CANDYRL: A Hybrid Reinforcement Learning Model for Gameplay

Sara Karimi *[†], Sahar Asadi[†], Francesco Lorenzo[†], Amir H. Payberah[*]
[*] KTH Royal Institute of Technology, Sweden
[†] King.com Ltd., Sweden
sara.karimi@king.com, sahar.asadi@king.com, francescolorenzo1996@gmail.com, payberah@kth.se

***Abstract*—Although Reinforcement Learning (RL) is becoming increasingly popular in gameplay, making a generalized RL model is still challenging. This paper presents CANDYRL, a generalized RL solution for match-3 games, particularly Candy Crush Friends Saga. CANDYRL rewards RL agents not only for achieving the game objectives but also for learning some intrinsic basic skills, which are not directly related to the game objectives. CANDYRL also includes a hybrid model that combines several pre-trained agents to determine the actions. We propose two approaches to determine the weights of the pre-trained agents in the hybrid model to reflect their importance by using (i) a heuristic method and (ii) an RL-based approach. We show that the hybrid model outperforms all the pre-trained agents used as its building blocks through the experiments. Moreover, the RL-based approach of learning weights in the hybrid model shows better generalization than the other approaches as it performs better on unseen new game levels.**

I. INTRODUCTION

Reinforcement Learning (RL) is a popular approach in game development that can speed up the process of play-testing and gameplay. However, developing RL models for match-3 games, such as *Candy Crush Friends Saga (CCFS)* [1], is challenging. Match-3 games, particularly CCFS, present properties such as large action space, stochastic transitions, and a large variety of in-game features and objectives that make them a challenging but interesting testbed for RL research [1]. Applying RL models using sparse reward functions in such environments faces (i) slow learning and (ii) challenges in generalization to new unseen levels.

To address the sparsity issue, Stout et al. [2] and Schmidhuber [3] propose adding a domain-specific intrinsic reward to the reward coming from the environment. To enhance the generalization of RL models, Shin et al. [4] define a set of strategies and train an RL model to return a probability value for each strategy at different game states. Inspired by this work, Lorenzo et al. [5] propose using a set of basic skills on CCFS that allow agents to learn more versatile behaviors that work across levels with different objectives. Another work on generalizing RL models is proposed by Seijen et al. [6], where they present a hybrid architecture to decompose the reward function into multiple reward functions where each function is only affected by a small number of state variables. However, these solutions either consider a fixed reward function or use multiple reward functions but do not allow a dynamic combination of their decisions. We theorize that dynamically weighting agents' decisions while combining them could lead to better generalization over different levels.

In this work, we present CANDYRL, a generalized RL model for gameplay, particularly for CCFS, that addresses the limitation mentioned above. The main idea of CANDYRL is to teach an RL agent a set of skills and combine them to enable the agent to approach new levels without starting tabula rasa. To this end, in addition to giving *extrinsic rewards* to the agent for achieving the objective of a level, we use some *intrinsic rewards* to teach the agent a set of *basic skills*. Using extrinsic reward, the agent is rewarded by the environment for getting closer to achieving the levels' objectives. While using intrinsic rewards, the agent rewards itself for achieving a more general goal that is not directly related to the level objective.

CANDYRL presents a hybrid model for combining multiple pre-trained agents (using extrinsic or intrinsic rewards) to improve the generalization by taking advantage of different skills in different game steps. The pre-trained agents in the hybrid model are weighted. To this end, we consider three different ways of assigning weights to them: (i) assigning equal weights statically to all the pre-trained agents, (ii) assigning static non-equal weights using a heuristic that reflects the importance of each pre-trained agent, and (iii) using an RL agent to learn weights for the pre-trained agents dynamically.

The main contributions of this paper are: (i) Proposing CANDYRL, a hybrid model that combines different pre-trained agents using different weighting techniques (i.e., static and dynamic) to reflect their importance. (ii) Conducting an ablation study to show how different combinations of pre-trained agents could affect the performance of the hybrid models (in terms of win-rate in CCFS). (iii) Comparing the performance of the hybrid models with each of the pre-trained agents trained using intrinsic and extrinsic rewards.

