Generalized Reinforcement Learning for Gameplay

Francesco V. Lorenzo[1,2], **Sahar Asadi**[1], **Alice Karnsund**[1], **Tianze Wang**[2], **Amir H. Payberah**[2]
[1] King Digital Entertainment, Sweden
[2] KTH Royal Institute of Technology, Sweden
fvlo@kth.se, sahar.asadi@king.com, alice.karnsund@king.com, tianzew@kth.se, payberah@kth.se

Abstract

Reinforcement Learning (RL) is becoming ever more prevalent in game development. However, there exist many challenges to overcome in order to use RL to its full potential. For example, an RL agent trained in one game environment cannot easily generalize to replicate the same level of performance in new game environments with different levels and variations. This paper presents a generalized solution for match-3 games, such as Candy Crush Friends Saga (CCFS). Our solution is a two-step process, inspired by human behavior in approaching such games: (i) learning basic skills while progressing through the levels, and (ii) combining and reusing them in different game states. These skills are not necessarily related to a level's objective, but their execution can indirectly help the player win the level. In this paper, we propose various basic skills for CCFS with *intrinsic rewards*, which do not necessarily have the same objective as the game level. We show that an agent trained with intrinsic rewards outperforms the agents that are trained with extrinsic rewards, despite not knowing how to win a level. Moreover, we show that by making a hybrid model and combining these basic skills, the agent can significantly outperform the baselines, winning more than twice as much as an agent trained with extrinsic rewards.

Introduction

The use of Reinforcement Learning (RL) in game development is becoming more prevalent. However, developing RL agents for *match-3 games*, such as Candy Crush Friends Saga (CCFS)[1] is challenging due to the sparsity of rewards and generalization problem. The former makes the learning process slower, as showed in popular benchmarks of the Arcade Learning Environment (ALE) (Bellemare et al. 2013), such as Montezuma's Revenge[2], and the latter prevents an RL agent trained on one environment to obtain the same level of performance in new environments with different levels and variations.

One way to tackle the sparsity reward problem is to reward agents progressively as they get closer to achieving the objective of a level (Karnsund 2019; Fischer 2019). However, this approach can lead to poor performance, where, in order to win, agents need to prioritize some other sub-goals before focusing on completing the objective of a level. To address the challenge of generalization, Shin et al. (Shin et al. 2020) propose to teach agents more versatile behaviors that work across levels. To this end, they use common salient play-styles recognized from human play, in which a set of *skills* is utilized to reach the goal. Nevertheless, they manually define skills through heuristics, introducing human bias, and limiting the capabilities of agents.

To overcome the sparsity of rewards and generalization problems, we propose a solution inspired by human players' behavior: new players that progress through a match-3 game (e.g., CCFS) usually learn *basic skills* (e.g., creating special candies) that are not necessarily related to the particular objective of the level they are playing. They then combine and use these skills to complete tasks that can help them win new levels. Following this observation, we introduce a two-step solution:

1. First, we teach an agent a set of basic skills that enables it to approach new levels without starting tabula rasa. To this end, instead of rewarding the agent for achieving the objective of a level (*extrinsic reward*), we teach them the basic skills by relying on the concept of *intrinsic motivation*, where an agent rewards itself for achieving goals that are not directly related to the objective of a level (*intrinsic reward)*. Singh et al. (Chentanez et al. 2004) and J. Schmidhuber (Schmidhuber 2010) study the intrinsic rewards in more details, while Zheng et al. (Zheng et al. 2020) propose a framework for learning intrinsic reward functions across multiple lifetimes of experience.
2. Then, we combine these skills by proposing a hybrid architecture, called *Average Bagging (AB)*, that allows the agent to select the most appropriate behavior according to the board configuration at hand.

