parameter counts 12×10^6 and 25×10^6, led to an agent that could serve tomato soup.



(a) Episode score

(b) Maximum Reward

Figure 5.9: Episode score and maximum reward of runs with different model sizes. The runs 'size3m', 'size6m', 'size12m', and 'size25m' were performed using size configurations with around 3×10^6, 6×10^6, 12×10^6, and 25×10^6 parameters respectively.

5.3 FuN

For the FuN algorithm, we started with the same experiments as in Section 5.2 using the same layout (shown in Figure A.1) and the 'Reward 4' configuration from Table A.2. Here, the agent receives an RGB image with a resolution of 84 by 84 at each step. For all experiments, we used the hyperparameters listed in Table B.1, with modifications to these values stated in the individual experiments.

Since this algorithm does not use a replay buffer, it can happen that all transitions collected in one algorithm iteration have a reward of zero, especially at the beginning of the training, when the agent has not learned anything about the environment yet. In our experiments, we observed that a single batch of transitions with all having rewards of value 0 was often followed by the agent not being able to achieve non-zero rewards again. Figure B.1 illustrates a run where this effect occurred. However, this effect can be mitigated by simply collecting more transitions per algorithm iteration. In our experiments, we collected 1000 transitions per agent with 16 agents totaling $16{,}000$ transitions in each iteration. To further decrease the number of transitions with zero rewards, we added a wrapper to the environment, which terminates the episode when the reward of all past 50 steps has been zero. With these modifications, the agent learned to cook and serve tomato soup after around 24×10^6 steps. See Figure B.2 for more information about this run.

5.3.1 Multiple Recipes

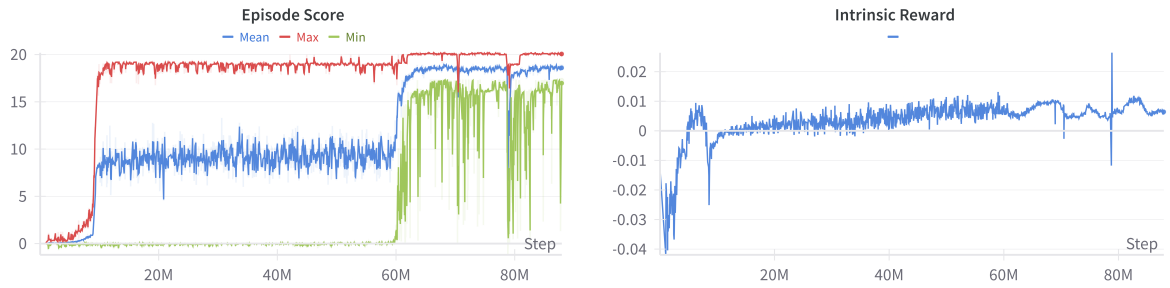
As stated in Section 3.4, we chose to include this algorithm in this thesis is because the hierarchical structure may be beneficial in settings with multiple recipes, as the same sub-policies may be suitable for different recipes. To examine whether this is the case, we used two different methods for dealing with multiple recipes.

5.3.1.1 Multiple Perception Modules

The first method we tried was to introduce a second perceptual module. The outputs of each module are then concatenated to create the new shared latent space of manager and worker. Similar to the experiments with multiple recipes in 5.2, the observation containing the information about the current recipe is a one-hot encoded vector indicating the ID of the current recipe.

First, we used the same setup for two recipes from Section 5.2.3. When automatically terminating the environment, as described in Section 5.3, we found that the agent would only manage to learn one recipe. This may be caused by this early termination, which causes the agent to generate more transitions from episodes with the recipe it has already partially learned, slowing down the learning progress for the other recipe. Figure B.3 shows a run where this effect occurs. However, after setting the episodes to the fixed length of 300 again, the agent did manage to serve the appropriate recipe reliably. The episode score and intrinsic reward are both shown in Figure 5.10. The minimum, mean, and maximum of the episode score are close, indicating that the agent has learned a policy that can cause rewards in episodes with either recipe. The intrinsic reward (defined in Equation 3.6) is also above zero in the end.

When performing a Principal Component Analysis (PCA) (as described in, e.g., Jolliffe and Cadima (2016)) on the goal vectors produced by the manager to reduce the goal space to 2 dimensions, we get the result pictured in Figure 5.11a. In this plot, one can clearly see a separation of the goal vectors from episodes with salad and from episodes with tomato soup with there still being some overlap.



(a) Episode score

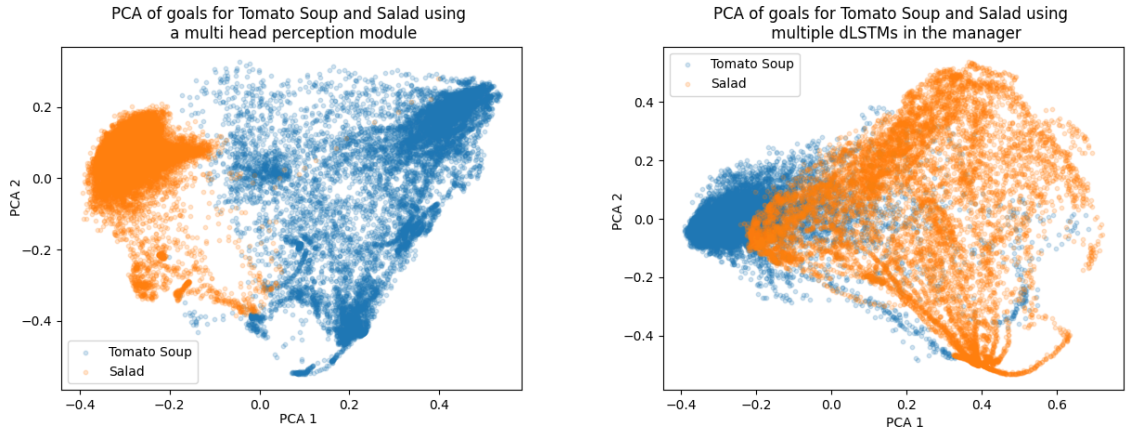
(b) Intrinsic Reward

Figure 5.10: Episode score and intrinsic reward of a run with 2 possible recipes where the episodes are always 300 steps long. The method here used is the one with two perception modules, described in Section 5.3.1.1.

5.3.1.2 Multiple dLSTMs

The other approach we attempted is to use multiple dLSTMs in the manager for each recipe, similar to the Mixture of Experts method (as described in Chen et al., 2022). However, instead of using a separate network for predicting the weighting of the individual dLSTMs, the single appropriate dLSTM is chosen depending on the current recipe. In this approach, the worker does not receive any information about the current recipe. Since the worker is trained using an actor-critic method, we also use multiple critic networks for each recipe for both the manager and the worker. This is necessary since we use different reward configuration depending on the recipe. For both dLSTMs we use the same f^{Mspace} for both managers as the goal spaces for all recipes need to be compatible with each other. The idea behind this approach is to further enforce the learning of sub-policies that can be reused for multiple recipes.

