DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Improving Generalization in Reinforcement Learning using Skill-based Rewards

An application to Candy Crush Friends Saga

FRANCESCO VITO LORENZO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Improving Generalization in Reinforcement Learning using Skill-based Rewards

An application to Candy Crush Friends Saga

FRANCESCO VITO LORENZO

Master in Computer Science
Date: June 16, 2020
Supervisor: Tianze Wang
Industrial Supervisor: Sahar Asadi
Examiner: Amir H. Payberah
School of Electrical Engineering and Computer Science
Host company: King Digital Entertainment
Swedish title: Förbättring av generalisering i förstärkningsinlärning med färdighetsbaserade belöningar

Abstract

Reinforcement Learning is a promising approach to develop intelligent agents that can help game developers in testing new content. However, applying it to a game with stochastic transitions like *Candy Crush Friends Saga (CCFS)* presents some challenges. Previous works have proved that an agent trained only to reach the objective of a level is not able to generalize on new levels. Inspired by the way humans approach the game, we develop a two-step solution to tackle the lack of generalization. First, we let multiple agents learn different skills that can be re-used in high-level tasks, training them with rewards that are not directly related to the objective of a level. Then, we design two hybrid architectures, called *High-Speed Hierarchy (HSH)* and *Average Bagging (AB)*, which allow us to combine the skills together and choose the action to take in the environment by considering multiple factors at the same time. Our results on CCFS highlight that learning skills with the proposed reward functions is effective, and leads to a higher proficiency than the baselines applying state of the art. Moreover, we show that AB exhibits a win rate on unseen levels that is twice as high as that of an agent trained only on reaching the objective of a level, and even surpasses human performance on one level. Overall, our solution is a step in the right direction to develop an automated agent that can be used in production, and we believe that with some extensions it can yield even better results.

Sammanfattning

Förstärkningsinlärning är en lovande metod när det kommer till att utveckla intelligenta agenter som kan stödja spelutvecklare i att testa nytt spelmaterial. Att använda intelligenta agenter på ett spel med stokastiska övergånger så som *Candy Crush Friends Saga (CCFS)* uppvisar en del utmaningar. Tidigare arbeten has visat att en agent som endast är tränad att klara av en spelnivås specifika objektiv inte lyckas generalisera till andra spelnivåer. Vi låter ett flertal agenter lära sig olika färdigheter som sedan kan återanvändas i överordnade uppgifter, sedan träna agenterna med belöning som inte är direkt relaterade till objektivet för den specifika nivån. Sedan designar vi två hybridarkitekturer, som vi kallar *High-Speed Hierarchy (HSH)* och *Average Bagging (AB)*, som tillåter oss att kombinera de olika färdigheterna tillsammans och sedan välja den handling agenten tar i miljön genom att ta hänsyn till flera faktorer åt samma gång. Våra resultat på CCFS utmärker sig i den mening att agenter lär sig färdigheter med den föreslagna belöningsfunktionen effektivt, vilket leder till en högre skicklighet i jämförelse med referensagenter som använder sig av state-of-the-art metoder. Därutöver visar vi att AB påvisar en vinstfrekvens på osedda spelnivåer som är dubbelt så hög mot en agent tränad på att endast klara av en spelnivås specifika objektiv. AB överträffar till och med mänsklig prestation på en spelnivå. Våran lösning är ett steg i rätt riktning gällande utveckling av en automatiserad agent som kan användas i produktion, och vi tror att med viss utbyggnad är det möjligt att nå ännu högre resultat.

Acknowledgments

I would like to start by thanking Tianze Wang and Amir H. Payberah, both from the division of Software and Computer Systems at the *Royal Institute of Technology (KTH)*, for your valuable assistance and academic support throughout this degree project. I would also like to thank Sahar Asadi, Lele Cao, Alice Karnsund and Alex Nodet, from *King Digital Entertainment*, for your constant supervision and guidance in the six months I spent conducting this research. I would not be able to present these results without your precious inputs. Furthermore, I would like to express my gratitude to Mathias Helgesson, my friend and colleague at King, for sharing every step of this journey with me. I will miss all the random Slack calls and table tennis games that made the ordinary much more fun. On this note, I would also like to thank Matteo Biasielli and everyone in the AI-team for making me feel part of the King family. Last but not least, I want to thank Michael Sjöberg and Sahar Asadi for giving me this opportunity in the first place.

I would like to thank my family for always believing in me during my academic career, especially when I was far from home. I want to conclude by expressing my gratitude to all the wonderful people I have come to know in Sweden, with a special thanks to Svenja Räther, Vittorio Denti, Francesco Staccone, Andrea Scotti and Gabriele Gullì, for putting up with me in this past year, even though it was not always easy. *Carichi, sempre*.

Stockholm, June 16, 2020
Francesco Vito Lorenzo

Contents

Acronyms

AB Average Bagging.

AHSR Average Human Success Rate.

AI Artificial Intelligence.

ALE Arcade Learning Environment.

Bv1 Blockers v1.

Bv2 Blockers v2.

CCFS Candy Crush Friends Saga.

CCJS Candy Crush Jelly Saga.

CCS Candy Crush Saga.

CCSS Candy Crush Soda Saga.

CCv1 Candy Creation v1.

CCv2 Candy Creation v2.

CNN Convolutional neural network.

CUv1 Candy Usage v1.

CUv2 Candy Usage v2.

DP Dynamic Programming.

DQN Deep Q Network.

DT Delta Tiles.

ELU Exponential Linear Unit.

ERB Experience Replay Buffer.

FCL Fully-Connected layer.

HRL Hierarchical Reinforcement Learning.

HSH High-Speed Hierarchy.

MC Monte Carlo.

MCTS Monte Carlo tree search.

MDP Markov Decision Process.

ML Machine Learning.

MLT Multi Level Training.

PC Principal Component.

PCA Principal Component Analysis.

PI Policy Iteration.

PJ Progressive Jam.

PT Progressive Tiles.

RL Reinforcement Learning.

SL Supervised Learning.

SLT Single Level Training.

SotA State of the Art.

SSN Single Stream network.

TD Temporal Difference.

UL Unsupervised Learning.

VI Value Iteration.

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides the reader with an overview of the whole thesis. Firstly, Section 1.1 introduces the company where the research has been carried out, its products, and the motivation for this project. Section 1.2 provides a brief history of the technical area on which this work builds upon. Then, Section 1.3 formulates the problem in details, followed by the research questions in Section 1.4. Section 1.6 explains the scope of the research and the constraints that limit it. Section 1.5 lists the contributions of the thesis. Finally, Section 1.7 provides an outline for the rest of the document.

1.1 Motivation

King Digital Entertainment, abbreviated to *King*, is a video game developer founded in Stockholm, Sweden in 2003. The company started by developing browser games on its portal, `King.com`. Then, it shifted its focus on developing Facebook-based games to attract a larger player base, becoming the second publisher on the platform in 2012, with games such as *Bubble Witch Saga* and *Candy Crush Saga (CCS)*. In the same year, King started porting these games for mobile platforms like iOS, pursuing a cross-platform strategy that allows players to synchronize their data between the Facebook version of the games and the mobile one. In 2013, the company removed all in-game advertising from their games, relying solely on its Freemium model based on microtransactions, which represented 99% of its revenues in that year. In 2014, King went public in the US with the largest initial public offering for a mobile gaming company at the time. From then on, the company stopped looking for another major hit like CCS and focused more on building a balanced portfolio of games. The company was acquired by Activision-Blizzard in 2016, becoming part of the Activision-Blizzard-King family, which holds the largest game network in the world. New releases of Candy Crush games followed in parallel, with *Candy Crush Soda Saga (CCSS)* in 2014, *Candy Crush Jelly Saga (CCJS)* in 2016 and *Candy Crush Friends Saga (CCFS)* in 2018, which is the latest addition to Candy's franchise. With four major games in that franchise, King is now focusing on enabling players with new content on their more popular games, while expanding the portfolio with new games that cover different areas from the available ones.

