BAZIGOOSHI: A Hybrid Model of Reinforcement Learning for Generalization in Gameplay

Sara Karimi *†, Sahar Asadi†, Amir H. Payberah*
* KTH Royal Institute of Technology, Sweden
† King.com Ltd., Sweden
sara.karimi@king.com, sahar.asadi@king.com, payberah@kth.se

***Abstract*—While Reinforcement Learning (RL) is gaining popularity in gameplay, creating a generalized RL model is still challenging. This study presents BAZIGOOSHI, a generalized RL solution for games, focusing on two different types of games: (i) a puzzle game Candy Crush Friends Saga and (ii) a platform game Sonic the Hedgehog Genesis. BAZIGOOSHI rewards RL agents for mastering a set of intrinsic basic skills as well as achieving the game objectives. The solution includes a hybrid model that takes advantage of a combination of several agents pre-trained using intrinsic or extrinsic rewards to determine the actions. We propose an RL-based method for assigning weights to the pre-trained agents. Through experiments, we show that the RL-based approach improves generalization to unseen levels, and BAZIGOOSHI surpasses the performance of most of the defined baselines in both games. Also, we perform additional experiments to investigate further the impacts of using intrinsic rewards and the effects of using different combinations in the proposed hybrid models.**

(a) (b)

Fig. 1. (a) a level in Candy Crush Friends Saga (© 2024 King.com), and (b) two different levels in Sonic the Hedgehog Genesis.

I. INTRODUCTION

Deep Reinforcement Learning (DRL) has gained popularity in the development of games. However, it still faces several challenges, such as slow learning due to the sparse reward signals (rewarding the agent solely based on the win-or-lose outcome at the end of a game round) and difficulties in generalizing to new levels. Two intriguing games to study the above challenges are *Candy Crush Friends Saga (CCFS)*[1] and *Sonic the Hedgehog Genesis (Sonic)*[2] (Figure 1). CCFS is a match-3 puzzle game that presents properties such as stochasticity in state transitions, a large action space, and diversity in features and objectives across different levels. On the other hand, Sonic is a platform game that presents a challenge across various levels, wherein the player must perform specific actions, such as jumping to open doors to progress through the game map.

To overcome the sparsity issue, studies [1] and [2] suggest using denser reward functions that gradually reward agents as they approach level objectives. Moreover, Stout et al. [3] and Schmidhuber [4] propose incorporating a domain-specific intrinsic reward along with the environment-based reward. Another approach to address the challenges associated with sparse rewards is to define a set of reward functions, each reflecting specific strategies or skills possible in that game. Shin et al. [5] suggest defining a set of strategies and training a DRL model to estimate the probability of each strategy based on the state of the game. Lorenzo et al. [6] build on this approach by proposing a set of basic skills that enable agents to learn adaptable behaviors applicable across levels with diverse objectives.

Seijen et al. [7] present an approach to generalize RL models, utilizing a hybrid architecture that decomposes the reward function into multiple reward functions, with each function influenced by only a small number of state variables. However, these methods either employ a fixed reward function or use multiple reward functions without considering a dynamic combination. The dynamic combination could be achieved by having state-conditioned dynamically changing weights that control the extent of the impact of each reward. We investigate how using dynamic weights to emphasize different agents' decisions could lead to improved generalization across unseen levels in the two games.

This work extends our previous research CandyRL [8], an RL model for generalization in gameplay, aiming to overcome the above-mentioned limitations. This approach involves training multiple RL agents referred to as *pre-trained agents* incentivized to learn a set of *basic skills* that are learned through the use of different *extrinsic rewards* or *intrinsic rewards*. These pre-trained agents are then combined using a hybrid model to allow approaching new levels without re-

[1]https://www.king.com/

[2]Sonic the Hedgehog Genesis is the copyright product of SEGA and is referenced for discussion purposes only. The authors of this paper have no relationship with Sonic the Hedgehog Genesis or SEGA.

training from scratch on those levels. The extrinsic rewards are provided by the game environment to incentivize the agent to achieve the level objectives. The intrinsic rewards, on the other hand, are self-rewarded by the agent for fulfilling a more generalized goal that is not directly tied to the level objective. The combination of pre-trained agents is performed by taking a weighted average of their decisions. We explore three approaches for assigning weights to the pre-trained agents: (1) assigning static equal weights to all pre-trained agents, (2) assigning non-equal static weights using a heuristic, and (3) assigning non-equal dynamic weights to the pre-trained agents that a trained RL agent determines.

BAZIGOOSHI also extends CandyRL [8] by performing additional experiments and providing comparisons with additional baselines to verify the generalization of BAZIGOOSHI to a different type of game. This paper makes the following main contributions:

1) Assessment of BAZIGOOSHI, a hybrid RL approach of combining the decision of multiple pre-trained agents using different weighting methods on two distinct types of game environments, i.e., CCFS and Sonic, to verify the generalization ability of the proposed method compared to a number of defined or state-of-the-art baseline models.
2) An ablation study on BAZIGOOSHI to understand the impact of different combinations of pre-trained agents on model performance and verify the consistency of results across the games.
3) Analysis on the impacts of using intrinsic reward functions versus extrinsic rewards through empirical evaluation.

Based on the experimental results, BAZIGOOSHI shows superior performance compared to the pre-trained agents (used as the building blocks of the hybrid models) and performs comparably to the three considered baseline models across most levels of CCFS and Sonic. The experiments demonstrate that among proposed hybrid models, the heuristic approach of assigning weights results in, on average, a higher training win-rate on CCFS. However, the RL-based model that dynamically determines the weights shows a better test performance on unseen levels. These findings confirm that our proposed approach of learning dynamic weights through an RL model provides better generalizability for gameplay.

II. PRELIMINARY

This section provides an overview of the two games, CCFS and Sonic, including their objectives and environments.

A. Candy Crush Friends Saga

CCFS is a puzzle game that involves matching three or more objects of the same color to progress toward a specific objective, such as reaching a score threshold. The game board comprises a 9×9 tile grid that contains multiple types of game elements, including candies of various colors and types (as shown in Figure 1 (a)). In the game, a basic action is matching, which is defined as swapping two candies on the board to create a sequence of three or more candies of the same color lined up across the horizontal or vertical axis. Once matched, candies are removed from the board and replaced with new ones from the top or randomly spawned candies.

Fig. 2. Different types of special candies in CCFS game (© 2024 King.com).

Fig. 3. Different types of blocker layers in CCFS game (© 2024 King.com).

The game board in CCFS comprises three main types of elements: *regular candies*, *special candies*, and *blockers*. Regular candies are the most frequently occurring elements in seven colors. The second type, special candies with six different variations shown in Figure 2 (in order from left to right: color bomb, color bomb, vertical and horizontal striped candy, wrapped candy, coloring candy, and fish), are generated by matching at least four candies of the same color creating specific shapes (e.g., "L" or "T" shapes) on the game board. Each special candy has a distinct effect that can be activated in different ways, including being involved in a match of candies of the same color being swapped with an adjacent candy or special candy. Blockers form the third type of element and are represented by fixed-sized layers placed on top of a candy on the board and block access to the candy, thus limiting the interactability of the board (Figure 3). Each *layer* of blocker is removed by making a match that involves candies located in the neighboring tiles of the blocker.

The CCFS Environment and Objectives: The CCFS *environment* is episodic, with each *episode* corresponding to a full round of gameplay on a level. The state space of the CCFS environment is represented in a three-dimensional format with dimensions of size $9 \times 9 \times 32$, where the first two dimensions represent the grid of the game board, and the third dimension encodes each of the 32 different elements (including different types of candies and blockers) using one-hot encoding channels. We use the same encoding as the one presented in [9] (see Figure 4)). To map to the CCFS environment, we include 32 channels representing the game elements.

As mentioned before, the definition of *action* in CCFS is swapping two arbitrary game elements located on adjacent tiles in the 9×9 game board. Therefore, to encode the possible actions in the action space, we assign a unique ID to the edges between any pair of adjacent tiles, labeling the action that is relevant between that pair of tiles. This encoding approach results in an *action space* of size 144 (Figure 5) [9]. A *policy* is a mapping from $9 \times 9 \times 32$ states to the available 144 actions. Each level in CCFS has a specific objective, and there are different categories of objectives in CCFS[3]. We focus on the *spread the jam* objective category, which requires players to cover the entire board with jam by making matches that involve tiles already covered by jam within the level-specific

[3]https://candycrushfriends.fandom.com/wiki/Levels

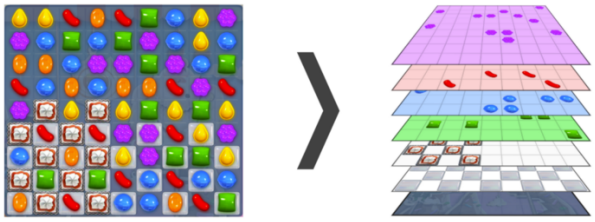


Fig. 4. An example of game board representation and state encoding [9].