Figure 5.12 shows the result of running an experiment with a manager using multiple dLSTMs depending on the recipe. Here, we can see that the difference between the mean, maximum, and minimum reward is significant, with the minimal reward being near zero. Looking at the intrinsic reward pictured in Figure 5.12b, we can see that this value was above zero in the beginning, but during the training, it went down to about -0.08. When performing a PCA on the goal vectors produced by the manager, like in Section 5.3.1.1, we get the result depicted in Figure 5.11b. Here, the goal vectors generated by episodes with the different recipes do differ; however, the separation into clusters is less apparent.

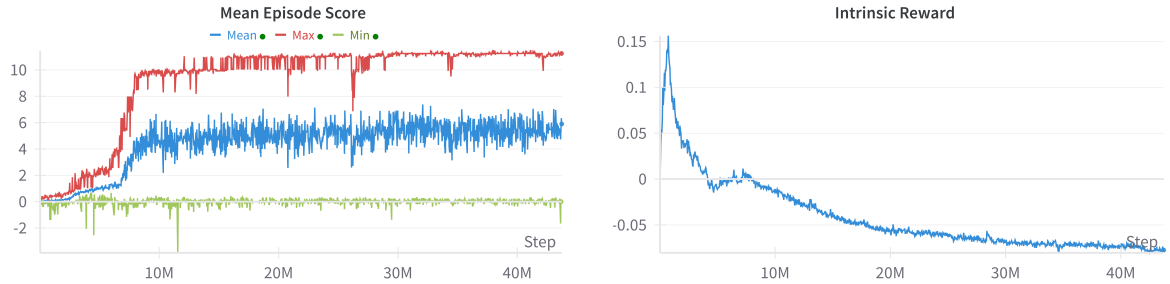


(a) Approach with multiple perception modules (Section 5.3.1.1)

(b) Approach with multiple dLSTMs inside the manager (Section 5.3.1.2)

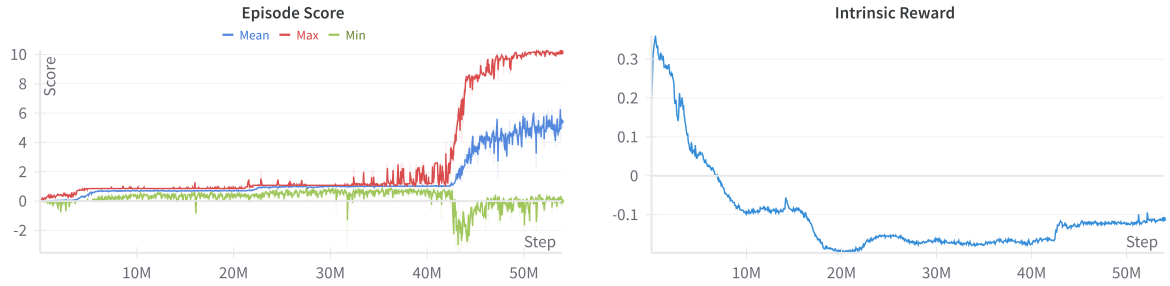
Figure 5.11: Result of performing a PCA on 20000 collected goal vectors with an equal amount of vectors coming from episodes with tomato soup and episodes with salad to reduce the amount of dimensions to 2. We also performed an Independent Component Analysis (ICA) with very similar results. The results of this can be seen in Figure B.4.

Recall that the intrinsic reward is weighted using the hyperparameter $\alpha \in [0, 1]$ (Vezhnevets et al., 2017). Hence, it may be sensible to increase this weight to increase the intrinsic reward. In the experiments in Vezhnevets et al. (2017), a high value for α also led to better results for some (not all) environments. Figure 5.13 showcases a run with $\alpha = 1$. Here, we can see that the intrinsic reward is high initially at around 0.35 but then decreases during the training to around -0.11.



(a) Episode score (b) Intrinsic Reward

Figure 5.12: Episode score and intrinsic reward of a run with 2 possible recipes where the episodes are always 300 steps long. The method here used is the one with two dLSTMs inside the manager, described in Section 5.3.1.2.



(a) Episode score (b) Intrinsic Reward

Figure 5.13: Episode score and intrinsic reward of a run with 2 dLSTMs inside the manager (Section 5.3.1.2) for each recipe with $\alpha = 1$, 2 possible recipes where the episodes are always 300 steps long.

6 Discussion

This chapter discusses and critically evaluates the results and experimental setup. Section 6.1 discusses the limitations of the experimental setup used. Section 6.2 and Section 6.3 discuss the results of the experiments involving the DreamerV3 and FuN model, respectively. Finally, Section 6.4 compares the results and capabilities of both models.

6.1 Limitations of the experimental setup

The hardware and time limitations of this thesis are reflected in the setup of the experiments. We often terminated runs early if the agent did not seem to make any progress for a long period of time. Since this progress is often tied to the agent randomly discovering a specific action sequence, it, e.g., might have happened that the agent in the runs depicted in Figure A.2 performed an action sequence with which it successfully cooks tomato soup and serves it.

Also, in our setup, we only ran each configuration once. Due to the same factor of the agent having to discover particular action sequences by random chance to advance in the completion of the targeted task, the exact point at which these advancements are made can vary from run to run.

For all the experiments, we also solely used RGB images as observations even though we theoretically have access to the environment's 'true state'. Vectorized representations can be beneficial, especially for model-based methods like DreamerV3, as the agent is limited by the performance of the world model.

Additionally, we did not perform any hyperparameter tuning except in Section 5.3.1.2. While DreamerV3 was designed to perform well across various domains with fixed

parameters, with FuN, this is not necessarily the case, and performing hyperparameter optimization could have been advantageous in this case. However, due to hardware and time constraints, we decided to stick to the default values for the hyperparameters.

Going back to the runs with multiple recipes, one major limitation of the representation used is that we only consider a finite set of fixed recipes. However, CCE allows for customizing these recipes using configuration files. If a user updates the recipes, the agent would need to be trained again, potentially using another reward configuration.

6.2 DreamerV3

Section 5.2.1 has shown us that solely rewarding the final step of cooking and serving a recipe is insufficient for an agent to complete this task. This means that it is necessary to include rewards for intermediate steps. However, these rewards usually need to be designed specifically for each recipe as the steps required to cook these can vary greatly, depending on the recipe configurations used. When designing these rewards, though, we need to ensure that completing the desired task is the only way to maximize the return of the episode and that the agent does not get stuck in local maxima. For example, with run configuration 'Reward 2' where trashcan usage is not punished, the agent learned to repeatedly cut tomatoes and throw them into the trashcan even though the episode score for cooking and serving tomato soups would be higher. This can be seen in Figure 5.1, where the reward configuration 'Reward 4' yields higher episode scores even though this configuration solely has additional punishments compared to 'Reward 2'.

When moving to a more complex recipe, we found that we needed to perform even more in-depth reward shaping. Nevertheless, one problem with performing in-depth reward shaping is that by introducing many rewards for intermediate actions, especially when rewarding actions in a specific order, we also essentially predetermine a specific action sequence, making reinforcement learning redundant.