Candy Crush Franchise

All games in the Candy franchise are *Match-3* games, with a game board that is a 9×9 grid filled with colored candies. The core *gameplay* features a player who takes actions in the game by swapping two adjacent candies in order to match horizontally or vertically three or more candies of the same color. Candies that are part of a match are removed from the board and substituted by other ones falling from the top, filling the board at random. If four or more candies are involved in the match, a new *Special candy* with a special ability is created. Different types of Special candies exist in the games, which are created based on the number of candies that are matched and their arrangement on the board. Levels can contain *Blockers*, which are elements that occupy a cell on the board and prevent a player from making matches there. Blockers can be removed by making matches on the neighboring tiles. Each level features a particular objective to complete, and players have a maximum number of moves to complete it, otherwise losing the level. An example of a level from CCFS is displayed in Figure 1.1, containing game features like Blockers and Special candies in the bottom. Its objective is to spread the *jam*, represented by a red background, all over the board. This can be done by making matches that involve a tile with jam already on it. In the figure, the player is prompted to match the three red candies in the top-right corner.

Playtesting

King's approach to supporting its current live games in the Candy franchise consists of constantly releasing new content and creating events to keep the players entertained. For instance, in 2019 King released around 1905 new levels for CCS. At King, the traditional release process without automation used to involve multiple steps:

1. Level designers create and develop new levels by leveraging their expertise;

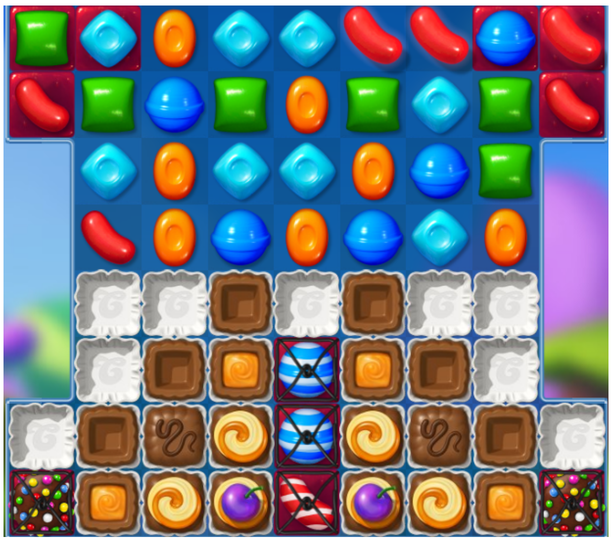
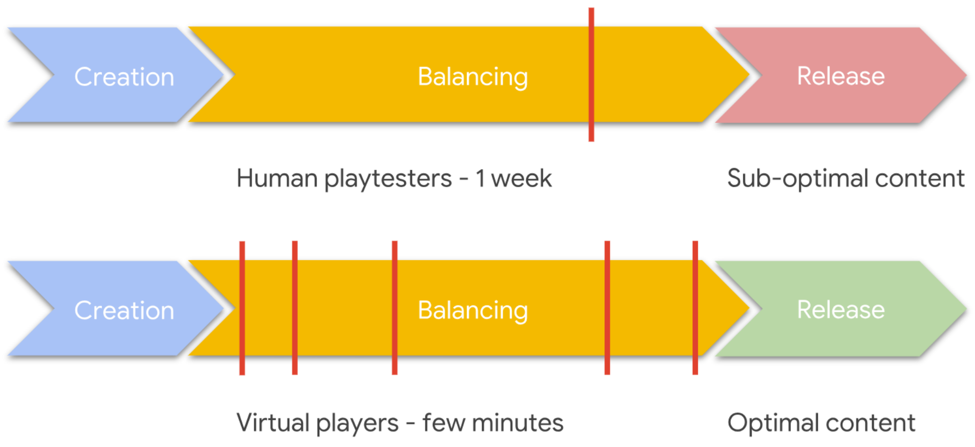


Figure 1.1: A screenshot taken from level 65 in CCFS, suggesting the player to swap the two candies in the foreground in the top-right corner. This level includes features like Blockers and Special candies in the initial board.

2. Newly developed levels are tested by dedicated people to make sure that they are balanced, such that players can enjoy them;
3. Level designers implement the feedback received and release the levels in the games.

The second step in this process is called *playtesting*, and it represents a technique used by gaming companies to assist game development by either performing content balancing, like in this case, or finding bugs in the game. However, playtesting without automation is the most time-consuming part of the process, since it can take up to a week to gather feedback from the testers. As a consequence, level designers have to switch context from their current work and implement the feedback they received for the levels they developed a week before. This is a major source of distraction and is not a creative process, so it is a waste of potential for the designers. Moreover, testers do not statistically represent the entire player base. They are biased by already knowing how to play the game at a good level, and are limited in the number of tests they can run within one week.

For the aforementioned reasons, in the last three years, King has been exploring alternative approaches to ease the bottleneck of playtesting, making this step fully automated. The first approach [1], which has currently been used in production for two years, is based on *Supervised Learning (SL)*, a sub-



Source: Tech at King, Medium [1]

Figure 1.2: An overview of the process of content release, where feedback is represented by red lines. **(Top)** Human playtesters. **(Bottom)** Automated playtesters.

field of *Machine Learning (ML)* that uses supervised data to build predictive models. In particular, the resulting model can predict the action that is most likely going to be taken by humans in a given board configuration, and thus allows level designers to receive immediate feedback when they create a level, rather than at the end of the whole process.

However, King wants to automate playtesting also for features that are yet to be released, and for new games. In both cases, human data is not available, so the previous model cannot be trained properly, and its performance would not be representative of that of the average human player. For this reason, the company is experimenting with *Reinforcement Learning (RL)*, another sub-field of ML that does not leverage on human data to build a predictive model. This work is part of this ongoing research at King and is focused on how to improve the performance of the existing RL implementation [2] for playtesting purposes.

1.2 Research Area and Context

RL is a sub-field of ML that deals with teaching a software agent how to take actions in an environment to maximize a reward signal. The topic is not a new research field, having its roots in the optimal control theory of Bellman [3],

[1]https://medium.com/techking/human-like-playtesting-with-deep-learning-92adafffe921

and in trial-and-error learning from behavioral psychology. The first major contribution to the field of RL was made by Witten [4] with his work on *Temporal Difference (TD)* learning, and was followed by plenty of research on the topic. The theoretical foundations of RL are better laid out in Section 2.2.

Games have always been a popular testbed for the research in the area of RL, as they are software entities that can be interacted with by a software agent at high computational speed. Moreover, results on games can easily be compared to those of humans, highlighting progress in the research. In fact, RL started gaining popularity among the public in 2015, when researchers at DeepMind published a paper in Nature [5] showing how software agents, trained with RL, were able to achieve human performances on a set of classic Atari games like Breakout. In March 2016, RL made worldwide headlines when DeepMind's AlphaGo agent [6], trained to play Go through RL techniques, was the first *Artificial Intelligence (AI)* to defeat the world champion Lee Sedol in a best-of-five match, achieving a task that was thought to require at least 10 more years of advancements in the field.

Generalization

In cognitive psychology, *generalization* refers to the ability of an agent (either a human or an animal) to re-use experiences from past situations in a present setting that is regarded as similar. Experiences are apprehended through the process of learning, where they are abstracted as rules and patterns that apply during a given situation. When a new situation with similar characteristics occurs in the present, the agent can transfer the knowledge learned in the past and act accordingly. Such a process happens naturally to infants in their earliest stages of life, as they can generalize and learn with only a few experiences. This allows them to better navigate the environment around them, recognize people, and learn to speak.