Through the experiments, we observe that the hybrid models outperform all the pre-trained agents used as the building blocks of the hybrid models. Moreover, the experiments show that our heuristic approach for assigning the weights provides the highest win-rate for training levels. However, our RL-based model that dynamically determines the weights performs better on the unseen and new levels. These results confirm that defining the weights using the RL model provides a more generalized solution for gameplay.

[1]Hidden due to anonymity

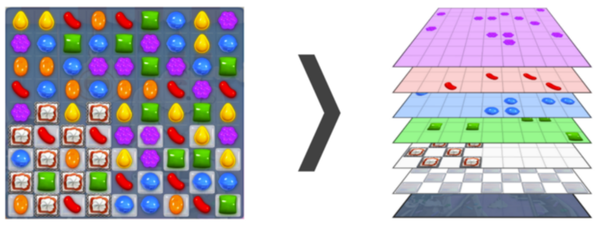


Fig. 1. An example of CCFS game board encoding.

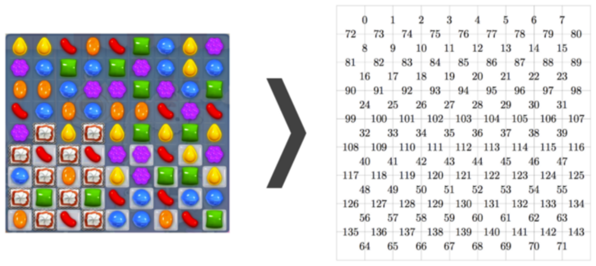


Fig. 2. Representation of possible actions in the game board.

II. Preliminary

In this section, we present the basic concepts of the game, the environment, the objectives, and the reward functions.

A. Candy Crush Friends Saga

Candy Crush Friends Saga (CCFS) is a match-3 puzzle game where a player needs to match three or more objects of the same color to advance toward a specified goal (e.g., reaching a specified score). The game board is a grid of 9×9 tiles that hold multiple types of game elements. A basic action in the game is *matching*, where swapping two candies on the board creates a horizontal or vertical sequence of three or more candies of the same color. The matched candies are then eliminated from the board and replaced with new ones above them or with random candies re-spawned from the top of the board.

There are three main types of elements in the CCFS game board: *regular candies*, *special candies*, and *blockers*. Regular candies are the most common type in the game, with seven different colors. Special candies are created by matching at least four candies of the same color. There are six different types of special candies, each with a different effect. Different special candies have different recipes for getting activated. Blockers are the third type of CCFS element that prevent players from using the tile they are located on. Each blocker has a fixed number of *layers*, and each layer is removed by making a match that involves its adjacent tiles.

B. Environment and Objectives

The CCFS *environment* is episodic, where each *episode* corresponds to a round of complete gameplay on a level. We represent the state space of the environment with a three-dimensional representation of the board, i.e., $9 \times 9 \times 32$. The first two dimensions (9×9) represent the game board grid, and the third dimension is a one-hot encoding channel of any of the 32 different elements (including different types of candies and blockers, etc.), each associated with a binary layer that describes if that element is present or not on each cell of the 9×9 grid [7]. Figure 1 depicts an example of the three-dimensional representation of the board [7].

We define an *action* as swapping the elements on any two adjacent cells of the game board. Thus, if we uniquely index the edges between the tiles as the labels of the actions, then for a 9×9 board, the *action space* consists of 144 actions, as shown in Figure 2 [7]. A *policy* is then a mapping from $9 \times 9 \times 32$ states to 144 actions.

Each level in CCFS is associated with an objective, and there are five different objectives in CCFS[2]. Players win a level if they fulfill the objective within the level-specific moves-limit. In this paper, we consider the levels with a specific objective type, called *spread the jam*, where players should cover the entire board with jam. The jam is initially present in only a few tiles and spreads by making matches involving tiles already covered by jam. In this paper, we only experiment on levels with this objective to make the presentation more precise, but we can easily apply our technique to the other objectives.