When introducing multiple recipes, the necessity for more in-depth reward shaping becomes even stronger. While in runs with only two recipes, more general reward configurations sufficed, we had to additionally tweak the rewards to see partial success in runs with four different recipes where the agent learned to serve three of the four recipes in the appropriate setting. With additional reward shaping, this may be improved,

but due to limited time constraints, we were not able to investigate this further. Adding the concept of orders, though, introduces another form of challenge. Here, we have the problem that the agent can still achieve a reward by cooking a recipe that is currently not ordered and waiting until it is ordered, as seen in Section 5.2.4. One limitation of the approach used here is that in the modeling of the representation used for the orders, the agent has no indication of how long an order will stay active. Also, this representation does not consider the number of the orders with a certain recipe.

Regarding the experiments involving increasing the layout complexity, we saw that the agent was able to generalize across randomly generated layouts of size 5 by 5. When increasing the layout size, we saw that the agent was not able to learn to serve tomato soup in a run with only layouts of size 7 by 7. However, when using a random size for each episode, the agent did manage to complete the task in about 69% of the episodes with a layout size of 7 by 7 when using padding to ensure that the cells always have the same size. This indicates that the agent could transfer the 'knowledge' gained from smaller layouts to larger layouts. Without this padding, we saw that the agent could not develop the same generalization capabilities, as the performance drastically declined when increasing the layout size. This effect may be caused by the non-scale-invariant nature of Convolutional Networks, which means that the autoencoder of the world model needs to utilize more filters to classify the same objects with multiple scalings (Xu et al., 2014). On the other hand, one problem with using this padding is that in settings where we can have very large layouts, this will also lead to a tiny scaling of the smaller layouts.

When decreasing the model size, we saw that the configuration with around 12×10^6 parameters allowed the agent to reliably cook tomato soup while failing with the configuration with around 6×10^6 parameters. On one hand, this shows that, at least for some setups, we can use a smaller parameter count, reducing the required resources for training the model. On the other hand, we also need to be cautious when reducing the parameter count, as the performance can quickly degrade.

6.3 FuN

The second algorithm we evaluated is FuN. With the same reward configuration that resulted from the reward shaping in the DreamerV3 experiments, the FuN agent could also complete the task with the same simple recipe.

In setups with multiple recipes, we explored two different methods: using a second perception module and using separate dLSTMs inside the manager for each recipe. With the method involving multiple dLSTMs, we saw that the intrinsic reward of the worker ended up being smaller than zero, indicating that the worker is not able to move the latent state of the manager into the direction given by the goals. Since, in this setup, the manager is the only part of the actor with access to the ID of the current recipe, the worker cannot cook the appropriate recipe on its own. In the PCA of the of the manager's goal space, it is possible to observe a difference between the goal vectors predicted by the separate dLSTMs. This means that the goals given to the worker did differ depending on the recipe. Considering that the goal vectors for the different recipes were predicted by independently trained networks, this is not surprising. However, since the worker did not manage to follow these goals reliably, this does not seem to have any meaning in this context. Thus, this model was not able to efficiently utilize the hierarchical structure of the FuN architecture.

One flaw in this method we did not consider is that each dLSTM only gets trained on about half of the transitions as the other parts of the model. When the number of recipes increases, this problem is amplified even further.

With the other method using multiple perception methods, we saw that the agent could reliably cook the appropriate recipe. Looking at the PCA of the goal vectors of this method, we can observe a clear structure with separate clusters for each recipe that, however, still have some overlap. Since we also see that in contrast to the approach using multiple dLSTMs, the intrinsic reward is larger than zero, this indicates that the model has learned multiple sub-policies, with most of them being specific to one recipe but some being reused for multiple recipes. This suggests that the hierarchical architecture of FuN might be advantageous in this case compared to end-to-end trained models. However, when increasing the number of recipes to 4, we saw that the agent did not learn to cook the different recipes within the given time budget. Increasing the complexity of the

model by adding more layers and training the model for more steps could be a strategy to explore further.

6.4 Comparison

Looking at the runs on the simple layout (Figure A.1) with the task of serving tomato soup, we see that the DreamerV3 Agent managed to complete the task after only about $250{,}000$ steps, while FuN took almost $25{,}000{,}000$ to achieve the same performance. However, when looking at the parameter count, we see that FuN is a much more lightweight model, having only around 1.4×10^6 parameters compared to the around 60×10^6 parameters used for the DreamerV3 model for the same experiment. This means that also the training time is much faster. While the run pictured in Figure B.2 took around 3 days and 17 hours on an NVIDIA GTX 1080 ti to reach 95×10^6 steps (around 296 steps per second), the run pictured in Figure 5.1 with the same setup but using the DreamerV3 algorithm took 20 hours and 30 minutes on an NVIDIA TESLA P100 with 16 GB VRAM to reach 10^6 steps (around 14 steps per second) which means that the training time per step was around 21 times as fast for the FuN model compared to the DreamerV3 model. Even considering that the actual implementation can significantly influence the runtime, we see that FuN is much faster. When looking at the runs with multiple recipes, we could see that DreamerV3 handled more possible recipes than FuN, which may be linked to the higher parameter count of about 50×10^6 used for this run.

7 Conclusion and Outlook

In this thesis, we have evaluated two different reinforcement learning algorithms on the CCE while incrementally increasing the complexity of the environment along different dimensions. Regarding question 1, we found that reward shaping is required even for simple layouts with relatively simple recipes such as tomato soup, as cooking and serving a recipe usually requires too many steps to be performed solely by random exploration. In order to achieve an agent that could cook and serve tomato soup in a simple layout, we had to introduce additional rewards for intermediate steps such as cutting a tomato or putting the tomato soup on a plate. However, when introducing these rewards, we also encountered issues with the agent finding 'loopholes' that allowed them to achieve a high reward without completing the desired task. To perform this reward shaping, we had to modify the environment by adding new hook callbacks. Additionally, we used a more simplistic implementation for the movement system, which, however, due to the time resolution of the agent, is equivalent to the original movement system. However, we did not need to make any further modifications to the dynamics of the environment to achieve a success with a simple layout.

Regarding question 2, we increased to complexity along different dimensions. First, when increasing the complexity of the recipe by switching to burgers, we found that more in-depth reward shaping is required. However, reward shaping requires intensive domain knowledge, which may be a problem for users who would like to use custom recipes. One concept that could be explored in the future is a method for automatically generating reward configurations given a recipe structure. Additionally, to decrease the necessary amount of intermediate rewards, a possible strategy to employ in the future could be to use an Intrinsic Curiosity Module (ICM), as proposed by Pathak et al. (2017), which encourages the exploration of new, unseen states.

Introducing multiple recipes while working for a recipe count of 2 required more tweaking when increasing the count to 4. Even with this tweaking, we were not able to get the

agent to serve more than 3 of the 4 recipes reliably. In future work, one could try more in-depth reward shaping to only reward steps required for the current recipe. When further introducing a time dependency with the concept of orders, we did not manage to train an agent that completes the current orders. Using an alternative representation for the orders could help with this problem in the future.

When increasing the complexity of the layout by using randomly generated layouts of random sizes, we found that the DreamerV3 model could transfer knowledge across different layout sizes as long as we utilized measures to ensure that the scaling of the objects in the observation is the equal for all layout sizes. When using large layouts, this can be problematic as the scaling of the objects would get very small, even for small layouts. In future work, we could explore the possibility of using a vectorized representation of the environment instead of the RGB images to combat this effect.