The contributions of this work are twofold: (i) we show that intrinsic motivation represents a successful technique to design reward functions to teach an agent a set of basic skills, and (ii) we present that the AB hybrid architecture enables an agent to combine the basic skills to improve the overall win-rate effectively. Through the experiments, we discover a strong correlation between learning the basic skills and the overall win rate. An agent trained with intrinsic rewards outperforms the baselines trained with extrinsic rewards, despite not knowing how to win a level. More-

[1]https://king.com/game/candycrushfriends

[2]https://en.wikipedia.org/wiki/Montezuma's_Revenge_(video_game)

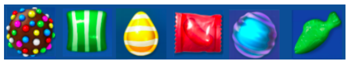


Figure 1: Special candies.

Figure 2: Blockers.

over, we observe that the AB architecture significantly outperforms the baselines, winning more than twice as much as an agent trained with extrinsic rewards, and even surpasses human performance on one of the test levels.

Preliminaries

In this section, we briefly present some basic concepts in *Candy Crush Friends Saga (CCFS)* and describe the objectives and environment of the game. CCFS is a match-3 game where the game board is a grid of 9×9 tiles that contain *Regular candies*, *Special candies*, or *Blockers*. A basic action in the game, called *match*, consists of swapping two candies on the board to create 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 if the action involves the top row.

Candies and Blockers Regular candies are the most common type of candies that are present in the game. There are seven types of Regular candies, each with a distinct color, and only candies of the same colors can be matched together. Special candies are created by matching at least four candies of the same color, Regular or Special, in a sequence. There are six different types of Special candies, each with a different effect (Figure 1)[3]. A Special candy can be used by swapping it with another candy without necessarily matching three or more candies of the same color together. Blockers (Figure 2) prevent a player from using the tile on which they are located. Each Blocker has a fixed number of *layers*, and each layer is removed by making a match that involves neighboring tiles. When the last layer is removed, the Blocker is removed from the board, and the tile becomes free.

Objective Each level in CCFS is associated with an objective. Players win a level if they reach its objective within the level specific move limits. There are five different objectives[4] in CCFS. In this paper, we consider *spread the jam* as the objective of the game, which requires players to cover the entire board with jam. The jam is initially present in only a few tiles and is spread by making matches involving tiles that already covered by jam.

Environment The *environment* in our study is episodic, where each *episode* corresponds to a full gameplay on a level. The *state space* of the environment consists of a three-dimensional representation of the board (i.e., $9 \times 9 \times 32$). The first two represent the game board grid, and the third dimension is a one-hot encoding of any of the 32 different elements (e.g., Candies, Blockers, etc.), each associated to a binary layer (Gudmundsson et al. 2018). We define an *action* as swapping any two cells on 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 (Gudmundsson et al. 2018). An agent *policy* is then a mapping from $9 \times 9 \times 32$ states to 144 actions. We use DQN (Mnih et al. 2013) as the learning algorithm to implement the policies, but any other RL algorithm could bee used to implement our solution.

As the *baseline reward*, we consider *Progressive Jam (PJ)* (Karnsund 2019), which is an *extrinsic reward*, i.e., it is given by the environment to agents. Whenever an agent makes a move that spreads at least one more 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). The reward R for taking an action a_t in a state $\mathbf{s}_t$ at time t is defined as:

$$R(\mathbf{s_t}, a_t) = \begin{cases} \frac{J}{B}, & \delta j > 0 \\ 0, & \delta j = 0 \end{cases} \tag{1}$$

where δj denotes the number of new tiles covered by jam after the action.

Solution

In this section, we present the two-step process of training an agent that includes (i) learning the basic skills, and (ii) combining these skills to make a hybrid architecture.

Basic Skill Learning

We define three basics skills: *creating Special candies*, *using Special candies*, and *removing Blockers*. Special candies allow spreading the jam more quickly since they affect multiple candies at the same time. On the other hand, Blockers prevent a player from performing matches on the board, so the jam cannot be spread on the tiles with Blockers. Therefore, to win a level with spread the jam objective, all Blockers must be taken out of the board. However, these skills are not directly related to achieving the goal of the level (i.e., covering the entire board with jam), so it is challenging to learn them by giving rewards for spreading jam. Nonetheless, these skills enable the player to complete most levels.