The same concept can also be applied to software agents, or AIs, when they undergo a process of induction. In particular, ML agents abstract and re-use past experiences, available in the form of data or interactions, to provide results when they are faced with new inputs. A lack of generalization means that an agent would be unable to act properly in new situations, effectively not serving its purpose.

An RL agent whose goal is to play new levels of Candy Crush to test if they are balanced must be able to generalize. When used in production by level designers, such agent will be presented with new levels that have different board configurations and possibly new game features from the ones it encountered before. Just like a human can extract common patterns from the previous levels he/she played, and apply those rules when facing a new level, an RL agent should transfer the knowledge it learned in the past and apply it to play the new level. If that agent had to learn everything from scratch, it would be unusable in a production environment for playtesting. Generalization in the context of RL is detailed in Section 5.5.

1.3 Problem Definition

The purpose of this research is to improve the generalization ability of the existing RL agent, developed in the previous years at King, to bring it one step closer to being used for playtesting in production. In particular, the focus is on addressing a key challenge faced by the current implementation, which struggles to generalize and transfer knowledge from the levels it was trained on to new levels it has never seen before.

The inspiration for this research comes from observing the behavior of human players; a new player that progresses through a Candy game learns basic skills that are not necessarily related to the particular objective of the level he/she is facing. These skills can be re-used to complete high-level tasks that can help the player to win new levels, regardless of their objective. For instance, in levels where the objective is to spread the jam, the player will immediately recognize the importance of a high-level task such as destroying all Blockers on the board. To do that, it first has to learn how to destroy each type of Blocker, which is a skill not directly related to spreading the jam.

A human does not need to learn everything from scratch since it can utilize already learned skills across different levels. Similarly, we first focus on how to teach an agent a set of basic skills, enabling it to approach new levels without starting tabula rasa. Instead of rewarding the agent for achieving the objective of a level (*extrinsic* reward), we teach these 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)*.

If this first phase is successful, we will then attempt to combine these skills by designing hybrid architectures that allow a new agent to select the most appropriate behavior according to the board configuration at hand.

1.4 Research Questions

To formalize what was mentioned in the previous section, we define the following research questions:

1. Can an RL agent learn a set of low-level skills that can be used in high-level tasks?
    1. Can we propose a set of good intrinsic reward functions to learn them?
    2. How well can an agent perform these skills?
    3. Can they be be transferred on new levels?
2. How can a new agent employ this set of skills to win levels?
    1. What are efficient architectures to combine the previous skills with a goal-oriented behaviour?
    2. Is the new agent better than one rewarded only by pursuing the objective of a level?
3. Do the previous architectures improve the generalization ability of the agent?
    1. What is the performance on test levels with the same objective used during training?

For each research question, we formulate a set of hypotheses that are either confirmed or invalidated by empirical evidence. These hypotheses are stated in Chapter 7, together with the experimental results needed to validate them. A discussion on whether the hypotheses are correct is provided in Chapter 8.

1.5 Contributions

The contributions of this work can be divided based on the three research questions. Firstly, we show that intrinsic motivation represents a successful technique to design reward functions that can be used to teach an agent a set of basic skills. We discover a strong correlation between these skills and the overall win rate: the agents trained with intrinsic rewards outperform the baselines trained with extrinsic rewards, despite not knowing anything about how to win a level. Regarding Special candies, in Section 5.3.1 we design a technique to normalize skill-based reward functions that significantly outperforms the approach presented in [7], and which allows creating all the different Special candies while maintaining a good balance and stability in training. This method also enables an agent to win more than one trained with extrinsic rewards, confirming the input of our level designers, who believe that creating Special candies is a key factor to win in CCFS. As part of this first contribution, we have submitted our results to *IEEE Conference of Games 2020* as a short paper, which got approved.

Secondly, we design two hybrid architectures, called *AB* and *HSH*, which allow us to combine the basic skills to improve the overall win rate of an agent. In particular, we show that AB significantly outperforms the baselines, winning more than twice as much as an agent trained with extrinsic rewards, and even surpasses human performance on one test level. With the results of HSH, we confirm the findings of [2], showing that sparse rewards do not work in environments like CCFS, and should be avoided in favor of more dense designs.

Finally, we compare a training pipeline where we train an agent on multiple levels at the same time to one where we train the same agent only on one level. The purpose is to verify if the former is more suited than the latter to train agents that generalize better across multiple levels. Our results do not confirm nor deny this claim and should be completed with further research. However, our hybrid architectures achieve a performance on unseen levels that is 53% closer to the average human player than extrinsic agents, proving that our approach is overall better to improve generalization.

1.6 Delimitation

This thesis directly builds on the work carried out by previous researchers at King [2], which will also be used as a baseline in the experiments. To ensure comparability with their results, we use CCFS as a testbed for our research. At the time of writing, the game features more than $3\ 000$ levels released for players. Reaching a sufficient degree of generalization on so many levels is not an easy task, and considering their heterogeneity and differences, highlighted in Chapter 4, it represents an open problem in the literature that is being successfully addressed only by a handful of specialized research companies.

The amount of computational resources is limited to what the company has, so most computation is performed on the server available on-premise for this research project, running one Nvidia Tesla P100 GPU. Google Cloud virtual machines instances are available in case of need, though we prefer to rely more on the machines in-house.

Another limitation is related to the execution speed of the game. CCFS has been developed and optimized to run on mobile platforms and on Windows, while for this research an ad hoc build of the game was developed to run in a container, which is not its native environment. Additionally, the game itself is not optimized to run in a simulation environment, so it effectively has to play at human speed.

Due to the aforementioned reasons, we narrow down the scope of this research to a subset of the available levels, objectives, and game features. In particular, we only use one game objective, five different training levels, and four different test levels. We select the training and test sets to include all the possible game features for the objective we use and to cover a broad range of level difficulties. Moreover, the game has five different characters that can help a player to complete a level, and one of them is selected at the beginning of each level. In this research, we always select the one used in [2]. Further details regarding delimitation are presented in Chapter 4.

1.7 Outline

The rest of the thesis is organized as follows. Chapter 2 provides an exhaustive introduction to RL. Firstly, it introduces the basics of ML. Then, it goes over the foundations of RL, lays down the theoretical framework behind it, illustrates a first family of solutions used when a model is available, and finally dives into RL approaches.

Chapter 3 highlights the contributions from existing literature that inspired this work. At first, it dives deep in the topic of generalization in RL, explaining existing solutions that directly address the problem, or which could be used for that purpose. Then, it sums up past results on playtesting, describing how it was approached by third parties and at King.

Chapter 4 explains the context in which this research is carried out. It first describes the mechanics and features of CCFS. Then, it provides useful details on how the game is implemented and how the environment is encoded.

Chapter 5 presents the first part of the solution, related to learning skill-based behaviors through intrinsic rewards. It first explains the reward functions designed for each skill. Then, it goes over the deep neural network used. Finally, it explains the different generalization techniques we used to train agents.

Chapter 6 presents the second part of the solution, which are the hybrid architectures to combine the skills. The first architecture introduced is inspired by ML techniques, while the second one is a more complex RL architecture based on hierarchies.

Chapter 7 presents the results of the experiments. Firstly, it details the common experimental setup shared by all experiments. Then, after stating a series of hypotheses to validate the research questions, it presents the results for skill learning, followed by the ones for hybrid architectures.

Chapter 8 provides our perspective on the results obtained. It first presents a discussion related to skill learning, touching on limitations and extensions. Then, it discusses the results of the hybrid models, also mentioning limitations and future work.

Chapter 9 concludes the whole thesis. It discusses key issues related to ethics and sustainability, summarizes the methods followed, and the results obtained, and finally provides insights to extend this work.