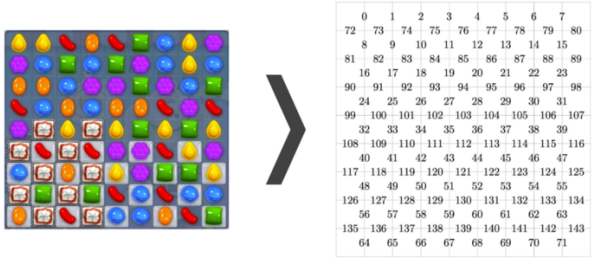


Fig. 5. Action space representation in the game board [9].

moves limit. We present our technique using the spreading jam objective, but our approach also applies to other objectives.

B. Sonic the Hedgehog Genesis

Sonic the Hedgehog Genesis (Sonic) is a video game originally released on the game console Sega [4]. A lightweight version of this game is released by Gym Retro [10], a project aimed at creating RL environments from various emulated video games for research purposes. Figure 1(b) shows a few frames of the game representing part of the game map and the playable character. In this game, the player controls the playable character and drives the character towards a finish line on the map while tackling a set of challenges, including jumping over obstacles, triggering buttons to open doors, avoiding enemies, etc.

The Sonic game is divided into *zones*, and each zone includes several *acts*. We define a *level* as an act in a zone, denoted by `zone.act`, e.g., `GreenHillZone.Act1`. Changing acts in the same zone (e.g., `GreenHillZone.Act1` and `GreenHillZone.Act2`) has a minor impact on the levels compared to changing zones (e.g., `GreenHillZone.Act1` and `StarLightZone.Act1`). Each zone has a unique set of textures and objects and acts inside a zone differ in spatial layouts of these textures and objects. The rules and objectives remain the same across different levels.

The Sonic Environment and Objectives: The Sonic *environment* is episodic, where each *episode* can end on three conditions: (i) the player completes a level successfully (in this benchmark, completing a level corresponds to passing a certain horizontal offset within the level), (ii) the player loses a life, or (iii) if 4500-time steps (approximately five minutes) have elapsed in the current episode. The observations returned by the environment are 24-bit RGB images of 320 pixels wide and 224 pixels tall. In the preprocessing step, we convert the observation images to grayscale, then crop them to a square shape and resize them to size 84×84 pixels.

In Sega games, the action space contains 12 buttons, represented as a vector: [`B`, `A`, `MODE`, `START`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `C`, `Y`, `X`, `Z`]. The actions are then encoded as binary vectors of size 12, where 1 means "pressed" and 0 means "not pressed." For example: {`LEFT`} to $[0,0,0,0,0,0,1,0,0,0,0,0]$, and {`DOWN`, `B`} encodes to $[1,0,0,0,0,1,0,0,0,0,0,0]$. However, in Sonic, only the following button combinations are valid actions [11]: {{}, {`LEFT`}, {`RIGHT`}, {`LEFT`, `DOWN`}, {`RIGHT`, `DOWN`}, {`DOWN`}, {`DOWN`, `B`}, {`B`}}.

III. Method

This section introduces the reward functions utilized in training RL agents for the games CCFS and Sonic, followed by an explanation of the BaziGooshi method that combines pre-trained agents with a hybrid architecture.

A. Reward Functions

Below, we explain the reward functions to train the pre-trained RL agents for the games CCFS and Sonic.

1) CCFS Rewards: For CCFS, we establish two types of rewards: *extrinsic* rewards that reflect the progress towards fulfilling level objectives provided by the environment, and *intrinsic* rewards that aid the agents in acquiring general skills.

In this study, we employ two extrinsic reward functions for CCFS, namely *Progressive Jam (PJ)* and *Delta Jam (DJ)* [1]. If an agent's action spreads jam to at least one additional tile, the reward function PJ grants the agent a reward equivalent to the total number of tiles currently covered by jam, denoted by J. The reward is then normalized by the total number of tiles on the board, B. Let j be the number of new board tiles covered by jam after the agent takes action a in state s. Using these parameters, we define the reward R for the agent as:

$$R(s,a) = \begin{cases} \frac{J}{B}, & j > 0 \\ 0, & j = 0 \end{cases} \tag{1}$$

In contrast to PJ, DJ rewards the agent proportional to the number of new tiles that are covered in jam:

$$R(s,a) = \frac{j}{B} \tag{2}$$

The extrinsic rewards mentioned earlier are specific to particular objectives (such as spreading jam). However, we need to define more generalized rewards to create a more adaptable system where agents can play different levels with varying objectives. One approach to achieving this is to use intrinsic rewards, allowing agents to reward themselves for accomplishing goals that may not be directly related to the level's objective. These intrinsic rewards are modeled after the fundamental skills that human players develop as they play and progress through different levels of CCFS. In this paper, we use three intrinsic rewards for CCFS: *Damage Blocker*

[4]https://www.sega.com/

(DB), *Candy Creation (CC)*, and *Candy Usage (CU)*, which we explain in detail below [6].

In CCFS, the blockers present at each level can vary in type and number of layers. To account for this variation, we introduce the DB reward function, which rewards the agents based on the amount of damage inflicted on each blocker of type b, which is further normalized by the initial number of blockers b_0 of that type. The DB function is formulated as follows:

$$R(s,a) = \sum_{b \in B} \frac{d^{(b)}}{b_0} \tag{3}$$

where B denotes the set of all blockers and $d^{(b)}$ represents the number of blockers of type b damaged by action a.

Another intrinsic reward function used in this paper is CC, which incentivizes agents based on the number of special candies they create of each type following an action. Special candies require a larger number of candies and particular shapes in the match to be created. Due to the presence of blockers and spatial constraints exposed by the board shape, the occurrence of special candies has different frequencies and is rare compared to regular candies. To account for this variance and to prevent skewed results, we normalize the values of each type of candy accordingly. The formula for calculating CC is as follows:

$$R(s,a) = \sum_{x \in X} c^{(x)} \times \big(1 - \frac{\mu^{(x)}}{\sum_{x' \in X} \mu^{(x')}}\big) \tag{4}$$

where $c^{(x)}$ is the number of special candies of type x that are created after doing action a, $\mu^{(x)}$ represents the average frequency of creating that specific special candy type in a single game episode. The denominator of the equation accounts for the total frequency of creating all special candies. Empirical evidence supporting the effectiveness of this intrinsic reward in generating more special candies is presented in section IV-D.

Although the CC function increases the creation of special candies, the ultimate goal is to use them in gameplay. Therefore, we introduce the CU function, which rewards agents based on their usage of special candies:

$$R(s,a) = \sum_{x \in X} u^{(x)} \times (1 - \frac{\mu^{(x)}}{\sum_{x' \in X} \mu^{(x')}}) \tag{5}$$

where $u^{(x)}$ represents the number of special candies of type x involved in the action a.

2) Sonic Rewards: In Sonic, we use the default *extrinsic* reward functions provided by the game environment and define a new *intrinsic* reward to train the agents. Below, we explain these three rewards: *Contest*, *X-position (x-pos)*, and *Progressive Rings (PR)*, where Contest and x-pos are extrinsic reward functions, and PR is an intrinsic reward function.

The Contest reward function rewards the agent proportional to the new horizontal progress (at each time-step) and incentivizes the agent to make more horizontal progress in less time. It consists of two components: a horizontal offset reward and a completion reward. The horizontal offset reward is normalized so that in all levels, the agent's total reward will be 9000 when it reaches the end of the level. The completion bonus is introduced to encourage the agent to finish the level quickly. This is formulated as

$$R(s_t,a_t) = \begin{cases} (p_t - p_{t-1}) \times 9000, & p_t < 1 \\ (p_t - p_{t-1}) \times 9000 + (1 - f_t) \times 1000, & p_t \geq 1 \end{cases} \tag{6}$$

where p_t is the x-axis progress (after doing action a_t) normalized by the full horizontal length of the level, and f_t is the fraction of played frames at time t divided by the frame limit (a number after which the completion reward drops to zero). This part of the reward encourages the agent to finish faster as in later time-steps f_t gets closer to 1, and according to the formula 6, the agent gets a smaller reward [11]. The x-pos reward function, as shown in Equation (7), rewards the agent by the amount of progress on the x-axis.

$$R(s_t,a_t) = x_t - x_{t-1} \tag{7}$$

where x_t is the position of agent on the x axis after action a_t.