Regarding the FuN model, we found that it was capable of training an agent that could serve tomato soup without any modifications using the layout and reward configuration resulting from the reward shaping for the DreamerV3 agent. In experiments with multiple recipes, we examined two different methods. Only the method involving multiple perception modules yielded successful results. Coming back to question 3 of this thesis, when analyzing the goal space of the manager, we also observed hints that the same sub-policies were reused for both recipes, meaning that the hierarchical structure of FuN was advantageous in this case. However, when increasing the number of recipes to 4, the agent did not learn to cook and serve more than one. Since the FuN model has very few parameters compared to the DreamerV3 model, one approach we could consider in the future is to increase the complexity of the FuN model by introducing more layers in the individual modules.

Generally, in future work, it may also be interesting to try more models like, e.g., DIAMOND (Alonso et al., 2024), which uses a similar approach as DreamerV3 but instead uses a diffusion model as a decoder which, according to Alonso et al. (2024), allows for more flexible world models. In addition to single-agent reinforcement learning, we could apply multi-agent reinforcement learning to CCE, as we only briefly touched on this subject in this thesis.

Abbreviations

CCE Cooperative Cuisine Environment

CTDE Centralized training and decentralized execution

dLSTM dilated LSTM

FuN Feudal Network

ICA Independent Component Analysis

ICM Intrinsic Curiosity Module

MARL Multi-Agent Reinforcement Learning

MDP Markov Decision Process

PCA Principal Component Analysis

POMDP Partially Observable Markov Decision Process

PPO Proximal Policy Optimization

SGD Stochastic Gradient Descent

Glossary

entropy scale Parameter which defines the scale of the entropy of the predicted action distribution in the objective function. 10

episode Sequence of steps from start to termination in an environment. v, 10, 20–30, 32, 33, 45, 48, 50, 52, 55, 56

episode reward The same as episode score. v, 23

episode score The sum of all rewards in an episode. v, 18, 19, 21, 24, 26–28, 30, 32, 46, 55, 59

experiment Investigation which can involve multiple runs in order to test a hypothesis, evaluate an algorithm, or compare different configurations under specific conditions . v, vii, viii, 1, 2, 5, 13, 15–21, 23, 25–29, 31, 33–35, 37, 45–48, 50, 53

policy Probability distribution of the action to take given the current state (MDP) or observation (POMDP) (Ding, Huang, et al., 2020). 8

reward rate Fraction of steps with a non-zero reward in an episode. This is a metric produced by the implementation of the DreamerV3 algorithm[1] . v, 22, 23

run Single instance of training an agent in an environment. v, vi, 14, 17–33, 35, 46, 48, 49, 52, 54, 55, 60

[1] https://github.com/danijar/dreamerv3

Bibliography

Alonso, E., Jelley, A., Micheli, V., Kanervisto, A., Storkey, A., Pearce, T., & Fleuret, F. (2024). Diffusion for world modeling: Visual details matter in atari. https://arxiv.org/abs/2405.12399

Bettini, M., Prorok, A., & Moens, V. (2024). Benchmarl: Benchmarking multi-agent reinforcement learning. *Journal of Machine Learning Research*, *25*(217), 1–10. http://jmlr.org/papers/v25/23-1612.html

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

Carroll, M., Shah, R., Ho, M. K., Griffiths, T. L., Seshia, S. A., Abbeel, P., & Dragan, A. (2020). On the utility of learning about humans for human-ai coordination. https://arxiv.org/abs/1910.05789

Chen, Z., Deng, Y., Wu, Y., Gu, Q., & Li, Y. (2022). Towards understanding the mixture-of-experts layer in deep learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in neural information processing systems* (pp. 23049–23062, Vol. 35). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2022/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf

de Witt, C. S., Gupta, T., Makoviichuk, D., Makoviychuk, V., Torr, P. H. S., Sun, M., & Whiteson, S. (2020). Is independent learning all you need in the starcraft multi-agent challenge? https://arxiv.org/abs/2011.09533

Ding, Z., & Dong, H. (2020). Challenges of reinforcement learning. In H. Dong, Z. Ding, & S. Zhang (Eds.), *Deep reinforcement learning: Fundamentals, research and applications* (pp. 257–259). Springer Singapore. https://doi.org/10.1007/978-981-15-4095-0_7

Ding, Z., Huang, Y., Yuan, H., & Dong, H. (2020). Introduction to reinforcement learning. In H. Dong, Z. Ding, & S. Zhang (Eds.), *Deep reinforcement learning: Fundamentals,*

research and applications (pp. 47–123). Springer Singapore. https://doi.org/10.1007/978-981-15-4095-0_2

Ghost Town Games Ltd. (2016). *Overcooked!* https://www.team17.com/games/overcooked/

Hafner, D., Lillicrap, T., Ba, J., & Norouzi, M. (2020). Dream to control: Learning behaviors by latent imagination. https://arxiv.org/abs/1912.01603

Hafner, D., Lillicrap, T., Norouzi, M., & Ba, J. (2022). Mastering atari with discrete world models. https://arxiv.org/abs/2010.02193

Hafner, D., Pasukonis, J., Ba, J., & Lillicrap, T. (2024). Mastering diverse domains through world models. https://arxiv.org/abs/2301.04104

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*, 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735

Jolliffe, I. T., & Cadima, J. (2016). Principal component analysis: A review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *374*(2065), 20150202. https://doi.org/10.1098/rsta.2015.0202

Kingma, D. P., Salimans, T., & Welling, M. (2016). Improving variational inference with inverse autoregressive flow. *CoRR*, *abs/1606.04934*. http://arxiv.org/abs/1606.04934

Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. In S. Solla, T. Leen, & K. Müller (Eds.), *Advances in neural information processing systems* (Vol. 12). MIT Press. https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

Laurent, G. J., Matignon, L., & Fort-Piat, N. L. (2011). The world of independent learners is not markovian. *Int. J. Knowl. Based Intell. Eng. Syst.*, *15*, 55–64. https://api.semanticscholar.org/CorpusID:220863181

Li, Y. (2018). Deep reinforcement learning. *CoRR*, *abs/1810.06339*. http://arxiv.org/abs/1810.06339

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, … Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, *abs/1602.01783*. http://arxiv.org/abs/1602.01783

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Pateria, S., Subagdja, B., Tan, A.-h., & Quek, C. (2021). Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.*, *54*(5). https://doi.org/10.1145/3453160

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. https://arxiv.org/abs/1705.05363

Poupart, P. (2017). Partially observable markov decision processes. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of machine learning and data mining* (pp. 959–966). Springer US. https://doi.org/10.1007/978-1-4899-7687-1_629

Rashid, T., Samvelyan, M., de Witt, C. S., Farquhar, G., Foerster, J., & Whiteson, S. (2018). Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. https://arxiv.org/abs/1803.11485

Schröder, F., & Heinrich, F. (2024). Cooperative cuisine environment. https://gitlab.ub.uni-bielefeld.de/scs/cocosy/cooperative-cuisine

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, *abs/1707.06347*. http://arxiv.org/abs/1707.06347