Chapter 2

Background

This chapter presents the theoretical foundations required to understand the content of this thesis. In particular, Section 2.1 provides a brief overview of ML and one of its sub-fields: *SL*. In particular, it explains the Bias-Variance trade-off, which is fundamental to understand what generalization means. Section 2.2 introduces the theoretical framework of RL, providing its foundations and explaining the concept of *Markov Decision Process (MDP)*. Section 2.3 goes over *Dynamic Programming (DP)*, a first family of solutions that can be used when enough information is available. Then, sections 2.4 and 2.5 explain the RL approach to tackle the same problem of DP, when less information is available. Finally, Section 2.6 introduces function approximators, which extend the previous RL solutions to work in environments with high dimensionality.

2.1 Machine Learning

A formal definition of ML by Mitchell [8] states that:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In particular, ML can be considered as a sub-field of AI where knowledge is represented by experience. Whereas in traditional programming data is fed into a program to generate an output, in ML data and outputs are used together to generate a program.

Traditionally, ML is divided in three main branches [9]: SL, UL and RL. An overview of the first is provided in this section, the second is out of the scope of this work, and the last one is discussed in the next section.

Supervised Learning

The goal of *SL* is to estimate the unknown model f that maps a set of known inputs X, called *features*, to a set of known outputs t, called *targets*. Experience is given in the form of a training set $D = \{\langle x \in X, t\rangle \mid t = f(x)\}$. According to the type of t, SL can be further classified as:

- $t \in \mathbb{R}$: *regression* problem
- $t \in [0, 1]$: *probability estimation* problem
- $t \in \{C_1, C_2\}$: *binary classification* problem
- $t \in \{C_1, C_2, \ldots, C_n\} \mid n > 2$: *multi-class classification* problem

Bias-Variance Trade-off

Estimating the unknown model f means finding a function h that approximates f well enough, according to some performance metrics. This process involves multiple steps. First of all, depending on the problem at hand, we need to define a loss function $L_{true}(t, y)$ that measures the cost payed by associating a label $y = h(x)$ to the input x instead of the true label $t = f(x)$. The expected loss is given by:

$$\mathbb{E}[L_{true}] = \int\int L_{true}(t, h(x))p(x, t)dxdt \tag{2.1}$$

This cost is usually measured as a form of distance between y and t. For example, a common loss function for regression is the *squared loss*:

$$\mathbb{E}[L_{true}] = \int\int (t - h(x))^2 p(x, t)dxdt \tag{2.2}$$

Then, we have to define a hypothesis space H comprising a family of functions that may contain f, and finally solve an optimization problem to find the function $h \in H$ that minimizes the defined loss L_{true}.

In real-world applications of ML, the amount of data in the training set D is limited, so we have to rely on a proxy of the true loss function L_{true}, calculated over the available input points:

$$\overline{L_{proxy}} = \frac{1}{N} \sum_{i=1}^{N} L_{proxy}(t_i, h(x_i)) \tag{2.3}$$

Such an approximation leads to an optimization problem whose result is generally different from the one we would get using L_{true}, and this requires prudence in the choice of the hypothesis space H. If L_{proxy} is not a good proxy of L_{true}, then:

- A larger family of models H, even though more likely to include f, can cause the optimization process to have very different results from f, as the variance is higher.
- A smaller family of models H causes the optimization process to be less affected by L_{proxy}, though since it may not include f, it can still lead to bad candidates h.

This trade-off in the expressiveness of H is called *bias-variance* trade-off, where a larger H represents a model family with low bias (more likely to contain f), but high variance (the results may differ a lot when repeating the same optimization process), whereas a smaller H represents a model family with high bias (less likely to contain f), but low variance (the results are closer when repeating the same optimization process). The trend in ML has been to rely on more expressive families of models, and use techniques such as regularization to decrease their variance.

Overfitting and Underfitting

Ideally, we are interested in finding a function h that minimizes L_{true}. Optimizing for the true loss means that the model h in H is obtained by minimizing the *prediction error*, which is the error over all the points in the input space. As mentioned before, this metric is impossible to evaluate, as the amount of data in D is limited. This forces us to minimize L_{proxy} instead, meaning that we are optimizing for the *training error*, which is the error over all the available input points.

One way to obtain good results when optimizing for the training error is to choose a more complex family of models H: the more complex the family is, the better it can fit the training data, leading to a very small training error. As shown in Figure 2.1, neither a sinusoidal g nor a linear model l can perfectly fit the available training points, and they will have a higher training error

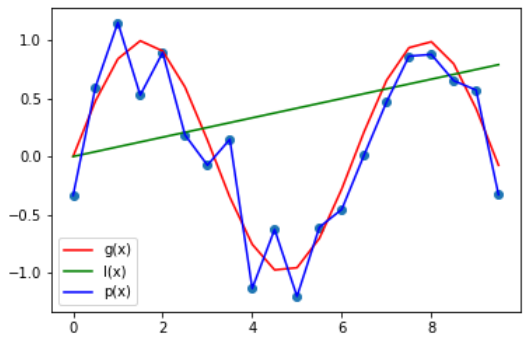


Figure 2.1: 20 points generated from $sin(x) + U(-0.5, 0.5)$. **g(x)** is $\sin(x)$, **l(x)** is $x/12$ and **p(x)** is a polynomial function of order 21.

than a more complex model like p, which instead perfectly fits all the training points. Still, it is clear from the picture that the model that is more adequate to approximate the distribution of the points is the sinusoidal one, even though it does not perfectly fit the training data.

The underlying issue is that, other than being limited, data can also be noisy. The labels provided in the training set D are not necessarily the same as $f(x)$, but they can be affected by noise, represented as a random variable with unknown distribution ϵ. For an additive noise, t becomes:

$$t = f(x) + \epsilon(x) \tag{2.4}$$

In fact, the true model $f(x)$ used to generate the data in Figure 2.1 is a sinusoidal one, with addictive uniform noise. Perfectly fitting the training data means that we are also fitting the noise in it, leading to undesired behaviors. If the previous models were used to predict the label associated with a new input point x, then $g(x)$ would be the closest one to $f(x)$. $p(x)$ represents a much more complex model than a sinusoidal, and it ended up fitting ϵ as well. In this case, the model is said to be *overfitting*. On the other hand, $l(x)$ should not be used for prediction either, as it is so simple that it cannot even properly fit the training data. In this case, the model is said to be *underfitting*.

Choosing the right model complexity is fundamental to have good predictive performances. The ability of a model to perform well over unseen data is called *generalization*, and it is a desirable property to have in SL, as the models should be used to provide predictions for new data, rather than for data that is already labeled. Section 5.5 investigates generalization in the context of RL.

2.2 Foundations of Reinforcement Learning

RL deals with how an *agent*, interacting with an *environment*, should select *actions* in order to maximize a cumulative *reward* signal. The problem can be formalized by considering how the actors interact, as represented in 2.2. At each timestep t, the agent:

1. Executes an action a_t by choosing it from the available actions
2. Receives a scalar reward r_t representing how well it is doing at time t
3. Receives a quantity s_{t+1} encoding the new state of the environment after a_t was taken

and an *interpreter*:

1. Executes action a_t in the environment and observes how it evolves
2. Emits the scalar reward r_t
3. Encodes and emits the new state s_{t+1}

The environment can be categorized based on the state representation of the agent and the environment itself. The information used by the interpreter to emit a reward and a new state are called *environment state*, while the one used by the agent to select the next action is called *agent state*. When the two representations are different, the environment is said to be *partially-observable*, while when they are equal it is *fully-observable*.