In the PR reward function, if the agent's current action leads to collecting one or more ring(s), the agent is rewarded with the total number of collected rings. This is formulated as

$$R(s,a) = \begin{cases} \frac{C}{T}, & c > 0 \\ 0, & c = 0 \end{cases} \tag{8}$$

where c is the number of new rings collected after the doing action a, C is the total number of rings collected so far, and T is the total number of rings available to collect in that level. We theorize that incentivizing the agent to collect rings encourages it to use the jump actions more frequently, improving the agent's progression through the platforms and obstacles.

B. The Hybrid Models - BAZIGOOSHI

In the preceding section, we use the extrinsic and intrinsic reward functions defined in Section III-A to build our model BAZIGOOSHI. We define a *pre-trained agent* as an RL agent trained using either an intrinsic or extrinsic reward function. BAZIGOOSHI is a hybrid architecture that combines these pre-trained agents to improve generalization. This paper outlines three ways of combining pre-trained agents in BAZIGOOSHI: (1) *Average Bagging (AB)* that utilizes an unweighted ensemble of pre-trained agents, (2) *Heuristic-based Average Bagging (HAB)* that incorporates heuristics to assign static weights to the pre-trained agents based on their significance, and (3) *Meta Controller Average Bagging (MCAB)* that uses an RL controller to assign weights to each pre-trained agent dynamically. In the following sections, we provide a detailed explanation of these three hybrid approaches.

1) Average Bagging (AB): AB is a technique that combines multiple pre-trained agents to enhance the agent's generalization ability and reduce variance [12]. As depicted in Figure 6, the AB model receives the current game state and feeds it simultaneously to each pre-trained agent, which generates a set of Q-values for every possible action. The Q-values for each action across different pre-trained agents are then aggregated

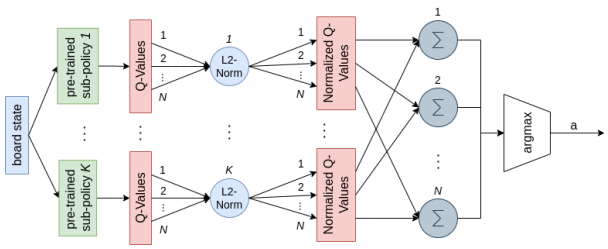


Fig. 6. Action selection in AB. The current game state is fed as input to all the pre-trained agents, and each pre-trained agent returns a set of N Q-values. Each set of Q-values is normalized before summing them with the corresponding Q-values from other pre-trained agents.

by computing their average. Subsequently, an action with the highest average Q-value is selected, as shown in (9).

$$a = \underset{a}{\operatorname{argmax}} \left(\frac{1}{K} \sum_{k=1}^{K} Q_k(s, a_n), n \in \{1, 2, ..., N\} \right) \quad (9)$$

where $Q_k(s, a_n)$ represents the Q-value of action a_n generated by the pre-trained agent k, and s denotes the input state. The number of pre-trained agents is K, and N is the number of actions in the action space. Since each pre-trained agent uses different reward scales, their Q-value outputs may have varying ranges, which could introduce bias when aggregating them. To address this issue, we apply L2-normalization to scale all the Q-values between $[0, 1]$.

2) Heuristic-based Average Bagging (HAB): The AB model is limited in assigning the same weight to all pre-trained agents. However, our ablation study in Section IV-E shows that some pre-trained agents perform better than others. To further exploit this, we propose the HAB model, which assigns different weights to pre-trained agents based on a simple heuristic. This heuristic calculates the win-rate of each pre-trained agent on specific levels after training for a fixed number of episodes and returns weights proportional to these values. Similar to AB, HAB feeds the current state to each pre-trained agent and collects their Q-values. However, instead of averaging the Q-values equally, HAB performs weighted averaging using the calculated weights. This is formulated in Equation (10):

$$a = \underset{a}{\operatorname{argmax}} \left(\frac{1}{K} \sum_{k=1}^{K} w_k Q_k(s, a_n), n \in \{1, 2, ..., N\} \right) \quad (10)$$

where w_k is in the range $[0, 1]$ and is calculated as follows:

$$w_k \propto \{\text{training win-rate of the } k\text{-th agent}\} \quad (11)$$

This heuristic highlights the significance of each pre-trained agent on a specific level within the hybrid model.

3) Meta Controller Average Bagging (MCAB): In the HAB model, we use a heuristic to assign static weights to pre-trained agents based on their final training win-rates on specific levels. However, this approach does not consider the importance of different skills conditioned on the state of the game. To address this, we propose the MCAB model, which uses a *meta-controller* RL agent to dynamically learn a set of weights for

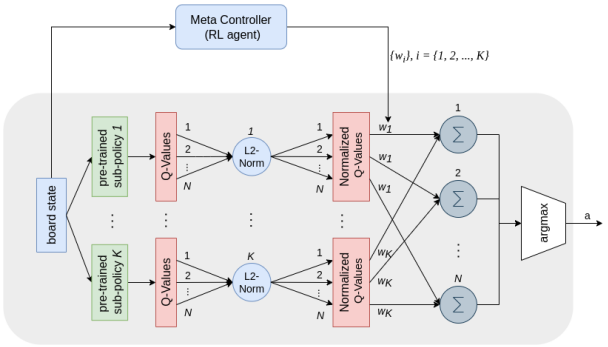


Fig. 7. Action selection in MCAB. The current game state is fed as input to the pre-trained agents and the RL meta-controller agent. The meta-controller agent returns a set of weights, and each pre-trained agent returns a set of N Q-values. Each set of Q-values is normalized and multiplied by their weights before they are summed with the corresponding Q-values from other pre-trained agents.

combining pre-trained agents conditioned on the game's current state. At each game step, the meta-controller is given the state as input and returns a set of weights that are then assigned to the pre-trained agents used in the hybrid model. The weights reflect the importance of the corresponding pre-trained agent. The MCAB architecture is illustrated in Figure 7.

To train the meta-controller agent, we use policy gradient methods like REINFORCE [13] and Proximal Policy Optimization (PPO) [14] since they are suitable for continuous action spaces. Specifically, in the experiments provided in this paper, we use PPO on continuous action space to train the meta-controller agent.