C. Reward functions

To train the RL agents, we use several *extrinsic* and *intrinsic* reward functions as explained below.

1) Extrinsic Rewards: An *extrinsic reward* is a reward defined based on the objective of the levels, and the environment gives it to the agents. In this work, to train baseline agents, we use two extrinsic reward functions proposed by Karnsund [8] for CCFS, *Progressive Jam (PJ)* and *Delta Jam (DJ)* [8]. According to PJ, if the agent's action spreads at least one tile with jam, it is rewarded with the entire amount J of tiles covered with jam at that moment, normalized by the total number B of tiles on the board (i.e., 9×9). If we show the number of new tiles covered by jam after the action by j, then we define the reward R for taking action a in a state s as:

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

On the other hand, DJ rewards the agent equal to the number of new tiles covered by the jam normalized by the total number of tiles on the board.

2) Intrinsic Reward: The above extrinsic rewards are defined for specific objectives (e.g., spread the jam), but we need to define the rewards such that the agents are generalized enough to play on various levels with different objectives. One approach to making more generalized models is to use *intrinsic rewards*, where an agent rewards itself for achieving goals that are not necessarily directly related to the objective of a level. The intrinsic rewards are inspired by the general basic skills humans gain after playing many levels of CCFS.

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

Below, we explain three intrinsic rewards for CCFS we use in this work, which are *Candy Creation (CC)*, *Candy Usage (CU)*, and *Damage Blocker (DB)* [5].

The CC function rewards the agent by the number of special candies of each type it creates after each action. However, these values are normalized due to the different frequency of occurrence of different types of special candies. The formulation of CC is as follows:

$$R(s,a) = \sum_{x \in X} c^{(x)} \times \left(1 - \frac{\mu^{(x)}}{\sum_{x' \in X} \mu^{(x')}}\right) \tag{2}$$

where $c^{(x)}$ is the number of special candies of type x created by action a, $\mu^{(x)}$ is the mean episodic frequency of creating special candy of type x. The denominator is the sum of all the frequencies of creation of all special candies.

The CC reward function leads to creating more special candies, but the ultimate goal of creating such candies is to use them to increase the chance of possible moves. Thus, the CU function rewards the agent when they use special candies in action:

$$R(s,a) = \sum_{x \in X} u^{(x)} \times (1 - \frac{\mu^{(x)}}{\sum_{x' \in X} \mu^{(x')}}) \tag{3}$$

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

The blockers in CCFS can be presented in different types, characteristics, and quantities across different levels. DB reward function rewards the agents for damaging each blocker of type b, normalized by the initial number b_0 of blockers of that type. DB is formulated as:

$$R(s,a) = \sum_{b \in B} \frac{d^{(b)}}{b_0} \tag{4}$$

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

III. Method

In the previous section, we introduced several extrinsic (e.g., PJ and DJ) and intrinsic (e.g., CC, CU, and DB) reward functions. We define a *pre-trained agent* as an RL agent trained using either an extrinsic or intrinsic reward function. Lorenzo et al. [5] show that using intrinsic rewards to train agents improves the model's winning rate performance. This paper extends that work and shows that we can make a more generalized model by combining pre-trained agents using a hybrid architecture. We call our hybrid model CandyRL. To this end, we introduce three ways of combining pre-trained agents in CandyRL: (i) *Average Bagging (AB)* that uses an un-weighted ensemble of pre-trained agents, (ii) *Heuristic-based Average Bagging (HAB)* that uses some heuristics to assign static weights to the pre-trained agents, reflecting their importance, and (iii) *Meta Controller Average Bagging (MCAB)* that uses an RL controller for assigning weights to each pre-trained agent dynamically. In the rest of this section, we explain these three hybrid approaches.