The agent can be categorized based on three main components that distinguish it. The first one is the *policy*, representing the behavior of the agent, i.e., how it selects the next action in a given state. An agent using a policy to select the next action is called *policy-based*. The second component is the *value function*, either measuring how good a state is (*state-value function*) or how good taking an action in a state is (*action-value function*). An agent using a value function to choose the next action to take is called *value-based*. If an agent uses both a policy and a value function it is called *actor-critic*. The third and final component is the *model*, which is the representation of the environment as perceived by the agent, and it is used to predict the next state and the immediate reward. An agent using an explicit representation of the model is called *model-based*, otherwise it is called *model-free*. The full taxonomy is provided in 2.3. This thesis builds on the work carried out at King in the previous years, so the focus will be on model-free and value-based agents. An explanation of the other techniques is available in [10].

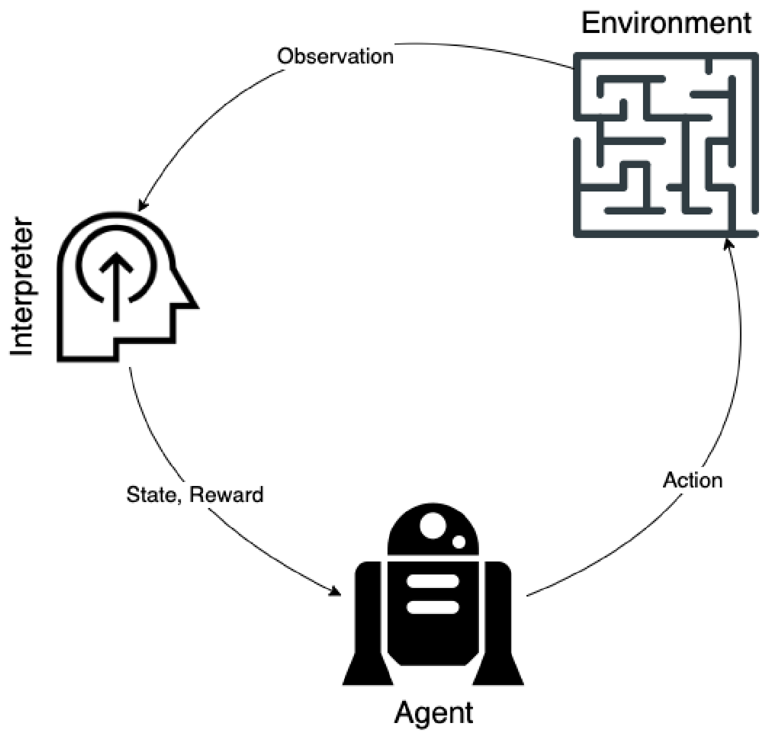


Figure 2.2: Example of an RL framework where an agent takes actions in an environment, and receives responses from an interpreter.

Markov Decision Processs

An environment in an RL problem is usually modeled using the framework of MDPs. The key idea behind them is the *Markov assumption*, which states that the future is independent of the past, given the present. Formally, at time t:

Definition 1: *A state S_t is Markov if and only if*

$$\Pr(S_{t+1} = j | S_t) = \Pr(S_{t+1} = j | S_1, \ldots, S_t) \tag{2.5}$$

The environment state S^e is Markov by definition. If the agent state S^a is Markov, the problem can be formalized as a MDP, otherwise it is formalized as a *partially-observable* MDP. This work uses an environment that is assumed to be fully-observable, thus modeled as a MDP.

A MDP models an environment in which all states are Markov and time is divided into stages. Formally, a MDP is defined as a 5-tuple $\langle S, A, P, R, \gamma, \mu \rangle$:

- S is the set of all the states in the environment
- A is the set of the actions that can be taken

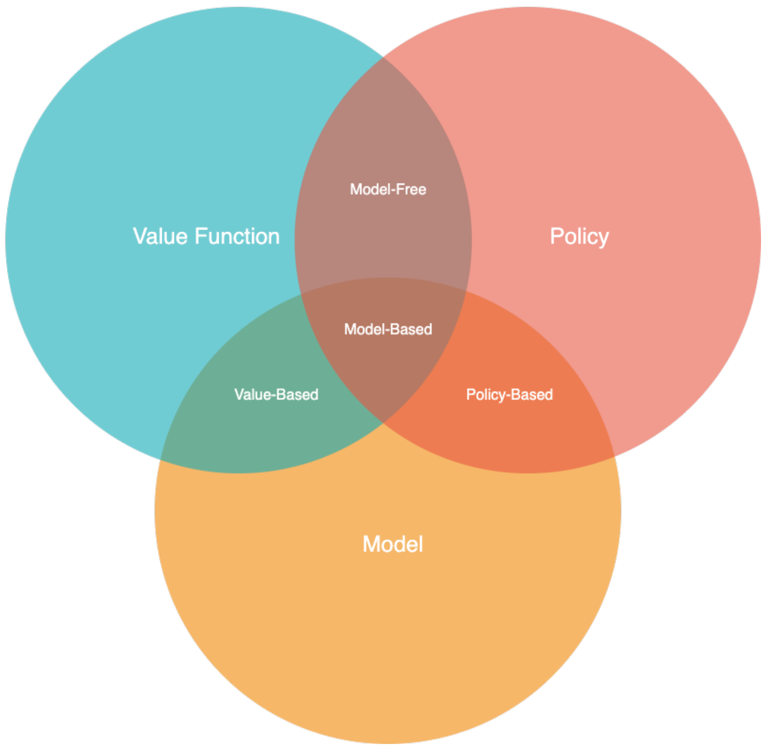


Figure 2.3: Taxonomy of RL agents according to policy, value function and model.

- P is a matrix indicating the probability $P(s'|s,a)$ of ending up in state s' by taking action a in state s
- R is a matrix indicating the expected immediate reward $R(s,a) = \mathbf{E}[r|s,a]$ obtained by taking action a in state s
- $\gamma \in [0,1]$ is the *discount factor*
- μ is a vector indicating the initial probability of being in each state $s \in S$

The *time horizon* of a MDP can either be finite, if the process ends after a pre-defined amount of steps, indefinite, if the process will end after an unknown amount of steps or infinite if the process will go on forever.

Return

An agent seeks to maximize a cumulative reward signal. More precisely, its goal is to maximize the total discounted sum of rewards, also called *return*. Formally, the return v_t at timestep t is defined as:

$$v_t = r_{t+1} + \gamma r_{t+2} + \ldots + = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.6}$$

where the discount factor γ represents the probability that the process will go on, or the present value of future rewards (like in finance).

Policy

An agent obtains rewards by taking actions in the environment according to a policy $\pi(a, s) = \Pr[a|s]$. A MDP tied to a defined policy is equivalent to a *Markov Reward Process* $\langle S, P^\pi, R^\pi, \gamma, \mu \rangle$, where:

- P^π is a matrix indicating the probability $\sum_{a \in A} \pi(a|s) \Pr[s'|s, a]$ of ending up in state s' by following policy π in s
- R^π is a matrix indicating the expected immediate reward $\sum_{a \in A} \pi(a|s) R(s, a)$ obtained by following policy π in state s

Value Function

The state-value indicates the utility of a state. Formally:

Definition 2: *The state-value function $V^\pi(s)$ represents the expected return obtainable by starting in state s and following policy π:*

$$V^\pi(s) = \mathbb{E}_\pi[v_t | s_t = s] \tag{2.7}$$

A more useful quantity to use in practice is the action-value function $Q^\pi(s, a)$, that is the utility of a state-action pair. Formally:

Definition 3: *The action-value function $Q^\pi(s, a)$ represents the expected return obtainable by starting in state s, taking action a and thereafter following policy π:*

$$Q^\pi(s, a) = \mathbb{E}_\pi[v_t | s_t = s, a_t = a] \tag{2.8}$$

Using the definition of return v_t, the value functions can be further decomposed by performing a one-step backup, i.e., looking one step ahead in the future. This formulation for the value functions is called *Bellman expectation equation.* For the state-value function:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1} | s_t = s)] \\ &= \sum_{a \in A} \pi(a|s) \big(R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s') \big) \end{aligned} \tag{2.9}$$

while for the action-value function:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \\ &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a^\in A} \pi(a'|s') Q^\pi(s',a') \end{aligned} \tag{2.10}$$

where both equations are now expressed as a function of the value function of the next state (or state-action pair).