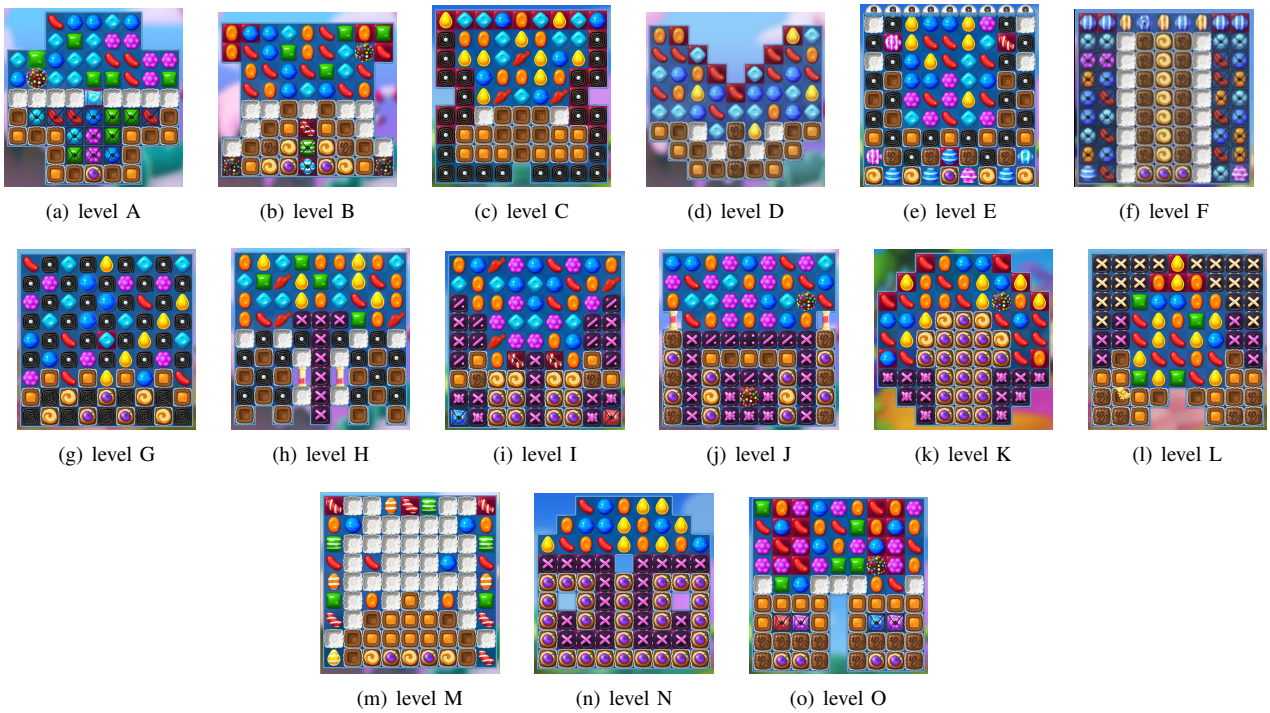
The meta-controller agent learns to output the parameters of a Dirichlet distribution and draws a sample of size K as the weights of pre-trained agents for computing a linear combination over their Q-values. The MCAB model then selects an action according to Equation (10) where the weights (w_k) are samples taken from a multivariate distribution, in our case, a Dirichlet distribution with probability density function as follows:

$$f(w_1, \ldots, w_K; \alpha_1, \ldots, \alpha_k) = \frac{1}{B(\alpha)} \prod_{i=1}^{K} w_i^{\alpha_i - 1} \quad (12)$$

where $\sum_{k=1}^{K} w_k = 1$ $(w_k \geq 0)$, α are the parameters of the distribution, K is the number of pre-trained agents, and B is the multivariate beta function.

IV. Evaluation

This section provides details of the experiments conducted on CCFS and Sonics, starting with the experiment settings and introducing the baselines. We then present the results, which are divided into five parts. First, we demonstrate the effectiveness of using intrinsic rewards compared to extrinsic rewards. Second, we conduct an ablation experiment to study the impact of different combinations of pre-trained agents in the hybrid model. Third, we compare different weighting methods, including AB, HAB, and MCAB, in the hybrid model. Fourth, we compare the hybrid models with agents



(a) level A (b) level B (c) level C (d) level D (e) level E (f) level F

(g) level G (h) level H (i) level I (j) level J (k) level K (l) level L

(m) level M (n) level N (o) level O

Fig. 8. The initial board representation of CCFS training and test levels. Levels A-F are training levels, and the rest are testing levels (© 2024 King.com).

trained with sparse rewards and agents pre-trained with intrinsic or extrinsic rewards. Finally, we draw a comparison between BAZIGOOSHI and three baseline models. In this work, we have executed over 2500 training and evaluation runs, the compiled results of which are presented in tables in this section.

A. CCFS Experimental Settings

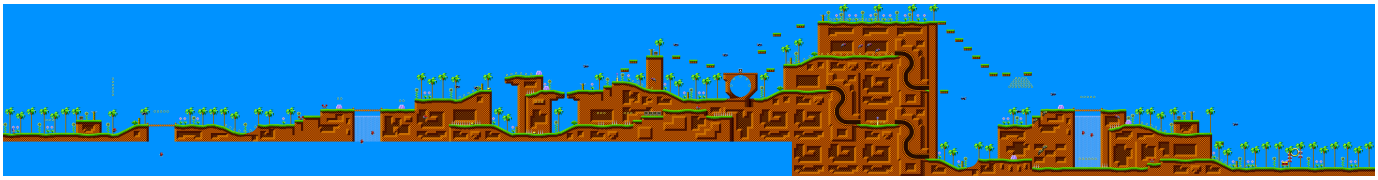
The CCFS models are trained for 80,000 *episodes*, and each episode consists of one round of gameplay resulting in a win or loss. When evaluating the models, each episode is associated with a unique random seed different from those used during training. This randomness is essential when the agent is tested on the levels it has been trained on. The *win-rate* is the metric we use to evaluate the performance of the trained models. It is computed as the proportion of episodes that result in a win state over all the episodes in a given trial. The win-rate value lies within the $[0, 1]$.

We train the agents on six levels with different settings (levels A, B, C, D, E, and F in Figure 8) and test them on nine new unseen levels (levels G, H, I, J, K, L, M, N, and O in Figure 8) with types of candies and blockers different than the ones present in the training levels. The levels have been selected in a way to cover a varied range of gameplay difficulties and a varied selection of game elements. Figure 8 depicts the initial board representations of these levels. The reported win-rate for each test level in Table IV is the average of the win-rates obtained from evaluating all the trained models on that level. Also, each evaluation of a test level has been repeated using three different sets of random seeds. We conducted all evaluation experiments by running 1000 game rounds.

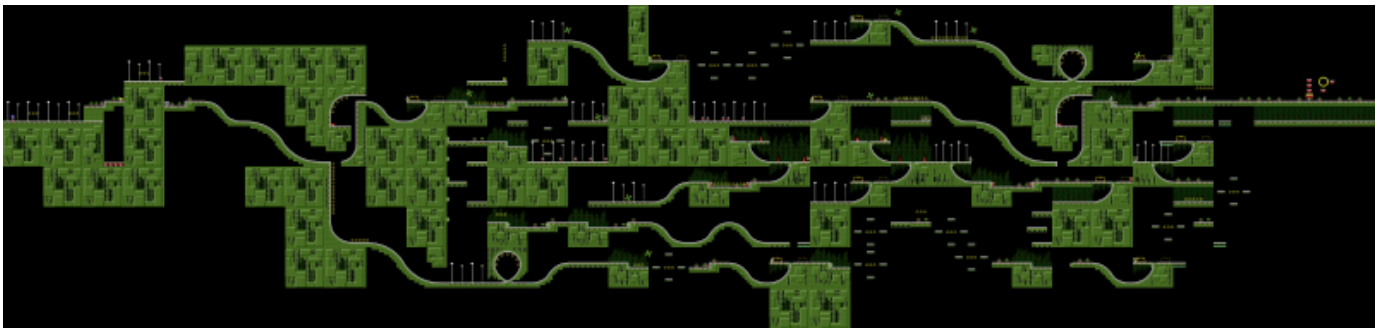
For training the pre-trained agents, we utilize DQN [15], while for training the meta-controller agent in the MCAB model, we use PPO [14]. The hyperparameters for our DQN models are primarily based on those used in the original DQN paper [15]. Nevertheless, certain hyperparameters, including the target network update, prediction network update, and discount factor, are sourced from a study on the CCFS environment [1], where a hyperparameter search was conducted. Similarly, we follow most PPO hyperparameters from the original PPO paper [14]. Table I provides a detailed summary of the DQN and PPO agents' hyperparameters. Through a limited architecture search, we came up with a network architecture that is different from the original DQN paper for CCFS. It consists of five convolutional layers of filter size 35 and kernel size 3 with ELU activation function followed by two dense layers with sizes 999 and 144 (i.e., corresponding to the number of actions). The same architecture is used in the PPO agent on CCFS; however, the last layer in that network is of size equal to the number of pre-trained agents.

B. Sonic Experimental Settings

In the Sonic environment, each training run comprises six million game frames, ending in one of the conditions mentioned in Section II-B. To train the agents, we use four levels from different zones (`GHZ.Act1`, `SLZ.Act1`, `SYZ.Act2` and `MZ.Act1`) and consider six unseen levels for testing (`GHZ.Act2`, `GHZ.Act3`, `SLZ.Act2`, `SLZ.Act3`, `SYZ.Act1` and `MZ.Act2`). We use `GHZ`, `SLZ`, `SYZ` and `MZ` to abbreviate `GreenHillZone`, `StarLightZone`, `SpringYatdZone` and `MarbleZone` zones, respectively. The full map representations of `GHZ.Act1` and `SLZ.Act1` are depicted in Figure 9.



(a)



(b)

Fig. 9. Sonic level example. The figures (a) and (b) show `GHZ.Act1` and `SLZ.Act1` levels, respectively.