Shakya, A. K., Pillai, G., & Chakrabarty, S. (2023). Reinforcement learning algorithms: A brief survey. *Expert Systems with Applications*, *231*, 120495. https://doi.org/https://doi.org/10.1016/j.eswa.2023.120495

Sunehag, P., Lever, G., Gruslys, A., Czarnecki, W. M., Zambaldi, V., Jaderberg, M., Lanctot, M., Sonnerat, N., Leibo, J. Z., Tuyls, K., & Graepel, T. (2017). Value-decomposition networks for cooperative multi-agent learning. https://arxiv.org/abs/1706.05296

Sutton, R. S., Precup, D., & Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1), 181–211. https://doi.org/https://doi.org/10.1016/S0004-3702(99)00052-1

Tan, M. (1997). Multi-agent reinforcement learning: Independent versus cooperative agents. *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:273638739

Uther, W. (2017). Markov decision processes. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of machine learning and data mining* (pp. 793–798). Springer US. https://doi.org/10.1007/978-1-4899-7687-1_512

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., & Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. https://arxiv.org/abs/1703.01161

Wang, R. E., Wu, S. A., Evans, J. A., Tenenbaum, J. B., Parkes, D. C., & Kleiman-Weiner, M. (2020). Too many cooks: Bayesian inference for coordinating multi-agent collaboration. https://arxiv.org/abs/2003.11778

Weitkamp, L. (2020). Feudal networks for hierarchical reinforcement learning. https://github.com/lweitkamp/feudalnets-pytorch

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3), 229–256. https://doi.org/10.1007/BF00992696

Xu, Y., Xiao, T., Zhang, J., Yang, K., & Zhang, Z. (2014). Scale-invariant convolutional neural networks. https://arxiv.org/abs/1411.6369

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., & Wu, Y. (2022). The surprising effectiveness of ppo in cooperative, multi-agent games. https://arxiv.org/abs/2103.01955

Zhang, H., Huang, R., & Zhang, S. (2020). Integrating learning and planning. In H. Dong, Z. Ding, & S. Zhang (Eds.), *Deep reinforcement learning: Fundamentals, research and applications* (pp. 307–316). Springer Singapore. https://doi.org/10.1007/978-981-15-4095-0_9

Zhang, J., Kim, J., O'Donoghue, B., & Boyd, S. P. (2020). Sample efficient reinforcement learning with REINFORCE. *CoRR*, *abs/2010.11364*. https://arxiv.org/abs/2010.11364

Appendices

A DreamerV3 Experiments

A.1 Basic Reward Shaping

A.1.1 Minimum number of steps required to serve a tomato soup

This section examines the minimum number of steps required to serve a tomato soup when using the layout pictured in Figure A.1.

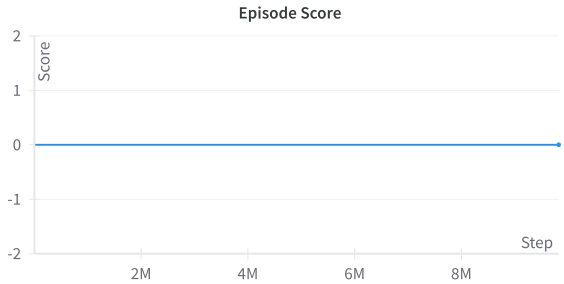
At the beginning of the episode, the agent always starts at the top right of the empty cells. From there, it takes one step to get to the tomato. Then the agent has to take the tomato, move down, put it on the cutting board, cut it, take it, turn to the pot, and put the tomato into it. This has to be repeated two times as the tomato soup requires three `ChoppedTomatos` in the configuration used. Then the agent has to move right, take a plate, go left to the pot, put the tomato soup into it, move to the counter for serving the meal, turn to the counter, and serve it.

In total, this adds up to 33 actions.

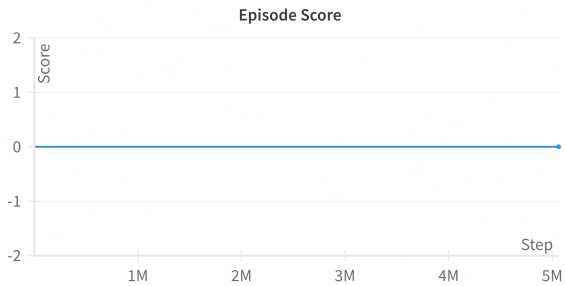
Table A.1: Model size configuration for the basic reward shaping experiments. For more information about the individual parameters see Hafner et al. (2024).

Parameter	Value
Hidden size	384
Recurrent units	6144
Base CNN channels	48
Codes per latent	48

Figure A.1: This is a simple 4 by 4 layout used for some single-agent experiments. This layout contains both tomatoes and lettuce, even though it was mostly used for single-recipe experiments with tomato soup as the target recipe. The incentive behind this is that when providing a reward for cutting an item this leaves more room for the reinforcement learning, as the agent still has to learn that one of those items is not necessary for the recipe.



(a) Regular training pipeline



(b) Training pipeline where the agent only explores until a reward was achieved

Figure A.2: Episode score of runs where only serving tomato soups is rewarded (Reward 1 in Table A.2).

Table A.2: The different reward configurations, we used for the reward shaping experiments involving a simple layout and tomato soup. These experiments are described in more detail in Section 5.2.1.

Event	Reward 1	Reward 2	Reward 3	Reward 4
Serve completed Meal	1	1.0	1.0	1.0
Put tomato on cutting board	0	0.1	0.1	0.1
Cut tomato (each step)	0	0.01	0.01	0.01
Finish cutting tomato	0	0.1	0.1	0.1
Put cut tomato on pot	0	0.1	0.1	0.1
Remove a cut tomato from the pot	0	0	0	−0.1
Put tomato soup on plate	0	0.1	0.1	0.1
Movement	0	0	0	0
Interaction with arbitrary counter	0	0	0	0
Put an item into the trash can	0	0	−1.0	−1.0

Figure A.3: Layout used for the experiments involving burgers

A.2 Recipe Complexity Experiments

Table A.3: Information about the individual runs. Due to time and hardware constraints, we terminated some runs early when we found that the agent got stuck in a loophole or when it learned to cook the burger successfully. 'Success' means that the agent has learned a policy with which it can reliably cook and serve burgers. Note that we used a higher episode length for the first three runs. As the agent has to complete multiple parallel subtasks and combine the results in order to assemble a burger, this can increase the chance of the agent completing multiple subtasks in the same episode during exploration. However, due to hardware and time constraints, we could not investigate further whether this actually is beneficial.

Run	Main change	Meat can burn	Episode length	Run duration in steps	Success
1	Initial configuration	Yes	30000	19.6×10^6	No
2	Higher trashcan punishment	Yes	30000	9.3×10^6	No
3	Do not reward cutting each tick	Yes	30000	8.6×10^6	No
4	Even higher trashcan punishment	Yes	300	9.6×10^6	No
5	Meat cannot burn anymore	No	300	9.7×10^6	No
6	More reward for putting RawPatty on a Pan	No	300	20×10^6	No
7	Punish Putting items on empty counters	No	300	9.6×10^6	Yes

Table A.4: These are the reward configurations that we used for runs involving burgers. Note that in runs 5, 6, and 7 meat cannot burn anymore and, therefore, cannot start fire, which means that the punishment for 'A fire starts' is irrelevant for these runs.