The first part of our solution deals with exploiting intrinsic motivation, in the form of rewards independent of the level objective, to find a set of functions that can drive an agent towards learning these skills. In particular, we focus on creating Special candies, using them, and getting rid of Blockers. We select a set of functions to learn policies that can retain good skills proficiency in unseen levels.

Creating Special Candies Special candies can be created multiple times when playing a level; however, it is not possible to predict beforehand how many Special candies can be created on a level. Consequently, normalizing the reward

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

to keep it constrained in a fixed range is not possible, and therefore techniques like *reward clipping* (Mnih et al. 2015) cannot be used. Moreover, some Special candies can be created more often than others, so rewarding an agent when it creates or uses a Special candy is not possible, as it will learn to exploit the more frequent ones, which in turn have a weaker effect.

An effective way to deal with features that have different frequencies is to use the frequency itself to balance the weight given to each feature and constrain the reward scale. In (Lorenzo et al. 2020), the authors test two types of frequency normalization for creating Special candies in CCFS: *Rarity of Events (RoE)* and *Balanced Rarity of Events (BRoE)*, where RoE is adapted from a proven method proposed in (Justesen et al. 2018), and BRoE improves RoE by normalizing RoE (Lorenzo et al. 2020). Given the promising results of BRoE, we adopt it as the candidate to learn how to create Special candies.

BRoE rewards an agent if it explores new parts of the environment, giving smaller rewards to skills that have already been observed. Normalization is performed by taking into account the proportion of an event's occurrence to all the others, such that the weights given to each one will always be in the range $[0, 1]$. The function is defined as follows:

$$r(\mathbf{s_t}, a_t) = \sum_{x \in X} c_t^{(x)} \times \left[1 - \frac{\mu_t^{(x)}}{\sum_{x' \in X} \mu_t^{(x')}}\right] \tag{2}$$

where x is the skill (i.e., creating a specific Special candy), $\mu_t^{(x)}$ is the mean episodic frequency of skill x at episode t, $c_t^{(x)}$ is the number of Special candies of type x created by a_t, and the denominator of the second term (weight) is the sum of all the frequencies of creation of all the Special candies. This method does not reward agents for winning a level or spreading jam, but it does so for using novel skills. We refer to (Lorenzo et al. 2020) for more details on the benefits of this method as opposed to RoE and the experimental results on the environment.

Using Special Candies The difference between using and creating Special candies is that the reward is given when a Special candy is involved in a move on the board, rather than when the player creates it. When combined with the previous skill, an agent will learn how to create Special candies and make proper use of them by understanding their effects. We define the *Candy Usage (CU)* reward, which is adapted from BRoE to use Special candies. We skip the usage reward akin to RoE after the preliminary tests highlighted the same findings of (Lorenzo et al. 2020). In formulas:

$$r(\mathbf{s_t}, a_t) = \sum_{x \in X} u_t^{(x)} \times \left(1 - \frac{\mu_t^{(x)}}{\sum_{x' \in X} \mu_t^{(x')}}\right) \tag{3}$$

where $u_t^{(x)}$ is the number of Special candies of type x used in the action a_t at time t. The same benefits highlighted for candy creation are also valid here.

Removing Blockers There are different types of Blockers in jam levels, each one with its characteristics. Defining a reward function that can work across multiple levels and generalize to new ones is not trivial. We propose two approaches to resolve it: *damaging Blockers* and *freeing tiles*. In both cases, the agent will learn the mechanics behind how Blockers are stripped of layers and remove them entirely from the board.