TABLE I
THE HYPERPARAMETERS FOR TRAINING AGENTS USING DQN AND PPO.

hyperparameter	DQN	PPO
learning rate	5e-4	1e-4
epsilon	0.01	0.1
batch size	32	N/A
replay capacity	50000	N/A
network update step	100	10
discount factor	0.5	0.99
optimizer	Adam	Adam
GAE discount	N/A	0.95
epochs per update	N/A	10
actor sequence length	N/A	128

We use *average score* to evaluate agents' performance, which is the average of the total reward per episode across all episodes [11]. Each score reported for a test level in Table VI is an average of the evaluation score of all the trained models on that test level, and each evaluation on a test level has been repeated using three different random seeds. The evaluation experiments are performed over a maximum of 10,000 game frames. We use Rainbow DQN [16] to train all Sonic's pre-trained agents. All Rainbow DQN hyperparameters, including the network architecture, are adapted from the original Sonic paper [11].

C. Baselines

This section briefly describes the baseline models to which we compare our hybrid models.

Binary Hybrid Model: The Binary Hybrid Model (BHM) is similar to the MCAB hybrid model with a minor weight difference. The controller in this model, similar to the controller in MCAB, returns a set of weights, but in the end, only the pre-trained agent with the highest weight decides the next action, and the decisions of other pre-trained agents are ignored. This method is not introduced as part of the BAZIGOOSHI models as it does not use an ensemble of agents but rather learns to select one of the pre-trained agents conditioned on the game's current state. The motivation for adding this model to the baselines is to highlight the importance of a weighted hybrid model like MCAB.

Combined Reward Agent: The Combined Reward Agent (CRA) is an agent that uses a linear combination of the defined rewards as a reward function for each game environment. For example, in CCFS, the reward is the (normalized) sum of PJ, CC, CU, and DB reward functions. CRA is another relevant baseline to benchmark against since it is trained with reward information from all the different subtasks (pre-trained agents).

Hybrid Reward Architecture: The Hybrid Reward Architecture (HRA) [7] is a hybrid model that provides a strategy for building easy-to-learn value functions. It decomposes the reward function of the environment into k different reward functions. These sub-rewards are assigned separate RL agents trained in parallel using off-policy learning. More specifically, a network with a shared body and k separate heads is used to learn the value functions corresponding to each of the k agents. The Q-values of the current state from each agent are then (linearly) combined into a single value for each action. The current action is chosen based on the combined Q-values.

We consider HRA as another baseline in our benchmarking of hybrid models. It is worth mentioning that, in HRA, the decomposition of the reward function should result in many sub-rewards affected only by a small subset of state space. However, to perform a fairer comparison in this study, we consider the same set of reward functions used by the BAZIGOOSHI models on each game environment and train the HRA model for the same number of episodes as BAZIGOOSHI.

All hyperparameters are taken from the original HRA paper, except for the network architecture, where we use the same architectures mentioned in sections IV-A and IV-B for each game.

D. Intrinsic vs Extrinsic

This experiment provides more transparency on the effectiveness of using intrinsic reward functions compared to extrinsic rewards through empirical evidence. Figure 10 shows the effect of using an intrinsic reward (CC) versus an extrinsic reward (PJ) on the probability of the creation of special candies (described in II-A) as well as the probability of a match-3 action in CCFS. Figures 10(a)-10(g) show the significant effect of using CC reward on increasing the number of special candies. In Figure 10(h), we see an increase in the number of created special candies, resulting in a higher win-rate in the agent trained with an intrinsic reward function. In the rest of this section, we show that this is also pivotal for achieving generalization across levels of different objectives.

E. Ablation Study

In this section, we report on an ablation experiment performed on both CCFS and Sonic environments. This study investigates the impact of different combinations of pre-trained agents (e.g., PJ, DJ, CC, CU, and DB in CCFS) in the AB hybrid model. The results of this experiment on CCFS confirm that assigning weights to these agents is necessary, as they contribute to the overall performance differently. Figure 11 illustrates the win-rate of the AB model with each pre-trained agent excluded from the combination. For instance, the line labeled DB shows the win-rate of the AB model without the pre-trained agent using the DB reward function, while the decisions from the remaining pre-trained agents are combined for the AB model. Each line in the plots presents an average of three runs with different random seeds. Figure 11 demonstrates that excluding specific pre-trained agents from the hybrid model affects the overall win-rate differently. For instance, removing CC decreases win-rate across all three levels, while excluding DB distinctly impacts each level.

To further investigate this hypothesis, we conducted a similar ablation experiment on Sonic, and the results, presented in Table II, further support our claim. Each score in the table is an average of three runs with different seeds, and it shows that excluding a pre-trained agent affects the overall scores. Similar to the results of the ablation study on CCFS, the Sonic study suggests that the hybrid model's performance could be affected significantly when different combinations of pre-trained agents are used. These findings motivated the proposition of non-uniform and trainable weights for the pre-trained agents in the hybrid models.

TABLE II
THE ABLATION STUDY IN SONIC: THE IMPACT OF EMPLOYING VARIOUS PRE-TRAINED AGENT COMBINATIONS ON WIN-RATES IN THE AB MODEL.

Excluded agent	Average score
PR	1733
x-pos	1917
Contest	1611

TABLE III
THE AGGREGATED WIN-RATES OF PRE-TRAINED AGENTS AND HYBRID MODELS ON CCFS TRAINING LEVELS.

Agents	**Level**					
	A	B	C	D	E	F
Random	0.045	0.000	0.004	0.079	0.006	0.004
Sparse	0.066	0.008	0.008	0.126	0.003	0.003
DJ	0.136	0.010	0.030	0.141	0.016	0.007
PJ	0.098	0.029	0.024	0.250	0.018	0.014
DB	0.212	0.059	0.041	0.270	0.018	0.019
CU	0.085	0.038	0.039	0.263	0.019	0.023
CC	0.331	0.014	0.053	0.436	0.050	0.010
CRA	0.240	0.036	0.047	0.318	0.038	0.022
BHM	0.069	0.022	0.037	0.288	0.021	0.003
HRA	0.093	0.028	0.036	0.237	0.027	0.006
AB	0.400	0.063	0.135	0.323	**0.076**	0.040
HAB	**0.533**	**0.066**	**0.159**	**0.560**	0.037	**0.045**
MCAB	0.379	0.060	0.129	0.391	0.053	0.045

F. Hybrid Model vs. Pre-Trained Agents

In this section, we compare the average win-rate and score of the hybrid models with several pre-trained agents trained using three rewarding approaches: (i) intrinsic skill-based rewards, (ii) extrinsic objective-based rewards, and (iii) sparse rewards. As shown in Table III, the three hybrid models (AH, MCAB, and HAB) in CCFS surpass the pre-trained agents on training performance. These results confirm that hybrid architectures enable more informed decision-making by considering the choices of various pre-trained agents. Another interesting observation is that the performance gap between the hybrid model and the pre-trained agents increases on more difficult levels (e.g., G and I in Table IV), underscoring the advantages of hybrid architectures in tackling more complex challenges. Moreover, when assessing the generalization performance of the models on test levels (Table IV), the MCAB hybrid model outperforms all of the pre-trained agents.

We can see a similar trend in the results of experiments on the Sonic environment. As shown in Table V, the performance of the hybrid models (especially MCAB) far surpasses that of the pre-trained agents (PR, x-pos, and Contest) in all but one case. On level `SLZ.Act1`, the PR agent receives a slightly higher score than AB and MCAB. Also, looking at the generalization performance on the unseen levels (Table VI), the hybrid models outperform all pre-trained agents (used as their building blocks) based on average scores across test levels. These results confirm the benefits of hybrid models compared to a single policy (pre-trained) agent in terms of both training and test performance.