Optimality

In each state, we should not be interested in the value function of that state under a generic policy π, but we should rather look at the maximum value function of that state under all available policies $\pi \in \Pi$. This allows us to define the optimal value function, representing the maximum over all policies. For the state-value function, we have that:

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.11}$$

For the action-value function, we have that:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{2.12}$$

where, if an agent knew the optimal action-value function for each action available in a state s, it could pick the one with the highest Q^*, and be assured that it would be achieving its goal of maximizing the return from that state.

As for the normal value functions, we can perform a one-step backup and obtain the *Bellman Optimality equations*, that are independent from a particular policy. For the optimal state-value function, we have that:

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ &= \max_a \left\{ R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right\} \end{aligned} \tag{2.13}$$

while for the action-value function, we have that:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a'} Q^*(s',a') \tag{2.14}$$

Policy

The value function V^π allows to define an order between policies, where $\pi \geq \pi'$ if $V^\pi(s) \geq V^{\pi'}(s) \quad \forall s \in S$. The following theorem, whose proof is provided in [10], formalizes the definition of an optimal policy π^*:

Theorem 1: *For any MDP:*

- *There exists at least one optimal policy* $\pi^* \geq \pi \quad \forall \pi \in \Pi$
- *All optimal policies achieve the optimal value functions* V^* *and* Q^*
- *There always exists one deterministic optimal policy*

In particular, the last point can be achieved by acting greedily in every state, i.e., choosing the action a with the highest $Q^*(s,a)$ in state s:

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \operatorname{argmax}_{a \in A} Q^*(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{2.15}$$

A MDP is said to be *solved* when an optimal policy is found. One final distinction to make is between *prediction* and *control* problems. In the former, we have a MDP (whose model may or may not be known) and a policy π, and the goal is to calculate the value function V^π or Q^π of that policy. In the latter, we only have a MDP (whose model may or may not be known) and the goal is to obtain the optimal policy π^*, together with the optimal value functions V^* and Q^*. The problem to be solved in the traditional formulation of RL is a control problem, even though this often requires solving a series of prediction problems as well.

2.3 Dynamic Programming

DP is a family of methods that can be used to solve MDPs when the model is available (model-based). This technique is called *planning*, where we can plan ahead and solve the Bellman equations either recursively or in exact form, thus finding an optimal policy. The two main DP techniques are *Policy Iteration (PI)* and *Value Iteration (VI)*, both addressing control problems.

2.3.1 Policy Iteration

PI finds the optimal policy through a process of continuous improvement of policies, until convergence. It starts from an initial policy π_0 and solves a prediction problem to estimate its value function V^{π_0}. Then, it generates a policy $\pi_1 \geq \pi$ by acting greedily over V^{π_0}. This two steps, respectively called *policy evaluation* and *policy improvement*, are repeated until convergence to π^* is achieved.

One way to perform policy evaluation is by using the Bellman equation for V^{π}. Expressing the equation for every state $s \in S$, we obtain a system of $|S|$ linear equations in $|S|$ unknowns, for which a closed-form solution exists. In matrix notation: $V^{\pi} = (I - \gamma P^{\pi})^{-}1R^{\pi}$. Another way to solve the prediction problem is by iteratively applying the Bellman equation to all states. We start from initial values $V_0(s)$ for all states, and then updates those values with a full backup (i.e. assuming knowledge of the value function of all states):

$$V_{k+1} \leftarrow \sum_{a \in A} \pi(a|s) \big[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s') \big] \tag{2.16}$$

where a recursive application of the previous equation is guaranteed to converge to V^{π} given its contraction properties.

Policy improvement is performed by acting greedily with respect to the value function V^{π}:

$$\pi'(s) = \operatorname*{argmax}_{a \in A} \big[R(s,a) + P(s'|s,a) V(s') \big] \;\; \forall s \in S \tag{2.17}$$

where the new value function $V^{\pi'}$, associated to π', is guaranteed to be at least as good as V^{π}, if not better [10]. When $V^{\pi'}$ is equal to V^{π}, π represents an optimal policy for the given MDP, so convergence is reached.

An improved version of PI, called *generalized PI*, introduces a stopping condition to the prediction problem, as we do not need to convergence to V^{π}. An intermediate value function, where the ordering between the utility of each state is respected, is sufficient to perform a correct step of policy improvement and saves computational time in the process.

2.3.2 Value Iteration

VI is conceptually similar to policy evaluation, where full backups are applied to all states to estimate a new value function. The difference is that the Bellman Optimality equation is used for the backups, as opposed to the Bellman Expectation equation:

$$V_{k+1} \leftarrow \max_{a \in A} \big[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s') \big] \tag{2.18}$$

No explicit policy is calculated with VI, and the intermediate V_k may not correspond to any legal policy at all. Nonetheless, convergence is guaranteed to V^* due to the contraction properties of Bellman Optimality backups.

2.4 Model-Free Prediction

RL techniques are used either when the model is not available or when it is too complex, and thus planning is impossible or too computationally demanding. The idea behind RL techniques for prediction and control remains similar to DP, but instead of full backups, they employ sample-based backups, relying on convergence properties in expectation. Prediction problems in RL are solved either through *Monte Carlo (MC)* or TD methods.

2.4.1 Monte Carlo

A value function is defined as the expected return starting from a state or state-action pair. MC methods use the empirical mean as a point estimator of the expected value of the return, and rely on the guarantees of the *Central Limit Theorem* for convergence. In MC, samples are taken from full episodes of experience $s_1, a_1, r_2, a_2, \ldots, s_T$ under policy π, and can only be applied to *episodic* MDPs (i.e. finite or indefinite horizon). Unlike DP, in each state s we only consider the state s' that came after it in that experience, not all the possible states that are reachable from s. Two main techniques are used: first-visit MC and every-visit MC.

In *first-visit* MC, the sample return of a state s is calculated only on the first occurrence of s in the episode by summing all the rewards that follow the state. If s is visited more than once, the sample return is not calculated again, and it is instead assumed to be the one of the first visit to s. The value $V^\pi(s)$ is the mean of the sample returns over all the available episodes of experience.

In *every-visit* MC, the sample return of a state s is calculated on every occurrence of s in the episode, so there will be generally more samples of the return v_t in each episode.

In both cases, the value of $V(s)$ can be updated incrementally when a new episode of experience is accumulated, instead of recalculating the whole mean over all the episodes. Given a new sample return v_t, with $N(s_t)$ the number of samples returns available, the new value function is:

$$V(s_t)^{k+1} \leftarrow V(s_t)^k + \frac{1}{N(s_t)}(v_t - V(s_t)^k) \tag{2.19}$$

The term $\frac{1}{N(s_t)}$ can be generalized as α, also known as *learning rate*, used to keep a running mean for non-stationary problems to forget past experiences.

2.4.2 Temporal Difference

TD methods [11] are similar to incremental update MC, but they are able to learn from incomplete episodes of experience. The return v_t can be decomposed in the TD target $r_{t+1}+\gamma V(s_{t+1})$, so we can look one step into the future, observe the reward obtained in that step, and use the estimated value of the next state s_{t+1} as a proxy of all the future rewards. This process is called *bootstrap*, as it updates the current guess of v_t using another guess, v_{t+1}, and is also used by DP techniques. The update formula for $V(s_t)$ becomes:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2.20}$$

where $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called TD error, and it is the difference between our current estimate of $V(t)$ and the TD target.