In damaging Blockers, we reward an agent whenever any Blocker loses a layer without distinguishing between different Blockers types. To this end, we define the *Damaging Blockers (DB)* reward function that rewards an agent for each Blocker x it damages, normalized by the initial number x_0 of Blockers of that type. In formulas:

$$r(\mathbf{s_t}, a_t) = \sum_{x \in X} \frac{d_t^{(x)}}{x_0} \tag{4}$$

where X is the set of all Blockers and $d_t^{(x)}$ is the number of Blockers of type x damaged at time t. For each Blocker type, the agent will accumulate a reward that adds up to 1 if it completely eliminates it. However, the total reward accumulated in an episode is equal to the number of different Blocker types at that level.

In freeing tiles, we define *Progressive Tiles (PT)* reward function that rewards an agent when it completely removes all the Blockers from a tile, making it free. Compared to the DB reward function, this one is more sparse because it only rewards when the last layer is removed. However, it introduces less bias as the agent is only rewarded for reaching the end goal, which is freeing up the board from Blockers. The PT function rewards an agent more when it gets closer to completely removing all Blockers, so it will be driven to get rid of the few remaining ones. This is done by rewarding the agent with the total number of free tiles F when a new one is freed every time. In formulas:

$$r(\mathbf{s_t}, a_t) = \begin{cases} \frac{F}{B}, & \delta f > 0 \\ 0, & \delta f = 0 \end{cases} \tag{5}$$

where δf is the number of new free tiles. The function is normalized by the total board size B, so the immediate reward for an action is at most 1, while the accumulated reward throughout an episode is unbounded.

Hybrid Architecture

Once a set of basic skills is available in the form of different policies, the question is how to put them together to improve the general win-rate of an agent. To answer it, we propose *Average Bagging (AB)*, an ensemble model to use multiple policies together to decrease variance (Breiman 1996). As explained in the previous section, there are 144 actions in the environment that can be selected by the policies. The *bagging* is performed by selecting an action with the highest average Q-value (that shows how good an action is) among all the policies. An overview of the architecture is highlighted in Figure 3 (The normalization and summation used in the figure are explained in the following sections). We propose two extensions to improve the performances: *normalization* and *summation*.

Normalization When dealing with different reward scales, working directly with Q-values can be problematic, as a policy trained with higher rewards might overshadow the others. Consequently, the effect of averaging

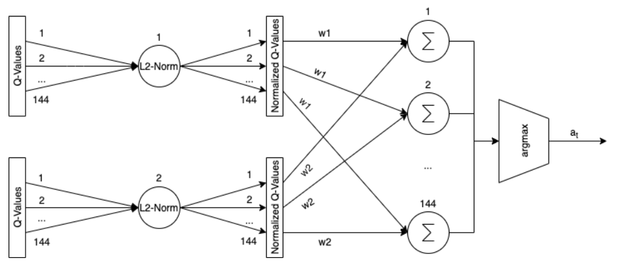


Figure 3: Action selection in AB. Two policies are displayed in the example. The first step is normalization of the Q-values for each policy. Then, the Q-value of the first policy is summed to the corresponding one of the second policy. Finally, the argmax operation is performed to select the action a_t with the highest sum.

the Q-values and selecting an action would be the same as just selecting an action directly from that policy. To address the issue, we propose to use L2-normalization to bring all the Q-values in the range $[0, 1]$. Given a vector $q = [q_1, q_2, \dots, q_{144}]$, representing the Q-values of a given policy, the output of this normalization step is a vector $y = [y_1, y_2, \dots, y_{144}]$ of the same dimension:

$$y_n = \frac{q_n}{\sqrt{q^T \cdot q}} \tag{6}$$

Summation The actual implementation of the architecture uses a trick to speed-up training, inspired by the work of (Seijen et al. 2017). Instead of performing the average over the Q-values of different policies, we sum their Q-values and then pick the action associated with the one with the highest total value. The result obtained is the same, thanks to a simple mathematical equivalence. The average is scaled by the number of points on which it is performed, which is equal to the number of different policies, denoted by m. All the 144 components of the final vector are scaled by m, which does not change the maximum value index selected by the argmax operation. However, summing instead of averaging is computationally faster in practice, so it is the preferred option. As a result, the n-th component t_n of the resulting vector $t = [t_1, t_2, \dots, t_{144}]$ is computed as follows:

$$t_n = \sum_{i=1}^{m} y_n^{(i)} \tag{7}$$

where $y_n^{(i)}$ is the normalized Q-value of action n taken by policy i, and the action selected remains the same, that is is:

$$a_n = \operatorname*{argmax}_n t_n \tag{8}$$

Evaluation

We follow the best practices from the literature to evaluate our models (Cobbe et al. 2018). We compare our models with three different baselines in the experiments:

1. The agent plays randomly, with no learning involved.
2. The agent is trained using the PJ extrinsic reward function, and represents the performances based on the level objective (i.e., covering the board with jam)
3. The available human data on those levels, from which we infer the human win rate.

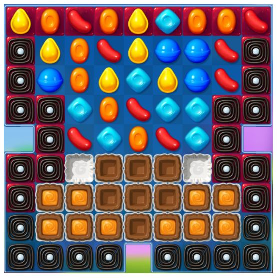
In the basic skill learning experiments, we use only the first two baselines, but we consider all three baseline models in the hybrid architecture experiments. We define *trial* as a single run of a model (i.e., an agent's policy). Given the high stochasticity in the game and the randomness involved when running a model, we perform multiple trials to account for the results' variance. We train each model for 80 000 *episodes*, where each episode is one full play on a level, either finishing with a win or a loss. For each model, we run five parallel trials with different initialized values; therefore, we have five trained versions of the same agent at the end of the training.

We test a trained model's performance both on the levels used during training and the test set. Every inference run consists of running the trained model for 10 000 episodes. Every episode is associated with a unique game seed, different from the ones used during training. Consequently, when the model is tested on the same levels of the training set, it will never face the same initial board configuration and random behavior seen during training. We perform one inference run for each trial of a given model, so we end up with five results for the same agent at the end of testing. These final results are averaged to decrease the variance encountered during training, and we report the aggregated values in this section. We always measure the inference performance through zero-shot evaluation, meaning that we leave all models' weights untouched after training and avoid fine-tuning them on the test levels.

The DQN hyperparameters we used in our implementation are the same among all the experiments, and they are mostly taken from the DQN paper (Mnih et al. 2015). Other hyperparameters, such as the discount factor, the target network update, and the prediction network update, are instead adopted from (Karnsund 2019) that performs hyperparameter search on the CCFS environment.

Special Candies We use three metrics to compare rewards functions to create and use Special candies: (i) the *creation probability* that measures the probability that an agent's action creates a Special candy of a given type, (ii) the *usage probability* that measures the probability that an agent's action uses at least one Special candy of a given type, and (iii) the *match-3 probability* that measures the probability that an agent's action does not create any Special candies, which is something that we want to minimize. These measures are averaged over the last 100 episodes through a running mean. We also measure the agent's *win rate* to understand the relationship between the considered skills and the ability to win the levels. We scale the win rate value for legal reasons, and the same applies to all win rates reported in the results.

For the policies trained to create and use Special candies, we focus on different levels with different possibilities of making them. In particular, we use levels 61, 82, and 151 in the training set, and levels 62, 147, and 163 in the test set (Figure 4). We train the two reward functions (i.e., BRoE and CU) on each level from the training set, separately. Then, we test them on both the same levels of the training set (using


(a)

(b)

(c)

(d)

Figure 4: (a) and (b) are the training levels 82 and 151, respectively, and (c) and (d) are the testing levels 147 and 163, respectively.

Table 1: Aggregated results in the training and test levels. Random policy and PJ baselines are included.