G. Comparison of Hybrid Models

Our study uses hybrid models to train agents in both CCFS and Sonic environments. Specifically, we examine three variations thoroughly explained in Section III (i.e., AB, HAB, MCAB). Within CCFS, we focus on four pre-trained agents,

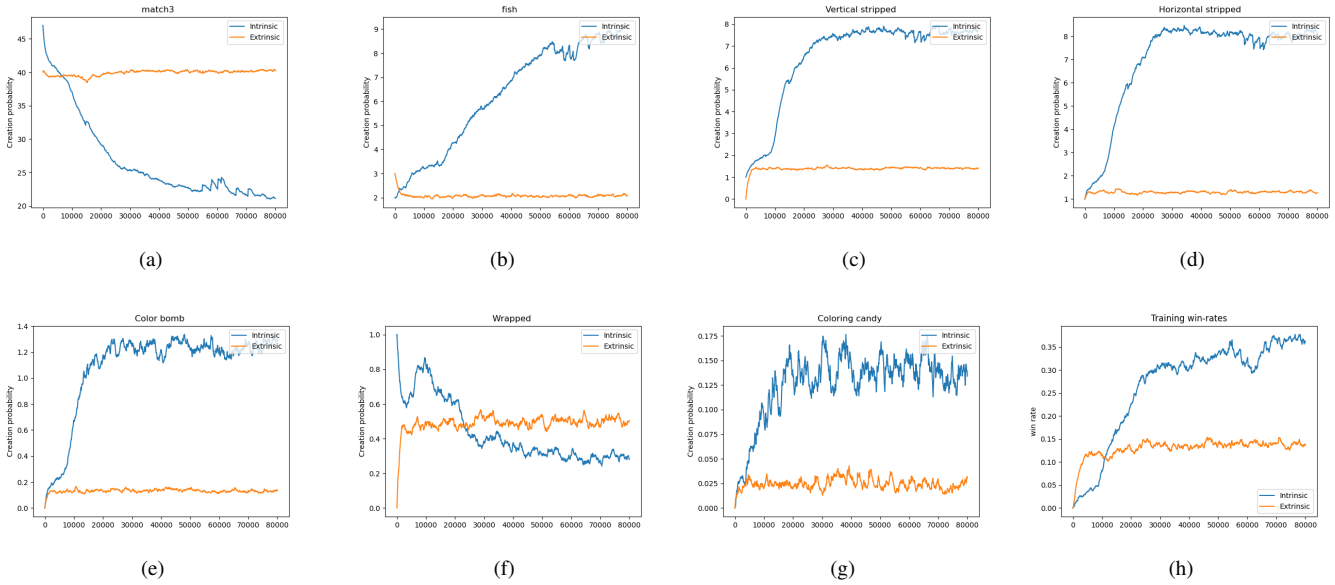


Fig. 10. (a)-(g) Probability of a match-3 action and probability of creation of each type of special candy in CCFS using an agent trained with extrinsic reward versus an agent trained with an intrinsic reward on level A. (h) Win-rates of these agents on level A. The x-axis shows the number of training episodes.

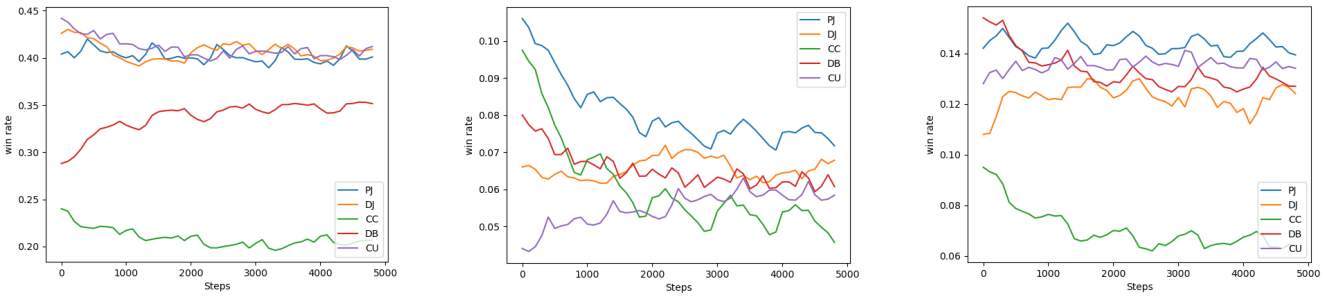


Fig. 11. The ablation study in CCFS - Plots showing the impact of using various combinations of the pre-trained agents on win-rates of the AB model on levels A, B, and C. Each line indicates the performance of an AB model with the specified pre-trained agent being excluded.

each with a distinct reward function: PJ, CC, CU, and DB. After observing the high similarity between DJ and PJ and the superior generalization ability of PJ, we decided to exclude DJ and only consider PJ in our combinations. Our analysis, detailed in Tables III and IV, involved evaluating the models' performance by measuring their average win-rate on both training and new, unseen test levels to assess their generalization ability.

As depicted in Table III, the HAB model, which selects weights proportionally to the final training win-rate of pre-trained agents, demonstrates superior performance on the training levels (A-F) compared to the other hybrid models in CCFS. This result confirms that the HAB model's weighting system more effectively highlights the importance of each pre-trained agent on the seen levels, in contrast to the MCAB model's learned weights. However, upon evaluating the agents' generalization abilities across unseen levels (levels G-O shown in Table IV), the MCAB model outperforms the other hybrid models on most levels. This outcome suggests that the MCAB model learns a more generalized set of weights, which more accurately reflect the overall importance of each pre-trained agent across various levels. Despite the HAB model's success on the seen levels, it still falls short in terms of generalization compared to the MCAB model. For instance, the HAB model fails to secure a win on all five trials of level F.

We only consider AB and MCAB hybrid models for training in the Sonic environment. In this environment, these hybrid models combine pre-trained agents trained with each of the three defined reward functions (e.g., Contest, x-pos, and PR). We evaluate the performance of models in terms of the average score on both the training and test levels. These results are presented in Tables V and VI. For comparing the performance of AB and MCAB hybrid models, we follow the evaluation procedure provided in [11] and look at the average performance of the agents across train and test levels (e.g., considering the average score of each agent on `GHZ.Act1` and `SLZ.Act2` as the training score of that agent).

Considering this evaluation method, the agent trained with

TABLE IV
THE AGGREGATED WIN-RATES OF PRE-TRAINED AGENTS AND HYBRID MODELS ON CCFS TEST LEVELS.

Agents	Level								
	G	H	I	J	K	L	M	N	O
Random	0.002 ± 0.001	0.002 ± 0.000	0.000 ± 0.000	0.002 ± 0.001	0.010 ± 0.002	0.002 ± 0.001	0.005 ± 0.001	0.000 ± 0.000	0.001 ± 0.001
Sparse	0.007 ± 0.001	0.004 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.015 ± 0.002	0.000 ± 0.000	0.008 ± 0.000	0.000 ± 0.000	0.001 ± 0.000
DJ	0.012 ± 0.001	0.008 ± 0.001	0.000 ± 0.000	0.000 ± 0.000	0.025 ± 0.002	0.001 ± 0.000	0.017 ± 0.001	0.001 ± 0.001	0.002 ± 0.000
PJ	0.018 ± 0.002	0.014 ± 0.002	0.000 ± 0.000	0.000 ± 0.000	0.033 ± 0.002	0.002 ± 0.000	0.029 ± 0.001	0.002 ± 0.001	0.007 ± 0.001
DB	0.017 ± 0.002	0.012 ± 0.001	0.000 ± 0.000	0.000 ± 0.000	0.025 ± 0.003	0.000 ± 0.000	0.027 ± 0.001	0.001 ± 0.001	0.001 ± 0.000
CU	0.024 ± 0.000	0.019 ± 0.002	**0.001 ± 0.000**	0.001 ± 0.000	0.045 ± 0.001	0.003 ± 0.000	0.024 ± 0.002	0.003 ± 0.000	0.010 ± 0.002
CC	0.022 ± 0.001	0.011 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.038 ± 0.001	0.000 ± 0.000	0.028 ± 0.001	0.002 ± 0.000	0.002 ± 0.001
CRA	0.031 ± 0.002	0.022 ± 0.001	0.000 ± 0.000	0.001 ± 0.000	0.045 ± 0.001	0.002 ± 0.000	0.034 ± 0.003	0.003 ± 0.001	0.011 ± 0.002
BHM	0.033 ± 0.022	0.023 ± 0.011	**0.001 ± 0.001**	0.001 ± 0.001	0.045 ± 0.019	0.003 ± 0.002	0.042 ± 0.022	0.004 ± 0.002	0.013 ± 0.008
HRA	0.016 ± 0.000	0.001 ± 0.000	**0.001 ± 0.000**	**0.036 ± 0.000**	0.002 ± 0.000	**0.023 ± 0.000**	0.004 ± 0.000	**0.006 ± 0.001**	0.017 ± 0.001
AB	0.063 ± 0.006	**0.040 ± 0.001**	**0.001 ± 0.000**	0.001 ± 0.000	0.066 ± 0.003	0.004 ± 0.001	0.075 ± 0.004	0.005 ± 0.000	0.020 ± 0.001
HAB	**0.066 ± 0.003**	0.033 ± 0.002	**0.001 ± 0.000**	0.001 ± 0.000	0.064 ± 0.003	0.005 ± 0.000	0.073 ± 0.006	0.005 ± 0.000	0.021 ± 0.001
MCAB	0.063 ± 0.001	**0.040 ± 0.002**	**0.001 ± 0.000**	0.002 ± 0.000	**0.067 ± 0.003**	0.006 ± 0.000	**0.076 ± 0.002**	**0.006 ± 0.000**	**0.023 ± 0.001**

TABLE V
THE AGGREGATED SCORE OF PRE-TRAINED AGENTS AND HYBRID MODELS ON SONIC TRAINING LEVELS.