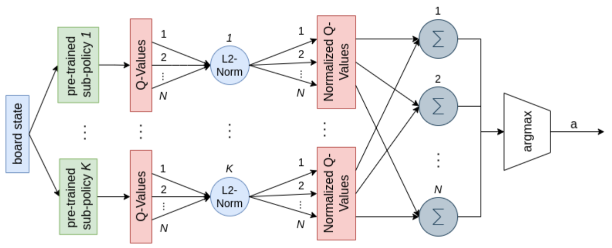


Fig. 3. Action selection in AB.

A. Average Bagging

Average Bagging (AB) is an ensemble model that combines multiple pre-trained agents to decrease variance and enhance the agent's generalization ability [9]. As Figure 3 shows, in the AB model, first, the current state of the game is fed to each pre-trained agent simultaneously, each of which returns a set of Q-values (144 Q-values for 144 possible actions). Then, we aggregate the generated Q-values for each action (by computing the average Q-value for each action across different pre-trained agents) and select an action corresponding to the highest average Q-value, as shown in (5). Here, $Q_k(s, a_n)$ is the Q-value of action a_n generated by the pre-trained agent k, and s is the input state. K is the number of pre-trained agents, and N is the number of actions, i.e., 144.

$$a = \underset{a}{\operatorname{argmax}} \left(\frac{1}{K} \sum_{k=1}^{K} Q_k(s, a_n), n \in \{1, 2, ..., N\} \right) \tag{5}$$

Since each pre-trained agent uses different reward scales, their outputted Q-values may have different ranges, adding bias when aggregating them. To combat this issue, we apply L2-normalization to bring all the Q-values in the range [0,1].

B. Heuristic-based Average Bagging

One limitation of the AB model is using the same weight (or coefficients) for all the pre-trained agents when averaging their Q-values. However, through an ablation study IV-B, we observe that some pre-trained agents are more effective than others. Therefore, we propose *Heuristic-based Average Bagging (HAB)*, which uses a heuristic to set different weights for the pre-trained agents. This heuristic takes the final training win-rate of each pre-trained agent trained for a fixed number of episodes on specific levels and returns a set of weights proportional to these win-rates values. This model follows the same procedure as in AB, but instead of un-weighted aggregation, it uses the calculated weights to perform a weighted aggregation over the Q-values of different pre-trained agents. Mathematically, this weighted aggregation is formulated in (6):

$$a = \underset{a}{\operatorname{argmax}} \left(\frac{1}{K} \sum_{k=1}^{K} w_k Q_k(s, a_n), n \in \{1, 2, ..., N\} \right) \tag{6}$$

where $w_k \propto$ {training win-rate of k-th agent} (range [0,1]).

Intuitively, this heuristic emphasizes the importance of each pre-trained agent in the hybrid model on a specific level.

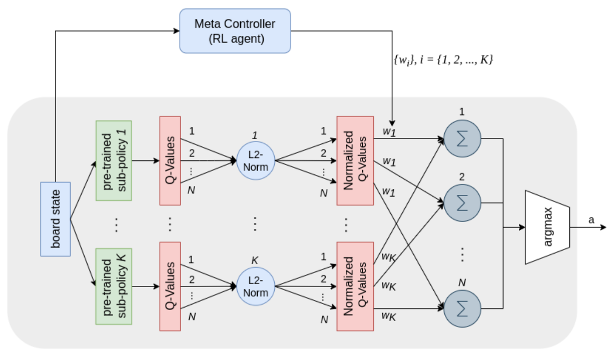


Fig. 4. Action selection in MCAB.

C. Meta Controller Average Bagging

In HAB, we assign fixed static weights to the pre-trained agents. However, different actions may have different impacts in different game steps; thus, we adjust the weights dynamically for the game's current state in the third model. To this end, we propose *Meta Controller Average Bagging (MCAB)*, where we use an RL agent, called the *meta-controller agent*, for learning a set of dynamic weights for combining the pre-trained agents. In MCAB, at each game step (consisting of choosing and performing an action in the environment), the meta-controller agent takes the game's current state as input and returns a set of weights for the pre-trained agents in the hybrid model. The weights learned by the meta-controller reflect the importance of their corresponding pre-trained agent. Figure 4 depicts the MCAB architecture.