Reward	**Win Rate**		**Match-3 (%)**		**Usage (%)**		**Creation (%)**	
	Train	Test	Train	Test	Train	Test	Train	Test
Random	4.03	1.77	90.17	89.30	7.03	7.32	1.93	2.10
PJ	**7.56**	3.20	84.97	83.16	8.29	8.80	1.71	1.90
CU	6.54	3.44	81.30	79.89	10.40	10.83	1.70	1.90
BRoE	7.54	**4.03**	**48.24**	**62.89**	**11.93**	**10.91**	**9.06**	**6.71**

different initial seeds) and on the test set. Table 1 shows the aggregated results for all the Special candies, both on the training and test set.

The CU function to use Special candies has a higher usage probability than the baselines, and the BRoE function to create Special candies has a higher creation probability than the baselines, considering both the training and test levels. In particular, the latter also shows a lower match-3 percentage than all the other models. Overall, BRoE qualifies as the best model under all performance metrics, creating up to three times as many Special candies as the other models, using more of them, and even having a higher win rate than PJ on the test levels. The most interesting result is the match-3 percentage of this model, which gets as low as 48% on the training levels. This means that, on average, an action every two creates a Special candy. Moreover, compared to an agent trained using PJ extrinsic rewards with the win rate of 3.2, the win rate on the test levels is improved to 4.03.

Blockers The primary metric we measure here is the *clearing percentage* that represents the percentage of removed Blockers in one episode compared to the initial number of available ones for each type of Blocker on a given level. We consider the average of this metric over the last 100 episodes. Although our removal blockers policies (i.e., DB and PT reward functions) are defined independent of the goal of the given levels, we also track the win rate that is the ratio of the total number of wins during the inference run and the total number of episodes played, which is 10 000.

We select the training levels such that they cover all the existing types of Blockers for jam spreading levels, while their board structures differ from one another. In particular, we consider three levels 65, 82, and 103 in the training set, and three different levels 136, 147, and 163 in the test data set (Figure 4). The DB and PT reward functions are trained on each level from the training set, separately. For testing, we first test the trained agents on the same training set levels, using different seeds, and then we test them on the levels from the test set. We aggregate the results of the five trials for each agent. Table 2 shows the win rate and the clearing percentage of all Blockers together.

Both DB and PT reward functions exhibit a higher clearing percentage than the baselines on the training and test levels. This confirms that the reward functions can teach an agent the desired skill. We also notice that PJ extrinsic reward function fails on more challenging levels due to its inability to deal with features like Blockers. An agent trained with the PT reward function, which is never rewarded for spreading jam or winning a level, achieves a win rate about twice as high as the one of PJ, it manages to improve the average win rate from 0.35 to 0.5 on the test levels and from 1.14 to 2.76 on the training levels. This is an interesting finding that shows the correlation between skills and game objectives. Moreover, it indicates that this skill is an essential component to use in hybrid architectures.

Hybrid Architecture The overall purpose of making the AB hybrid architecture is to win the game; thus, we measure the win rate. To build the AB model, we do not train the whole model on different levels; instead, we combine the basic skills of agents trained on level 82 to access both skills related to Blockers and Special candies. In particular, we first concentrate on level 82 to find two combinations that work well enough, compared to the baselines. We then test these candidate models on all the other levels where the sub-policies (the basic skills) have not been trained and as-

Table 2: Aggregated results in the training and test levels. Random policy and PJ baselines are included.

Reward	Win rate		Clearing (%)	
	Train	Test	Train	Test
Random	0.21	0.05	54.81	63.63
PJ	1.14	0.35	65.08	71.18
PT	**2.76**	**0.50**	**78.92**	**73.29**
DB	1.95	0.34	75.82	71.35

Table 3: Win rate of the two best performing combinations of AB. The sub-policies are trained on level 82. Results are grouped by whether L2-normalization was used (**Left**) or not (**Right**).