Event	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
Finish combining burger	0.3	0.3	0.3	0.3	0.3	0.3	0.3
Cut an item (each tick)	0.01	0.01	0	0	0	0	0
Finish cutting an item	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Item burned	-1	-1	-1	-1	-1	-1	-1
Trashcan usage	-0.1	-0.15	-0.15	-0.4	-0.4	-0.4	-0.4
Put an item on any cooking equipment	0.1	0.1	0.1	0.1	0	0	0
Put a `RawPatty` on a Pan	0	0	0	0	0.1	0.4	0.4
Put an item on a plate	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Put an item on an empty counter	0	0	0	0	0	0	-0.1
Pick up an item from cooking equipment	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1
A fire starts	-1	-1	-1	-1	-1	-1	-1
Serve finished meal	1	1	1	1	1	1	1

A.3 Recipe Count experiments

Table A.5: The reward configuration we used for the experiments where the agent was trained using two recipes in episodes where salad was selected as the target recipe. Note that due to how the environment was configured, it was not possible for the agent to put an item other than a cut tomato or cut salad on a plate.

Event	Reward
Serve a salad	1.0
Use the trashcan	-0.15
Use the cutting board (each tick)	0.01
Finish cutting an item	0.1
Put an item on a plate	0.1
Remove an item from cooking equipment	-0.1

Table A.6: Initial reward configuration for experiment with 4 recipes.

Event	Tomato Soup	Onion Soup	Salad	Chips
Serve the correct meal	1.0	1.0	1.0	1.0
Try to serve the wrong meal	0	0	0	0
Use the trashcan	−0.15	−0.15	−0.15	−0.15
Use the cutting board (each tick)	0.01	0.01	0.01	0.01
Remove an item from cooking equipment	−0.1	−0.1	−0.1	−0.1
Finish cutting an item	0.1	0.1	0.1	0.1
Put `RawChips` in a Basket	0	0	0	0.1
Put `Chips` on a Plate	0	0	0	0.1
Put `ChoppedOnion` into a Pot	0	0.1	0	0
Put `OnionSoup` on a Plate	0	0.1	0	0
Put `ChoppedTomato` or `ChoppedLettuce` on a Plate	0	0	0.1	0
Put `ChoppedTomato` into a Pot	0.1	0	0	0
Put `TomatoSoup` on a Plate	0.1	0	0	0

Table A.7: Reward configuration for experiment with 4 recipes where trying to serve the wrong recipe is punished.

Event	Tomato Soup	Onion Soup	Salad	Chips
Serve the correct meal	1.0	1.0	1.0	1.0
Try to serve the wrong meal	−0.1	−0.1	−0.1	−0.1
Use the trashcan	−0.15	−0.15	−0.15	−0.15
Use the cutting board (each tick)	0.01	0.01	0.01	0.01
Remove an item from cooking equipment	−0.1	−0.1	−0.1	−0.1
Finish cutting an item	0.1	0.1	0.1	0.1
Put `RawChips` in a Basket	0	0	0	0.1
Put `Chips` on a Plate	0	0	0	0.1
Put `ChoppedOnion` into a Pot	0	0.1	0	0
Put `OnionSoup` on a Plate	0	0.1	0	0
Put `ChoppedTomato` or `ChoppedLettuce` on a Plate	0	0	0.1	0
Put `ChoppedTomato` into a Pot	0.1	0	0	0
Put `TomatoSoup` on a Plate	0.1	0	0	0

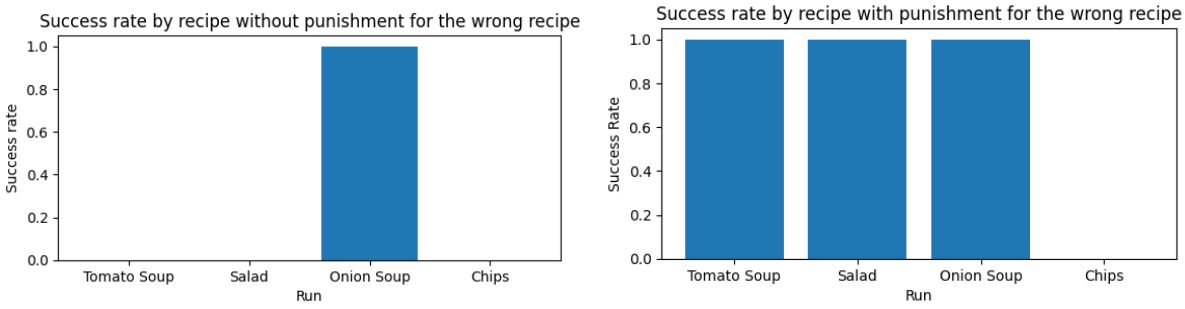
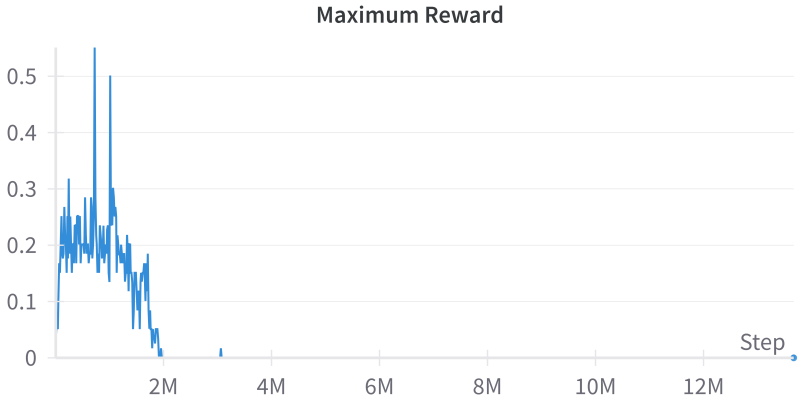


Figure A.4: Fraction of successful episodes after 30000 steps for both reward configurations using the last checkpoint from each runs.

B FuN Experiments

Table B.1: Base hyperparameters used for the experiments. If any hyperparameters were changed for an experiment, this is stated in the description of the experiment.