To this end, we need an algorithm to train the meta-controller agent to generate continuous values as the pre-trained agents' weights. Policy gradient methods, such as REINFORCE [10] and Proximal Policy Optimization (PPO) [11], are good candidates when it comes to continuous action spaces. In the policy gradient methods, the meta-controller agent learns the policy in the form of a distribution over its action space. Here, we use PPO on continuous action space for training the meta-controller agent. The meta-controller agent first learns the parameters of a multivariate distribution, i.e., Dirichlet distribution. After creating such a distribution using these parameters, the meta-controller agent draws a sample of size K (where K is the number of pre-trained agents) as the weights of pre-trained agents for computing a linear combination over their Q-values. The MCAB model, then, chooses an action according to (6) where the weights (w_k) are samples taken from the Dirichlet distribution. The probability density function of Dirichlet distribution is as follows:

$$f(w_1, \dots, w_K; \alpha_1, \dots, \alpha_k) = \frac{1}{B(\alpha)} \prod_{i=1}^{K} w_i^{\alpha_i - 1} \tag{7}$$

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

IV. Evaluation

In this section, we first describe the experimental settings and then explain the conducted experiments and analyze the results.

A. Experimental Settings

All models are trained for 80,000 episodes, and each episode is one round of gameplay, resulting in winning or losing that level. We evaluate each model using five random seeds different from those used during training, exposing the agent to a new board initialization. For performance metric, we use *win-rate*, the ratio of the number of episodes ending in a win state over all the played episodes (ranging between [0, 1]).

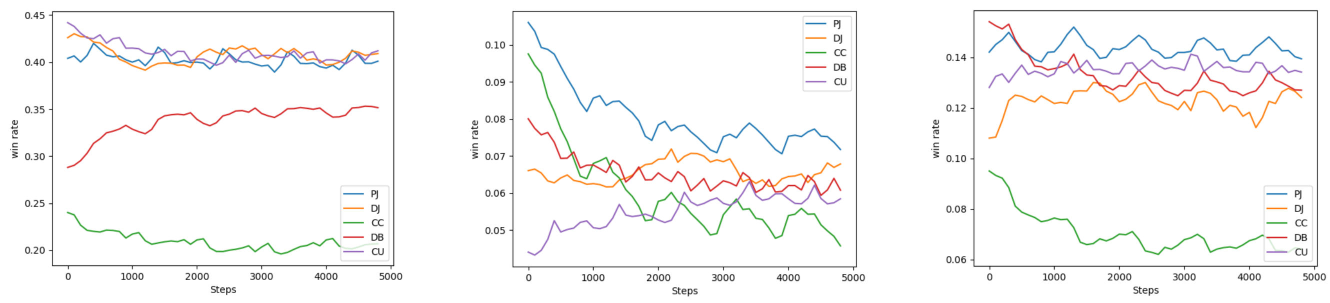
We select three levels for training the models (levels A, B, and C) and six new unseen levels for testing (levels D, E, F, G, H, and I). We select these levels such that to include various difficulties and board features. We also select the test levels with game elements and blockers different from those in the training levels. Each win-rate reported for a test level in Table I is an average of the evaluation win-rate of the three trained models on the test level. All the evaluation experiments are performed over 1000 episodes.

We use Deep Q-Networks (DQN) [12] for training all the pre-trained agents and PPO for training the meta-controller agent in the MCAB model. The DQN hyperparameters are mostly taken from the DQN paper [12]. However, we adapt some DQN hyperparameters based on [8] that performs a hyperparameter search on the CCFS environment. We set the discount factor to 0.5 and the network update step to 100. We also take most PPO hyperparameters from the original PPO paper [11] but change the horizon to size 128.