Level	Combination	Win Rate	
		L2	None
82	PJ+PT+BRoE	7.02	7.40
	PJ+PT+DB+BRoE	7.33	**8.08**
62	PJ+BRoE	16.44	16.25
	PJ+BRoE+CU	**17.37**	15.83
136	PJ+PT+BRoE	3.84	4.12
	PJ+PT+DB+BRoE	3.51	**4.41**
147	PJ+PT+BRoE	2.39	2.65
	PJ+PT+DB+BRoE	2.45	**3.01**
163	PJ+PT+BRoE	0.1	0.12
	PJ+PT+DB+BRoE	0.09	**0.14**

sess the architecture's overall generalization capability. We average the results over the five training trials and report the performance of the two best performing combinations in Table 3. Table 4 presents the win rate of the best AB combination on each level, compared to the performance of all the baselines.

The majority of AB combinations, including those using fewer sub-policies, always have a higher win rate than both the agent trained with PJ extrinsic rewards and the one playing randomly. Moreover, the best models also have a significantly higher win rate than the best performing sub-policy used in the combination, meaning that the bagging technique gives the expected improvements. The effect of normalization seems to be strictly dependant on the skills used in the combination. For instance, including CU works better with L2-normalization, whereas models that do not use that skill perform better without normalization. That can be because CU overshadows the other sub-policies that have a better win rate due to a higher reward scale and thus only works with normalization. Given these results, we cannot claim that L2-normalization brings any significant improvements to tested combinations. However, if we extend this architecture with more models, which could present win rates on a different scale, using L2-normalization might be a safer choice.

Table 4: Win rate of the best AB combination measured on both train and test levels. All baselines are included.

Level	Humans	AB	PJ	Random
82	**9.60**	8.08	1.33	0.13
62	**21.92**	17.37	10.21	5.22
136	3.10	**4.41**	0.81	0.08
147	**6.90**	3.01	0.55	0.08
163	**1.03**	0.14	0.01	0

Finally, it is worth noticing that the best performing model uses only a subset of all the skills. In particular, PJ, PT, and BRoE are the three fundamental skills that significantly improve the win rate. DB seems to provide a small benefit to the hybrid, bringing its performance closer to humans. On the other hand, adding CU to the hybrid instead worsens the performance, proving that using all the sub-policies together does not seem to be the best choice.

Related Work

The generalization ability of RL agents has not always been the primary focus of research in the field. The most used benchmarks, like the ALE (Bellemare et al. 2013), report the performances of an agent over the same environment where it is trained, thus not making a clear distinction between training, validation, and testing. This trend is well analyzed by Cobbe et al. (Cobbe et al. 2018), who propose a new benchmark to measure the RL generalization. In particular, they create an environment, called *CoinRun*, where an agent can be trained over some levels and tested on others by measuring the zero-shot performance. Separating training and testing levels enables us to assess how well an agent is generalized. The authors also show that more training set levels lead to better performances in the test set.

Florensa et al. (Florensa et al. 2017) propose a hierarchical model to tackle environments with sparse rewards. They first let different agents learn different skills, unrelated to the final goal, in a pre-training environment with proxy rewards. Then, they train one high-level policy for the downstream task to solve, which picks one of the pre-trained policies and sticks to it for τ steps. The weights of the high-level policy can be jointly optimized with those of the lower-level ones.

Simpkins et al. (Simpkins et al. 2019) try to solve the problem of reusing RL modules with different reward scales without having to sum them linearly. They propose an architecture where an arbitrator chooses one of the modules at each timestep and executes the action proposed by that module. This allows being scale-invariant, as each module can have Q-values of any magnitude without affecting the arbitrator's choice since they are not summed in any way.

A straightforward way to incorporate relevant events into the reward function is through a linear approach. Lample et al. (Lample et al. 2017) make use of this idea to train an agent in the *VizDoom* environment, where additional positive rewards are given for events such as picking up ammunition or walking on the map. In contrast, negative rewards are given for events like wasting ammunition. These rewards are summed to the one given by the environment, representing the simplest linear approach to combine intrinsic and extrinsic rewards. A more complex architecture is proposed by Van Seijen et al. (Seijen et al. 2017) that consists of decomposing a reward function in multiple ones, learning a value function for each reward stream.