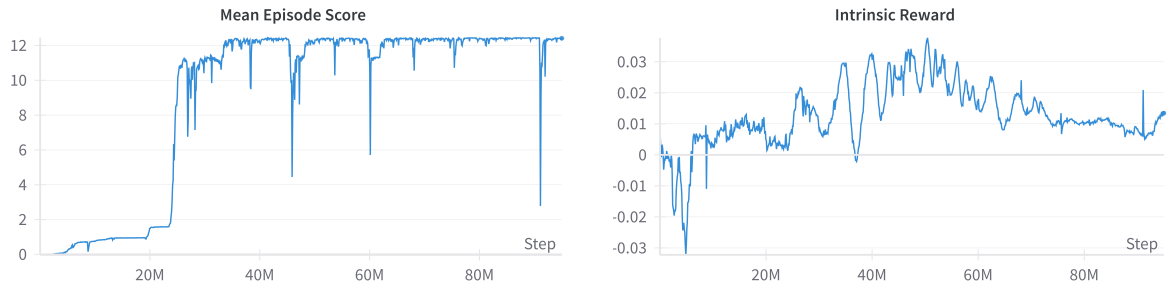
Hyperparameter	Value	Description
α	0.5	Weighting factor of the intrinsic reward
ϵ	0.01	Random Gaussian goal for exploration
r	10	Dilation radius
c	10	Time horizon
d	256	Hidden dimension of the manager
k	16	Hidden dimension of the worker
γ_w	0.999	Discount factor of the worker
γ_m	0.99	Discount factor of the manager



(a) Maximum reward

(b) The layout used for this run

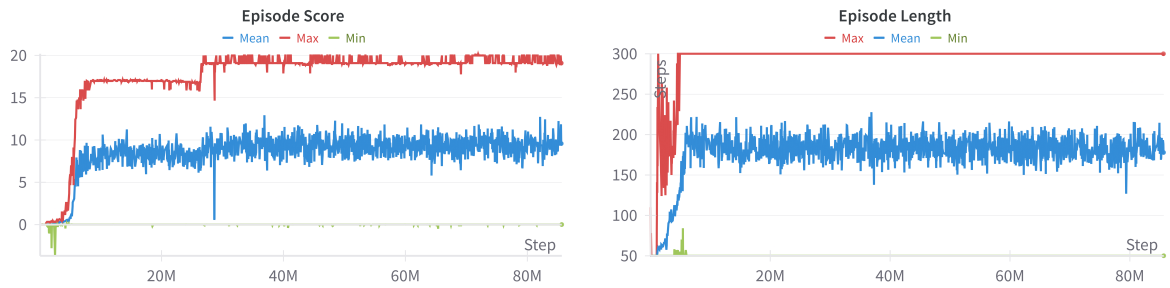
Figure B.1: Maximum reward of a run where only 300 steps per agent per iteration were collected. After around step 3.1×10^6, the agent did not achieve a reward higher than 0 until it was stopped after around 13.6×10^6 steps. Figure B.1b also shows the layout that we used for this run.



(a) Mean episode score

(b) Intrinsic Reward of the Worker

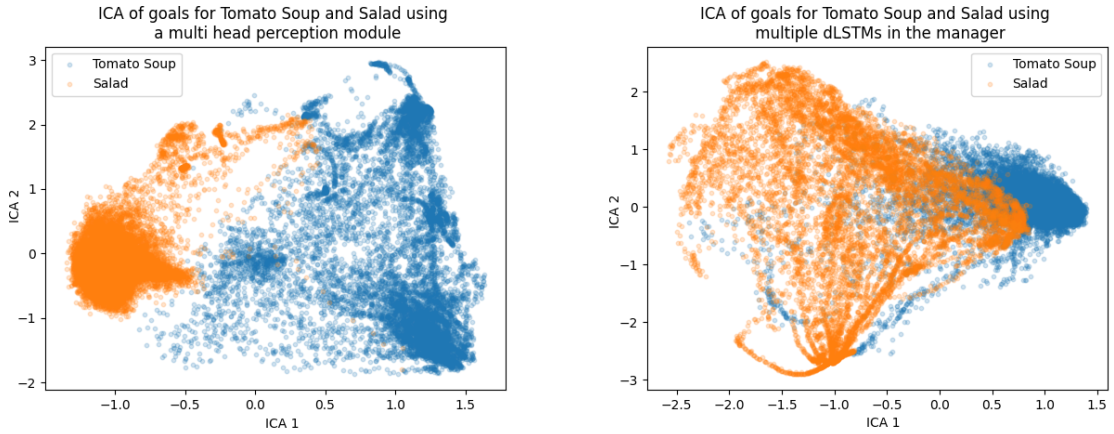
Figure B.2: Successful run with only tomato soup on a simple layout.



(a) Episode score

(b) Episode Length

Figure B.3: Episode score and length of a run with two possible recipes where the episodes are automatically terminated if all past 50 rewards have been zero. Here, we used method with two perception modules. You can see a large difference between the minimal and maximal values for both graphs. In some episodes, the agent is able to achieve high rewards and prevents the episode from terminating early, and in others, it is not able to achieve a reward larger than zero, which leads to this episode terminating early. Looking at the videos (Video D.1.6.5), we can see that the agent is only able to cook one of the recipes.



(a) Approach with multiple perception modules (Section 5.3.1.1)

(b) Approach with multiple dLSTMs inside the manager (Section 5.3.1.2)

Figure B.4: The result of performing an ICA on 20000 collected goal vectors with an equal amount of vectors coming from episodes with tomato soup and episodes with salad to reduce the number of dimensions to 2.

C BenchMARL

This chapter outlines what we did in regard to MARL. Section C.1 presents the basic principles of MARL and Section C.2 shows the experiments we conducted.

C.1 MARL

MARL is a subfield of Reinforcement Learning, where multiple agents simultaneously act in a shared environment to accomplish a cooperative or competitive goal. Here, we only consider cooperative MARL, where multiple agents act together in the kitchen environment and aim to cook recipes together.

When talking about MARL, one usually differentiates between centralized and decentralized approaches. When using a centralized method, we have only one model that predicts the actions of all agents using global observations of the environment. If approaching MARL with a centralized approach, this can essentially be considered a single-agent Reinforcement Learning problem where the actions are permutations of the actions of the agents and the observations are concatenations of the observations of the individual agents. However, this approach implies that the action space increases exponentially with the number of agents, which can lead to an infeasible number of actions rapidly. In Sunehag et al. (2017), it was also discovered that centralized approaches can lead to inefficient policies being learned where only one agent is able to learn a useful policy while the other agent is discouraged from exploration to not hamper the learning of the other agent, leading to an overall worse reward.

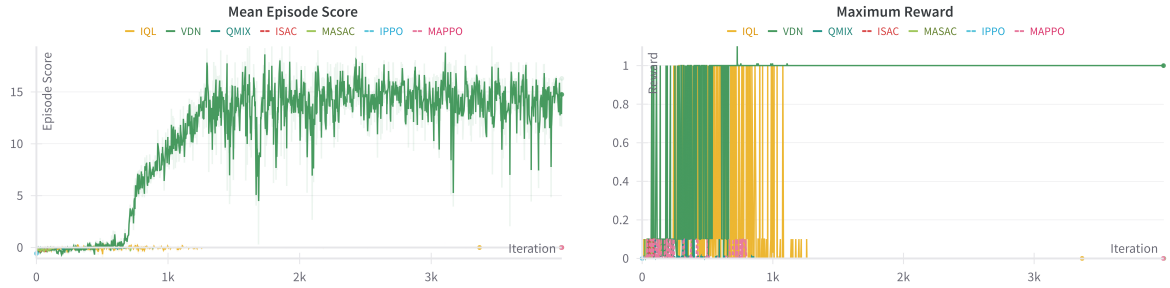
Decentralized methods, on the other hand, utilize models that only predict actions for single agents, using the local observations of the agents, which can differ from each other. One problem that arises from decentralized methods is that the dynamics of the environment change when the agents are updated, meaning that from the perspective

of one agent, the environment essentially changes over the training process as the other agents update their policies (Laurent et al., 2011). One approach to mitigate this limitation is to use Centralized training and decentralized execution (CTDE) methods. These methods follow a centralized approach for the training process only while using a decentralized approach for execution (Yu et al., 2022).