In the following sub-sections, we summarize the results of the experiments consisting of three parts: (i) an ablation experiment to study the effects of using different combinations of pre-trained agents in the hybrid model, (ii) a comparison of different weighting methods (i.e., AB, HAB, and MCAB) in the hybrid model, and (iii) comparison of hybrid models with the baselines including agent trained with sparse reward and agents trained with intrinsic or extrinsic rewards.

B. Ablation Study

Here, we present the results of an ablation experiment to show the effects of using different combinations of pre-trained agents (i.e., PJ, DJ, CC, CU, and DB) in the AB hybrid model. This experiment aims to highlight the need for weights as different agents have different levels of influence on the results. In Figure 5, each plot line presents the win-rate of the AB model that excludes the mentioned pre-trained agent from the combination. For example, "DB" means that the agent pre-trained with the DB reward function is excluded from the combination, and only the decisions from the rest of the available pre-trained agents are combined when performing AB. As Figure 5 shows, excluding different pre-trained agents from the combinations has a different influence on the overall win-rate. For example, removing CC causes a drop in win-rate across all three levels, while removing DB affects each level



(a) Level A (b) Level B (c) Level C

Fig. 5. The effect of using different combinations of pre-trained agents in the AB model on win-rates on levels A, B, and C.

differently. The results of this ablation study inspired the idea of non-equal and learnable weights in the hybrid models.

C. Comparison of Hybrid Models

As explained in Section III, we consider three variations of hybrid models (i.e., AB, HAB, MCAB) to combine the pre-trained agents. This experiment considers four pre-trained agents with PJ, CC, CU, and DB reward functions in the hybrid model. Due to the similarity between DJ and PJ and better generalization of PJ, we only keep PJ in the combinations. As reported in Table I, to assess the generalization of the models, we evaluate their performance in terms of average win-rate on both the training and new unseen test levels.

Table I shows that the HAB model outperforms the other hybrid models on the training levels (i.e., A, B, and C). This confirms that the weights used in the HAB model better emphasize the importance of each pre-trained agent on the seen levels compared to the learned weights in the MCAB model. However, looking at the generalization abilities of agents over the unseen levels (i.e., D, E, F, G, H, and I), the MCAB model shows better performance on most levels. The better generalization performance of the MCAB model compared to the other hybrid models indicates that the MCAB model learns a more generalized set of weights, meaning that they correspond more accurately to the general importance of each pre-trained agent across different levels. Although the HAB model performs best on the seen levels, it does not reach the generalization performance of the MCAB model.

D. Hybrid Model vs. Pre-Trained Agents

Here, we compare the average win-rate of the hybrid models with some baselines, including agents trained using (i) intrinsic skill-based rewards (consisting of CC, CU, and DB), (ii) extrinsic objective-based rewards (consisting of PJ and DJ), and (iii) the sparse reward (+1/-1 for winning/losing the level). As we see in Table I, all the proposed hybrid models outperform the baseline models in training performance. These results confirm that the hybrid architectures allow the agent to make a more informed decision by considering the decisions of multiple pre-trained agents. Interestingly, the performance gap between the hybrid and non-hybrid models increases on more challenging levels (e.g., G and I), highlighting the benefits of hybrid architectures in solving more complex levels. Also, considering the generalization performance on the unseen levels in Table I, the MCAB hybrid model outperforms all the baselines in terms of win-rate when evaluated on the test levels.

V. Related work

To tackle the sparsity of rewards, Florensa et al. [13] propose a framework to train a set of agents using intrinsic rewards that only require minimal domain knowledge about the downstream tasks. On top of them, they train a model to select an agent among the set of agents. Another work that addresses the sparsity of rewards is [14], where the authors propose a linear method for combining smaller intermediate rewards corresponding to in-game features.