A natural extension of the previous method is to bootstrap after more than one step in the future. Ideally, we could observe n steps of experience from an episode, and then bootstrap:

$$v_t = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{t+n} r_{t+n} + \gamma V(s_{t+n}) \tag{2.21}$$

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t)) \tag{2.22}$$

where this method is called *TD(n)*, as opposed to the previous one, *TD(0)*. The benefit of looking more steps into the future is reducing the bias, as there is less reliance on the estimation $V(s_{t+n})$ and more evidence from actual experience. On the downside, the variance is increased, as more steps are being taken in the environment, and each step is only one of many possible paths that can be reached from the previous one. TD(n) gets closer to MC in this sense, where in the limit there is no bias (with first-visit MC) but high variance.

A more sophisticated extension of TD(n) is *TD(λ)*, where the n-step returns are averaged through a weighted sum, allowing to express a trade-off between bias and variance. For more details on TD(λ), the reader can consult [10], since it is not relevant in the scope of this work.

2.5 Model-Free Control

A key distinction to make before introducing control techniques in RL is between *on-policy* and *off-policy* methods. In the former, the policy that is is being learned is the same used to collect the experience, while in the latter the experience can be generated from a different one. Three methods will be presented next: MC control and SARSA as on-policy, and Q-learning as off-policy.

2.5.1 Monte Carlo Control

The idea behind *MC control* is the same as in PI, but it is done model-free. As seen in 2.17, greedy policy improvement requires a model of the environment, as we need to look at the state-value functions of all the next states. If the improvement step is done using the action-value function $Q(s, a)$, the model is not needed:

$$\pi'(s) = \operatorname*{argmax}_{a \in A} Q(s, a) \ \ \forall s \in S \tag{2.23}$$

The policy evaluation step will evaluate $Q(s, a)$ instead of $V(s)$, doing an equivalent MC update where v_t is associated to a state-action pair, rather than a state:

$$Q(s_t, a_t)^{k+1} \leftarrow Q(s_t, a_t)^k + \frac{1}{N(s_t, a_t)}(v_t - Q(s_t, a_t)^k) \tag{2.24}$$

Another issue with policy improvement is that the greedy policy is deterministic by definition, so it will not allow us to know the Q-values of all the state-action pairs, thus making the improvement step impossible. To solve the issue, in model-free control we must use stochastic policies. These are obtainable by using *ϵ-greedy* strategies, consisting of selecting a random action with probability ϵ, and the greedy one with probability $1 - \epsilon$:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon, & \text{if } a^* = \operatorname{argmax}_{a \in A} Q^(s, a) \\ \frac{\epsilon}{m}, & \text{otherwise} \end{cases} \tag{2.25}$$

where this allows us to explore the state-action space and collect experience for more pairs. Moreover, ϵ-greedy policy improvement has the same properties of greedy policy improvement, so the new policy is still greater or equal than the previous one.

On a final note, as in PI, we can perform shallow policy evaluation steps, not converging to Q^π but obtaining a sufficient approximation of it. The MC control algorithm uses this modified policy evaluation step, paired with ϵ-greedy policy improvement, to find an optimal policy π^* from experience. Convergence is guaranteed if all state-action pairs are explored infinitely many times (GLIE) [10].

2.5.2 SARSA

SARSA [12] is similar to MC control, but instead of MC policy evaluation it uses TD policy evaluation steps, exploiting the lower variance properties of

TD methods, and their ability to learn from incomplete episodes. It is still on-policy, and the update formula for $Q(s,a)$ becomes:

$$Q(s_t,a_t)^{k+1} \leftarrow Q(s_t,a_t)^k + \alpha\big[r_{t+1} + \gamma Q(s_{t+1},a_{t+1})^k - Q(s_t,a_t)^k\big] \quad (2.26)$$

where the next action a_{t+1} is also chosen according to the current policy π. SARSA can be seen as the sample version of PI methods from DP. Convergence guarantees for it require GLIE policies and learning rates respecting the *Robbins-Monro conditions* [10]. Finally, SARSA can be naturally extended with TD(n) and TD(λ) policy evaluation, leading to *SARSA(n)* and *SARSA(λ)*.

2.5.3 Q-Learning

Q-learning [13] is an off-policy method, where the policy π we are trying to learn is generally different from the policy $\bar{\pi}$ used to collect experience, also known as *behaviour policy*. This decoupling allows to use experience collected in the past, and learn in both offline and online settings. Whereas SARSA is the sample version of PI, Q-Learning can be seen as the sample version of VI methods from DP. In fact, the formula uses the $\max$ operator to choose the next action, and is decoupled from the true choice that was made when collecting the experience. The update formula becomes:

$$Q(s_t,a_t)^{k+1} \leftarrow Q(s_t,a_t)^k + \alpha\big[r_{t+1} + \gamma \max_{a' \in A} Q(s_{t+1},a')^k - Q(s_t,a_t)^k\big] \quad (2.27)$$

A common choice for the behavior policy is to use an ϵ-greedy version of the target policy π, so that it also improves during the learning process, leading to faster training.

2.5.4 Summary

To conclude this section, a final comparison between DP and RL techniques is presented and summarized in Figure 2.4. Both families of techniques can be classified based on their use of sampling and bootstrap.

Since the model of the MDP is available, DP techniques use expectations (full backups) in their updates and do not rely on sampling. On the other hand, MC and TD methods do not have the model of the MDP, so they rely on sampling from experience to approximate expectation (sample backups).

DP techniques and TD methods make updates by looking one step in the future, and then bootstrap using the current estimate of the next state's value function (shallow backups). MC methods instead look at the full trajectory of the episode, and they never bootstrap (deep backups).

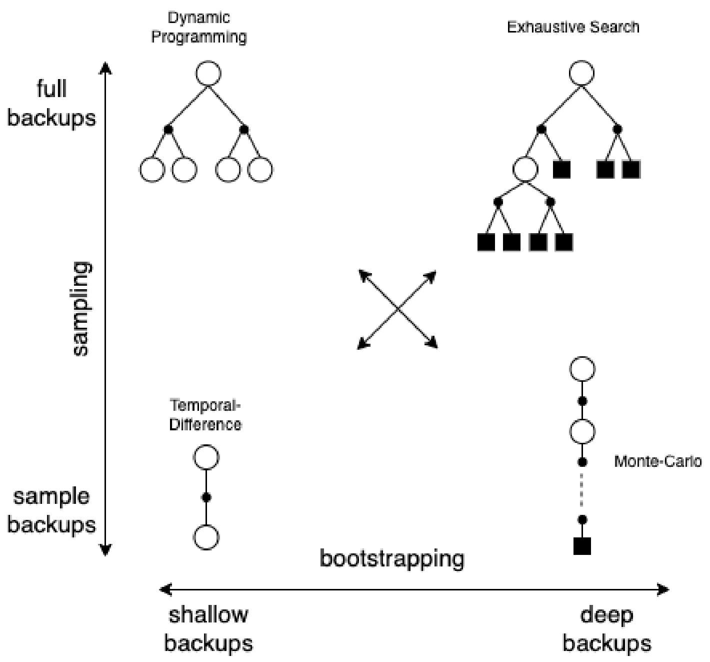


Figure 2.4: Categorization of methods to solve MDPs based on sampling and bootstrap.

The family of methods using deep and full backups (no bootstrap, no sampling) is that of *exhaustive search*. These techniques require the full model of the MDP, and they plan by unfolding the entire state-action space and then collecting the information to the root node. They rarely are applicable in real-life applications with high-dimensional state-action spaces due to computational constraints.