C.2 Experiments

For the experiment in this section, we used the reward configuration 'Reward 4' from Table A.2. In this setting, both agents have a shared reward. Figure C.2 shows the layout used for this experiment. The divider in the middle of the layout forces the two agents to cooperate as multiple items need to be handed across this divider in order to make tomato soup. In this experiment, we compared the performance of the MAPPO (Yu et al., 2022), IPPO (de Witt et al., 2020), IQL (Tan, 1997), VDN (Sunehag et al., 2017), QMIX (Rashid et al., 2018), ISAC (Bettini et al., 2024), and MASAC (Bettini et al., 2024). We ran each algorithm for 4000 algorithm iterations with 2000 transitions collected in each iteration, resulting in $8 \cdot 10^6$ steps in the environment.

Figure C.1 shows the result of this experiment. The episode score shows that VDN was the only algorithm that consistently achieved a nonzero reward. Looking at the maximum reward, we can also see that this algorithm did achieve rewards of value 1, indicating that the agent managed to serve the recipe. Video D.1.7.1 shows the two agents successfully cooking and serving tomato soup. Interestingly, as we can see in the maximum reward, IQL did manage to achieve rewards of value 1 at the beginning of the training process; however, after around 1300 algorithm iterations, the maximum reward went down to zero.



(a) episode score (b) Maximum reward

Figure C.1: Episode score and maximum reward. Note that the x-axis indicates the number of iterations in the algorithm and not the number of steps in the environment.

Figure C.2: The layout used for the multi-agent experiment

D External Material

D.1 Videos

This section contains links to videos of the trained agents acting in the environment.

D.1.1 Basic Reward Shaping

This section contains videos of runs from Section 5.2.1. Video D.1.1.3 shows a video of the first run where the agent successfully served tomato soup.

Video D.1.1.1 Reward config 'Reward 2': https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/basic_reward_shaping/no_trashcan_punishment.gif

Video D.1.1.2 Reward config 'Reward 3': https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/basic_reward_shaping/no_pickup_punish.gif

Video D.1.1.3 Reward config 'Reward 4': https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/basic_reward_shaping/success.gif

D.1.2 Layout Complexity

This section contains videos of runs involving a higher layout complexity described in Section 5.2.5. Video D.1.2.1 shows a run with randomly generated 5 by 5 layouts and Video D.1.2.2 shows a run with 7 by 7 layouts. Videos D.1.2.3 to D.1.2.6 show the runs with random size with padding and Videos D.1.2.7 to D.1.2.10 show the behavior of the agent in the run without padding for each size.

Video D.1.2.1 Random layout 5x5: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random5x5.gif

Video D.1.2.2 Random layout 7x7: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random7x7.gif

Video D.1.2.3 Random layout, random size, with padding, 4x4: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_padding/4x4.mp4

Video D.1.2.4 Random layout, random size, with padding, 5x5: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_padding/5x5.mp4

Video D.1.2.5 Random layout, random size, with padding, 6x6: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_padding/6x6.mp4

Video D.1.2.6 Random layout, random size, with padding, 7x7: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_padding/7x7.mp4

Video D.1.2.7 Random layout, random size, without padding, 4x4: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_no_padding/4x4.mp4

Video D.1.2.8 Random layout, random size, without padding, 5x5: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_no_padding/5x5.mp4

Video D.1.2.9 Random layout, random size, without padding, 6x6: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_no_padding/6x6.mp4

Video D.1.2.10 Random layout, random size, without padding, 7x7: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/layout_complexity/random_size_no_padding/7x7.mp4

D.1.3 Recipe Complexity

Videos of the runs described in Section 5.2.2.

Video D.1.3.1 Burger, run 1: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_1.gif

Video D.1.3.2 Burger, run 2: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_2.gif

Video D.1.3.3 Burger, run 3: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_3.gif

Video D.1.3.4 Burger, run 4: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_4.gif

Video D.1.3.5 Burger, run 5: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_5.gif

Video D.1.3.6 Burger, run 6: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_6.gif

Video D.1.3.7 Burger, run 7: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_complexity/run_7.gif

D.1.4 Recipe Count

Videos of the runs described in Section 5.2.3. The videos from runs with 4 recipes are all from the last run, where attempting to serve the wrong recipe is punished.

Video D.1.4.1 2 recipes, tomato soup: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/2_recipes/tomato_soup.gif

Video D.1.4.2 2 recipes, salad: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/2_recipes/salad.gif

Video D.1.4.3 2 recipes, orders: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/orders.gif

Video D.1.4.4 4 recipes, punish serving of the wrong recipe, chips: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/4_recipes/chips.gif

Video D.1.4.5 4 recipes, punish serving of the wrong recipe, onion soup: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/4_recipes/onion_soup.gif

Video D.1.4.6 4 recipes, punish serving of the wrong recipe, salad: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/4_recipes/salad.gif

Video D.1.4.7 4 recipes, punish serving of the wrong recipe, tomato soup: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/recipe_count/4_recipes/tomato_soup.gif

D.1.5 Model Sizes

Videos of the runs described in Section 5.2.6.

Video D.1.5.1 3×10^6 parameters: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/model_size/3m.gif

Video D.1.5.2 6×10^6 parameters: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/model_size/6m.gif

Video D.1.5.3 12×10^6 parameters: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/model_size/12m.gif

Video D.1.5.4 25×10^6 parameters: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/dreamer/model_size/25m.gif

D.1.6 FuN

Videos of the runs described in Section 5.3.

Video D.1.6.1 Simple layout, tomato soup: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/basic.gif

Video D.1.6.2 Simple layout, salad: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/salad.gif

Video D.1.6.3 2 recipes, multiple dLSTMs: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/recipe_count/multi_dlstm_2_recipes.gif

Video D.1.6.4 2 recipes, multiple dLSTMs, $\alpha = 1$: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/recipe_count/multi_dlstm_2_recipes_high_alpha.gif

Video D.1.6.5 2 recipes, multiple perception modules, episode automatically terminates after 50 steps with zero reward: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/recipe_count/multi_perception_2_recipes_autoterminate.gif

Video D.1.6.6 2 recipes, multiple perception modules: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/recipe_count/multi_perception_2_recipes.gif

Video D.1.6.7 4 recipes, multiple perception modules: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/fun/recipe_count/multi_perception_4_recipes.gif

D.1.7 BenchMARL

This section contains the videos from the BenchMARL experiment, described in Section C.2.

Video D.1.7.1 VDN: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/benchmarl/VDN.mp4

Video D.1.7.2 IQL: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/benchmarl/IQL.mp4

Video D.1.7.3 ISAC: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/benchmarl/ISAC.mp4

Video D.1.7.4 MAPPO: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/benchmarl/MAPPO.mp4

Video D.1.7.5 MASAC: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/benchmarl/MASAC.mp4

Video D.1.7.6 QMIX: https://gitlab.ub.uni-bielefeld.de/IchbinLuka/coop_cuisine_ba/-/raw/master/videos/benchmarl/QMIX.mp4