An interesting intrinsic-only method is proposed by Justensen et al. (Justesen et al. 2018), where rewards are given using an automated heuristic. An expert first defines a series of relevant events for the task at hand, and the agent is rewarded for achieving those events: the more the agent achieves them, the less reward it will gain. The idea is that this temporal frequency heuristic will allow the agent to explore more complex behaviors and learn to achieve them. This approach can be adopted to learn skills that are difficult to define with a manual reward without introducing human bias.

Conclusions

We proposed one approach to develop an RL agent to gameplay Candy Crush Friends Saga (CCFS). The previous works show that an agent fails to generalize over the test levels if it is trained using extrinsic rewards. We thus tackle the generalization problem by drawing inspiration from human behaviors. Human players learn strategies and patterns as they progress through the game and combine and reuse them when they are in a suitable board state. These strategies are not required to be explicitly related to the objective of a level, but their execution can indirectly help the player towards achieving it.

We identified a set of basic skills that a player should execute to improve its win rate. In particular, we focus on creating Special candies (BRoE), using Special candies (CU), and two removing Blockers skills: damaging Blockers (DB) and progressive tiles (PT). We design a set of candidate intrinsic reward functions for learning the defined skills and train one agent for each function. After obtaining a suitable set of policies, we designed an architecture that allows a new agent to use the learned skills. Through the Average Bagging (AB) hybrid architecture, we combine the sub-policies' values associated with each basic skill and pick the action with the highest average value.

Through the experiments, we show that BRoE and CU improve the creation and usage of Special candies, respectively. We also observe that BRoE outperforms all the baseline policies under all performance metrics, creating up to three times as many Special candies as the other models and using more of Special candies. BRoE also shows a higher win rate than Progressive Jam (PJ) extrinsic reward on the test levels. Moreover, DB and PT reward functions exhibit a higher clearing percentage than the baselines on the training and test levels. Finally, we observe that the AB architecture that combines different basic skills outperforms the baselines, winning more than twice as much as an agent trained with PJ extrinsic reward and surpassing human performance on one test level.

Given the benefits of AB, a promising direction for future work would be to extend the architecture by making use of a weighted average, and learning the set of weights. The aim is to find a set that works across levels and improve generalization on the test levels even more, hopefully closing the gap with human performance.

References

Bellemare et al., M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47: 253–279.

Breiman, L. 1996. Bagging predictors. *Machine learning* 24(2): 123–140.

Chentanez et al., N. 2004. Intrinsically motivated reinforcement learning. *Advances in neural information processing systems* 17: 1281–1288.

Cobbe et al., K. 2018. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341* .

Fischer, M. 2019. *Using Reinforcement Learning for Games with Nondeterministic State Transitions*. Master's thesis, Linköping University.

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

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

Justesen et al., N. 2018. Automated curriculum learning by rewarding temporally rare events. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Karnsund, A. 2019. *DQN Tackling the Game of Candy Crush Friends Saga: A Reinforcement Learning Approach*. Master's thesis, KTH Royal Institute of Technology.

Lample et al., G. 2017. Playing FPS games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Lorenzo et al., F. 2020. Use All Your Skills, Not Only The Most Popular Ones. In *2020 IEEE Conference on Games (CoG)*, 682–685. IEEE.

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

Mnih et al., V. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540): 529–533.

Schmidhuber, J. 2010. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development* 2(3): 230–247.

Seijen et al., H. 2017. Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems*, 5392–5402.

Shin et al., Y. 2020. Playtesting in Match 3 Game Using Strategic Plays via Reinforcement Learning. *IEEE Access* 8: 51593–51600.

Simpkins et al., C. 2019. Composable modular reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4975–4982.

Zheng et al., Z. 2020. What Can Learned Intrinsic Rewards Capture? In *International Conference on Machine Learning*, 11436–11446. PMLR.