Agents	Level			
	GHZ.Act1	SLZ.Act1	SYZ.Act2	MZ.Act1
Random	366	449	449	196
PR	1729	**3268**	822	1989
x-pos	2386	2538	1041	1965
contest	2085	481	820	2376
CRA	1406	1551	987	2074
HRA	792	618	822	201
AB	1484	**3203**	1108	2098
MCAB	**2742**	**3090**	**1401**	**2602**

MCAB performs better than the AB model. The higher average test score of the MCAB model indicates that the weights learned by the meta-controller lead to better generalization, confirming the benefits of using MCAB in combining different agents. Looking at the test scores on each level individually, the AB model achieves a higher test score on one test level, SLZ.Act2. By looking at the level maps and comparing their dynamics, we can see that SLZ.Act2 has significantly more loops than the other test and train levels; therefore, the weight distributions learned in the MCAB model do not work as well on SLZ.Act2. In the next session, we compare the performance of hybrid models with non-hybrid models.

H. Hybrid Model vs. Baselines

In this section, we compared the average win-rates of the hybrid models to several baseline models, i.e., CRA, BHM, and HRA, in the CCFS and Sonic environments. As shown in Table III and IV, BAZIGOOSHI outperforms all the baselines on most levels in CCFS. Based on the evaluation results on unseen levels, we conclude that BAZIGOOSHI provides a stronger generalizer than methods that consider a single agent's decision, such as CRA and BHM. Furthermore, comparing BAZIGOOSHI to the state-of-the-art hybrid model HRA, trained equally long on the levels from CCFS, we see BAZIGOOSHI performs on average better than HRA. It is worth mentioning that HRA outperforms BAZIGOOSHI on three levels (J, L, and N). Based on this observation, one might argue that an HRA model trained for a longer time or using a more significant number of decomposed sub-rewards might be able to surpass the performance of MCAB on more levels. However, in situations where computational resources are limited, training MCAB proves more straightforward than HRA. This is because pre-trained agents do not need simultaneous training; moreover, they can be effectively used in different combinations. Future works can focus on comparing these models without time and computational limitations.

On Sonic, we have CRA and HRA baselines. The performance comparison between BAZIGOOSHI and these baselines confirms the CCFS conclusions. Interestingly, the CRA model performs relatively well on Sonic and generalizes better than MCAB to some unseen levels. We attribute the success of CRA in Sonic to the setting of those levels [5]. More specifically, GHZ.Act2 and MZ.Act2 levels have less number of rings to collect compared to the other levels, which affects the sparsity of the PR reward. We believe the multi-reward model is more prone to this sparsity as it combines several reward functions, so the sparsity of one reward signal could affect the results less dramatically. Whereas in the hybrid models, having a low-performing pre-trained agent could directly affect the final decision of the model.

I. CCFS vs. Sonic

In this section, we present an analysis of the results of each game, draw comparisons between the results, and provide insight into which type of games the hybrid models might best suit. Considering the fundamental differences between CCFS and Sonic in terms of game logic, dynamics, and state and action space size, it is fair to expect a different trend in the results of the studied models. While, on average, BAZIGOOSHI shows a good performance in both games, the performance gap between the pre-trained agents and hybrid models in the CCFS is more meaningfully significant than that of Sonic. One aspect that can explain this observation is how much each game is affected by the order of actions performed in a game round.

In CCFS, performing certain actions must precede other actions for the game to result in a win. For example, before removing a blocker that covers a tile with candy, the candy

[5]Level settings can be found here: https://info.sonicretro.org

TABLE VI
THE AGGREGATED SCORE OF PRE-TRAINED AGENTS AND HYBRID MODELS ON SONIC TEST LEVELS.

Agents	Test					
	GHZ.Act2	GHZ.Act3	SLZ.Act2	SLZ.Act3	SYZ.Act1	MZ.Act2
Random	295 ± 8	256 ± 7	476 ± 2	355 ± 10	273 ± 9	36 ± 9
PR	752 ± 41	673 ± 4	660 ± 6	1490 ± 28	272 ± 1	1189 ± 22
x-pos	1280 ± 81	1127 ± 82	738 ± 18	1406 ± 13	326 ± 7	1239 ± 12
contest	885 ± 18	788 ± 26	851 ± 14	1277 ± 7	278 ± 5	1377 ± 8
CRA	**2113 ± 28**	**1198 ± 12**	623 ± 4	1147 ± 13	**446 ± 5**	**1736 ± 15**
HRA	168 ± 0.6	135 ± 0.3	**1854 ± 257**	697 ± 2	158 ± 0	901 ± 03
AB	1047 ± 287	645 ± 33	1046 ± 26	1902 ± 31	381 ± 4	1482 ± 12
MCAB	**1977 ± 29**	**1215 ± 9**	864 ± 4	**1948 ± 25**	**451 ± 4**	**1588 ± 12**

underneath is not interactable. This translates into the importance of hybrid models that provide a more flexible decision-making ability. However, the order of actions is less critical in gaining higher scores in Sonic. For example, if the agent misses a ring, it can always go back and perform a jump action. So, the hybrid models in Sonic may not provide as strong of an advantage as they do in CCFS. The hybrid models, especially MCAB, could show their full potential when applied to games where the interactions between elements are affected by the order of actions, which could significantly affect the gameplay.

V. RELATED WORK

In this section, we first review the state-of-the-art RL-based solutions for gameplay-testing and then explore the recent models for reward shaping and generalization.

A. Gameplay-testing with RL

One interesting application of RL in games is to perform automated gameplay-testing. These automated tests can be applied for different objectives, such as finding bugs in the game application environments, checking the game's feasibility given the dynamics, assessing the difficulty of the games, and building in-game Non-Playable Characters (NPC). Bergdahl et al. [17] use RL to improve gameplay-testing to find unintended video game exploits. Alonso et al. [18] train an RL agent to develop skilled NPCs to navigate the 3D environment or check the feasibility of procedurally generated goals. Woillemont et al. [19] utilize RL for gameplay-testing in games where they train an agent to emulate the players' play styles, even on previously unseen levels, with little need for human data. Moreover, Borovikov et al. [20] propose a learning and planning framework based on RL to create intelligent game agents in service of the development processes of the game developers.

B. Reward Shaping and Generalization

In order to address the issue of sparse rewards, Florensa et al. [21] introduce a framework for training a group of agents using intrinsic rewards that demand minimal domain knowledge about the tasks downstream. They additionally train a model to select the best agent from this group. Another approach that addresses the problem of sparse rewards is discussed in [22], where the authors suggest a linear method for combining smaller rewards associated with in-game features.

Hierarchical learning is a common method for enhancing the generalization of RL models. For instance, Van Seijen et al. [7] introduce a hybrid reward architecture, decomposing the reward function into various reward streams and utilizing a multi-head network to learn the value functions of each stream. Their approach effectively solves games with large state spaces that are hard to learn and generalize. Another study by Sahni et al. [23] explores the concept of recursively composing policies to generate hierarchies displaying complex behaviors. They suggest a network architecture that trains different composable policies separately and combines their learned embedding into a single embedding. The composition function demonstrates good generalization performance to unseen tasks. Additionally, Sutton et al. [24] suggest the idea of "options", which are temporally extended actions trained in parallel using intrinsic reward functions. After training a set of options, a higher-level agent, whose action space comprises these learned options, evaluates them using its own reward function.