A common approach to improving the generalization of RL models is hierarchical learning. For example, Van Seijen et al. [6] propose a hybrid reward architecture that decomposes a reward function into multiple different reward functions and uses a multi-head network to learn the value functions of each reward stream. They show the effectiveness of their approach in solving games with large state spaces that are hard to learn and generalize across. Sahni et al. [15] study the idea of recursively composing policies to create hierarchies that display complex behaviors. They propose a network architecture that trains various composable policies in isolation and composes their learned embedding into a single embedding. The composition function shows a good generalization performance to unseen tasks. Sutton et al. [16] propose the idea of “options”, which are temporally-extended actions trained in parallel based on intrinsic reward functions. After training a set of options, a higher-level agent whose action space consists of these learned options, evaluates them using its own reward function.

Kulkarni et al. [17] propose a two-stage hierarchy consisting of a controller and a meta-controller. The meta-controller receives a state and chooses an option, and then the controller selects an action using the state and the selected option. To further increase the flexibility of options, Bacon et al. [18] propose the option-critic architecture capable of learning the

TABLE I

AGGREGATED TRAINING WIN-RATES. SPARSE REWARD, OBJECTIVE-BASE, AND SKILL-BASED AGENTS ARE COMPARED WITH THE HYBRID MODELS.

Agents	**Train**			**Test**					
	A	B	C	D	E	F	G	H	I
Sparse	0.066	0.008	0.008	0.003	0.003	0.0	0.0	0.011	0.0
DJ	0.136	0.010	0.030	0.013	0.008	0.0	0.0	0.021	0.001
PJ	0.098	0.029	0.024	0.017	0.012	0.0	0.0	0.025	0.002
DB	0.212	0.059	0.041	0.016	0.011	0.0	0.0	0.019	0.0
CU	0.085	0.038	0.039	0.023	0.022	0.001	0.001	0.041	0.003
CC	0.331	0.014	0.053	0.021	0.009	0.0	0.0	0.038	0.0
AB	0.400	0.063	0.135	0.059	**0.044**	0.001	0.001	0.061	0.004
MCAB	0.379	0.060	0.129	**0.065**	0.039	**0.002**	**0.002**	**0.065**	**0.005**
HAB	**0.533**	**0.066**	**0.159**	**0.065**	0.039	0.0	0.001	0.061	**0.005**

internal policies, the termination conditions of options, and the policy over options without the need to provide any additional rewards. In a more recent paper, Barreto et al. [19] propose a framework for combining learned sequences of actions (a.k.a skills) to generate many distinct behaviors. It does so by learning options associated with a set of pseudo-rewards and generating new options induced by any linear combination of these pseudo-rewards without any learning involved.

In our work, we also make a hierarchical model of multiple different reward functions; however, unlike the above models, our approach accounts for the decision of all agents trained with different reward functions to choose the final action. Moreover, we perform a weighted combination of the agents' decisions using learnable weights that give more freedom in highlighting the importance of each agent at different steps of play. We theorize that this freedom could lead to better generalization over unseen levels.

VI. CONCLUSIONS

In this work, we introduced CANDYRL, a generalized RL model for match-3 games. This work is inspired by how human players learn basic skills and combine them to play different game levels. These skills do not necessarily correspond explicitly to the objective of a level, but they still help players achieve the game objective. After training a set of RL agents based on these basic skills and the game objectives, CANDYRL combines the pre-trained agents in a hybrid architecture to pick the best action at each game step. Since the pre-trained agents have different impacts, we proposed two main weighting approaches to combine them, a heuristic method and an RL-based one. In the experiments, we observed that the heuristic approach of defining weights showed a better win-rate on levels that the agents were trained on. However, when evaluating the generalizability of CANDYRL on unseen levels, the RL-based weighting resulted in a higher win-rate, highlighting the benefits of having learned weights.

ACKNOWLEDGEMENTS

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] I. Kamaldinov et al., "Deep reinforcement learning in match-3 game," in *2019 IEEE conference on games (CoG)*. IEEE, 2019, pp. 1–4.

[2] A. Stout et al., "Intrinsically motivated reinforcement learning: A promising framework for developmental robot learning," Massachusetts Univ Amherst Dept of Computer Science, Tech. Rep., 2005.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

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

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

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

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

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

[16] 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.

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

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

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