Kulkarni et al. [25] present a hierarchical approach that consists of two stages: a meta-controller and a controller. The meta-controller takes a state as input and selects an option, which the controller then uses to choose an action based on the state and the selected option. Bacon et al. [26] add to the flexibility of options by introducing the option-critic architecture that can learn the internal policies, termination conditions, and policy over options without additional rewards. In a recent study, Barreto et al. [27] propose a framework that combines learned sequences of actions, called skills, to generate a variety of behaviors. This approach learns options associated with a set of pseudo-rewards and creates new options induced by any linear combination of these pseudo-rewards without the need for additional learning.

C. Cooperative Multi-Agent RL

BAZIGOOSHI shares ideas with some of the techniques used in cooperative multi-agent RL. In this domain, achieving centralized learning among agents has been demonstrated to be effective through the learning of a joint action-value. However, the optimal strategy for deriving decentralized policies remains ambiguous. Sunehag el al. [28] introduce Value Decompo-

sition Networks (VDN), a learned value-decomposition approach applied to individual agents. VDN offers a method for training individual agents within a multi-agent environment, employing a network architecture specifically designed for learning the decomposition of the team value function into individual agent-wise value functions. Results highlight the superiority of this value decomposition-based architecture over prior state-of-the-art methods.

Building upon the insights of VDN, Rashid et al. [29] introduce QMIX, a value-based approach capable of training decentralized policies in a centralized end-to-end manner. In QMIX, a network conditioned on local observations estimates the joint action-values of agents through a non-linear combination of each agent's values. Evaluation of QMIX across a set of tasks in the game of StarCraft II demonstrates significant improvement over existing value-based multi-agent RL methods.

Techniques that use an ensemble of various models or algorithms often provide strong generalization capabilities [30]. Perepu et al. [31] apply such a technique to time series forecasting. Their method uses RL to dynamically weigh several different (non-RL) models in a hybrid architecture and also compares RL-based and neural network-based methods of learning the weights. They report good generalization performance using this model. Similar to some of the mentioned works, our work proposes a hierarchical model encompassing multiple reward functions. However, unlike prior approaches used in RL applications, our method considers the decisions of all RL agents trained with various rewards to determine the final action. Additionally, we utilize learnable weights to combine the RL agents' decisions in a weighted manner, allowing us to emphasize the importance of each agent differently depending on the game state. We show that this flexibility leads to improved generalization on new levels.

VI. Conclusions

This paper introduces BAZIGOOSHI, an extension of CandyRL, which is a Reinforcement Learning (RL) framework for gameplay. This framework leverages a hybrid architecture for combining a set of pre-trained agents, each specializing in acquiring basic skills, to determine the optimal actions for each step in a game. BAZIGOOSHI incorporates three distinct approaches for combining these pre-trained agents: (1) Average Bagging (AB) that assigns static and equal weights to all pre-trained agents, (2) Heuristic-based Average Bagging (HAB) that employs a heuristic to assign non-uniform static weights, and (3) Meta Controller Average Bagging (MCAB) that utilizes a meta-controller RL agent to allocate weights to the pre-trained agents dynamically.

To assess the adaptability and broader applicability of BAZIGOOSHI, we selected two different types of game environments, Candy Crush Friends Saga (CCFS) and Sonic the Hedgehog Genesis (Sonic), as our experimental platforms. In evaluating the generalization capabilities of the hybrid models on unseen levels, it becomes evident that dynamic weight assignment using the meta-controller RL in MCAB yields a higher win-rate than most static weighting approaches across the games. However, results are inconclusive when comparing MCAB with the Hybrid Reward Architecture (HRA) baseline in CCFS and with the Combined Reward Agent (CRA) baseline on Sonic, as HRA outperforms MCAB on two levels in CCFS and CRA gives comparable results to MCAB on some unseen levels in Sonic. Moreover, we conducted ablation studies on both games to further explore our initial hypothesis regarding the positive influence of dynamic weights on the agents' generalizability. The results demonstrated that varying combinations of agents had distinct effects on BAZIGOOSHI's performance.

One limitation of our proposed hybrid models is the requirement for handcrafted reward functions. Future research in this domain may delve into automatic reward function discovery, thereby eliminating the need for handcrafted reward functions. One approach to achieve this is to leverage offline datasets from various sources, such as player gameplay data. Moreover, CCFS offers a vast array of diverse levels, and attaining strong performance on more intricate levels proves to be more challenging for certain agents than others. Another interesting direction for future work involves the exploration of the transferability of skills and learnings from agents trained to play more straightforward levels to train agents with better performance on more complex levels.

Acknowledgements

The authors would like to acknowledge the thesis work of Francesco Lorenzo conducted at King Ltd. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] A. Karnsund, "DQN tackling the game of candy crush friends saga: A reinforcement learning approach," Master's thesis, KTH Royal Institute of Technology, 2019.

[2] M. Fischer, "Using reinforcement learning for games with nondeterministic state transitions," Master's thesis, Linköping University, 2019.

[3] A. Stout, G. D. Konidaris, and A. G. Barto, "Intrinsically motivated reinforcement learning: A promising framework for developmental robot learning," 2005.

[4] J. Schmidhuber, "Formal theory of creativity, fun, and intrinsic motivation (1990–2010)," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 3, pp. 230–247, 2010.

[5] Y. Shin et al., "Playtesting in match 3 game using strategic plays via reinforcement learning," *IEEE Access*, vol. 8, pp. 51 593–51 600, 2020.

[6] F. Lorenzo et al., "Use all your skills, not only the most popular ones," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 682–685.

[7] H. Van Seijen et al., "Hybrid reward architecture for reinforcement learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 5392–5402.

[8] S. Karimi, S. Asadi, F. Lorenzo, and A. H. Payberah, "CandyRL: A hybrid reinforcement learning model for gameplay," in *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2022, pp. 567–572.

[9] S. Gudmundsson et al., "Human-like playtesting with deep learning," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 1–8.

[10] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in rl," *arXiv preprint arXiv:1804.03720*, 2018.

[11] ——, "Gotta learn fast: A new benchmark for generalization in rl," *arXiv preprint arXiv:1804.03720*, 2018.

[12] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[13] R. Sutton et al., "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[14] J. Schulman et al., "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[15] V. Mnih et al., "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[16] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[17] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 600–603.

[18] E. Alonso, M. Peter, D. Goumard, and J. Romoff, "Deep reinforcement learning for navigation in aaa video games," *arXiv preprint arXiv:2011.04764*, 2020.

[19] P. L. P. de Woillemont, R. Labory, and V. Corruble, "Automated play-testing through rl based human-like play-styles generation," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 18, no. 1, 2022, pp. 146–154.

[20] I. Borovikov, Y. Zhao, A. Beirami, J. Harder, J. Kolen, J. Pestrak, J. Pinto, R. Pourabolghasem, H. Chaput, M. Sardari *et al.*, "Winning isn't everything: Training agents to playtest modern games," in *AAAI Workshop on Reinforcement Learning in Games*, 2019.

[21] C. Florensa et al., "Stochastic neural networks for hierarchical reinforcement learning," *arXiv preprint arXiv:1704.03012*, 2017.

[22] G. Lample et al., "Playing fps games with deep reinforcement learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[23] H. Sahni et al., "Learning to compose skills," *arXiv preprint arXiv:1711.11289*, 2017.

[24] R. Sutton et al., "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[25] T. Kulkarni et al., "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," *Advances in neural information processing systems*, vol. 29, 2016.

[26] P. Bacon et al., "The option-critic architecture," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[27] A. Barreto et al., "The option keyboard: Combining skills in reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[28] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls *et al.*, "Value-decomposition networks for cooperative multi-agent learning," *arXiv preprint arXiv:1706.05296*, 2017.

[29] T. Rashid, M. Samvelyan, C. S. De Witt, G. Farquhar, J. Foerster, and S. Whiteson, "Monotonic value function factorisation for deep multi-agent reinforcement learning," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7234–7284, 2020.

[30] Y. Song, P. Suganthan, W. Pedrycz, J. Ou, Y. He, and Y. Chen, "Ensemble reinforcement learning: A survey," *arXiv preprint arXiv:2303.02618*, 2023.

[31] S. K. Perepu, B. S. Balaji, H. K. Tanneru, S. Kathari, and V. S. Pinnamaraju, "Reinforcement learning based dynamic weighing of ensemble models for time series forecasting," *arXiv preprint arXiv:2008.08878*, 2020.