2.6 Function Approximators

Using the methods presented above is challenging with complex MDPs that have a high dimensional state-action space, as they require an estimation of the value functions for each state or state-action pair. To solve this, it is possible to use a function approximator for the value functions, and update its parameters with the RL algorithm of choice, instead of directly updating the value functions. This allows to generalize over unseen states and state-action pairs, and scale the solution to even continuous state-action spaces. The parametrization for state-value functions and action-value functions is the following:

$$\hat{V}(s, \boldsymbol{w}) \approx V(s) \tag{2.28}$$

$$\hat{Q}(s, a, \boldsymbol{w}) \approx Q(s, a) \tag{2.29}$$

where $\boldsymbol{w}$ represents the parameters of the function approximator.

In general, these approximators must be differentiable, because the parameters are updated through gradient-based methods. In this section, *stochastic gradient descent* [14] will be used as gradient-based optimizer to illustrate the algorithms using function approximators, but other optimizers like *RMSProp* [15] and *Adam* [16] can be used as well.

2.6.1 SARSA

Approximate SARSA performs the same ϵ-greedy policy improvement step as exact SARSA, but uses $\hat{Q}(s, a, \boldsymbol{w})$ for approximate policy evaluation. Like in SL, we first define the loss function to minimize, which in this case is the *mean squared error* between the approximate action-value function and the true action-value function:

$$L(\boldsymbol{w}) = \mathbb{E}_\pi[(Q_\pi(s, a) - \hat{Q}(s, a, \boldsymbol{w})^2] \tag{2.30}$$

and then the optimization procedure to minimize the given loss function, which in this case is SGD:

$$\Delta\boldsymbol{w} = \alpha(Q_\pi(s, a) - \hat{Q}(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{Q}(s, a, \boldsymbol{w}) \tag{2.31}$$

where $\nabla_{\boldsymbol{w}}\hat{Q}(s, a, \boldsymbol{w})$ represents the gradient of the approximate action-value function with respect to the parameters.

Unlike SL, the true action-value function is not given as part of the dataset D, and must be substituted by a target. In the case of approximate SARSA(0), the target is the TD(0) target, so the update formula for the parameters becomes:

$$\Delta\boldsymbol{w} = \alpha(r_{t+1} + \gamma\hat{Q}(s_{t+1}, a_{t+1}, \boldsymbol{w}) - \hat{Q}(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{Q}(s, a, \boldsymbol{w}) \tag{2.32}$$

where the approximate action-value function is itself used as a proxy for the value of the next state-action pair (s_{t+1}, a_{t+1}). As for standard SARSA, one could use TD(n) and TD(λ) targets, leading to approximate SARSA(n) and approximate SARSA(λ).

2.6.2 Q-Learning

Approximate Q-learning shares the same loss function of approximate SARSA, specified in 2.31. The difference lies in the target used, that is not tied to the behavior policy but it is a greedy target related to the target policy:

$$\Delta \boldsymbol{w} = \alpha(r_{t+1} + \gamma \max_{a' \in A} \hat{Q}(s_{t+1}, a', \boldsymbol{w}) - \hat{Q}(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a, \boldsymbol{w}) \quad (2.33)$$

2.6.3 Deep Q Network

Deep Q Network (DQN) [17, 5] is an improvement of approximate Q-learning, aimed at stabilizing the algorithm to work with a nonlinear function approximator such as a *Convolutional neural network (CNN)* [18, 19]. CNNs are useful when learning is done from high dimensional inputs such as pixels from a videogame or frames from a camera sensor. The two main ideas behind DQNs are the *Experience Replay Buffer (ERB)* and the target network.

DQN uses two networks, a *prediction network* and a *target network*. The prediction network is the same used for approximate Q-learning, and its goal is to approximate the action-value function $\hat{Q}(s, a, \boldsymbol{w})$. In particular, DQN uses a variant of the prediction network where the input is a state, and the output is the Q-values for all the actions available in that state, such that in one forward-pass the agent can pick the greedy action with the highest Q-values, performing the policy improvement step.

The target network is architecturally identical to the prediction network, but has a different set of parameters that are kept frozen for a given number of steps, and then updated by copying the parameters from the prediction network. This is done to provide an update target that is stable like in SL, and not a varying one like in the approximate Q-learning algorithm, that is using a target given by the same network that is being constantly trained. The new update formula for the parameters becomes:

$$\Delta \boldsymbol{w} = \alpha(r_{t+1} + \gamma \max_{a' \in A} \hat{Q}(s_{t+1}, a', \boldsymbol{w}^-) - \hat{Q}(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a, \boldsymbol{w}) \quad (2.34)$$

where $\boldsymbol{w}^-$ is the set of parameters from the Target network.

The ERB is a set of experiences, collected in the past, either by the agent itself or by someone else, in the form of:

$$B = \{\langle s_0, a_0, r_0, s'_0 \rangle, \langle s_1, a_1, r_1, s'_1 \rangle, \ldots, \langle s_n, a_n, r_n, s'_n \rangle\} \quad (2.35)$$

where n is the size of the set. It is populated with a given strategy, the simplest being random, to form a collection of independent transitions. This alleviates the problem of using a dataset D that is not i.i.d. and would lead to unstable training, especially with non-linear function approximators like CNNs. When performing a training step on the network, the agent samples a mini-batch of experiences from the buffer, calculates the loss over those samples, and then updates the parameters $\boldsymbol{w}$ by performing an optimization step. The use of the ERB calls for an off-policy algorithm like Q-learning, as the experience is collected from a policy that is generally different from the current one.

DQN was used to obtain SotA performances on Atari games, and have since been extended with various improvements. Novel methods like Rainbow DQN [20], ApeX DQN [21] and the newest Agent57 [22], all make use of DQN as part of their architecture.

Chapter 3

Related Work

This chapter goes over most of the literature and past contributions that inspired this work. At first, Section 3.1 tackles generalization in the context of RL. It identifies the main contributions to the topic and then explains two families of solutions, *Hierarchical Reinforcement Learning (HRL)* and *reward shaping*, both used to design the proposed solution. Finally, Section 3.2 presents previous research on playtesting for game development, highlighting how it was approached in the past by third parties and at King.

3.1 Generalization in Reinforcement Learning

In the context of RL, generalization specifically refers to the ability of an agent that is trained in one environment to perform well in an environment with different characteristics, possibly never seen before during training. The focus of this thesis is on improving zero-shot generalization, which is measured as the performance of a model on the inference environment, without tuning its parameters there. In our case, where the training environment contains a set of levels and the inference environment contains different ones, we can say that an agent A generalizes better than an agent B if, after training both of them on the training environment, A displays higher performances on the inference environment. This means that A can transfer the knowledge learned during training at inference time better than what B does.

However, the ability of an RL agent to generalize has not always been the primary focus of research in the field. The most used benchmarks, like the *Arcade Learning Environment (ALE)* [23], report the performances of an agent over the same environment where it is trained on, thus not making a clear distinction between training, validation and testing like in the SL field.

The first systematic study of generalization is performed by Packer et al. [24], who propose a defined set of environments, metrics, and baselines to assess the abilities of agents to generalize. Surprisingly, they find out that architectures specifically developed to improve generalization fail to obtain better performances than simpler ones like vanilla *PPO* and *A2C*, two other RL algorithms.

This trend is well analyzed by Cobbe et al. [25], who propose a new benchmark that encourages the measurement of 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. This split allows us to assess how well the agent is generalizing, and the authors show that more levels in the training set lead to better performances in the test set. Finally, they try different techniques from the SL field such as deeper network architectures, *regularization*, *dropout*, and *batch normalization*, proving how they can decrease overfitting in RL tasks as well.

On the same line of research, Justesen et al. [26] also focus on a clear distinction between training and test environments, and integrate the idea of *curriculum learning*, in which every new episode in the training process features a level that is harder (easier) if the agent performed well (poorly) in the previous one. In particular, the levels are all created via *procedural content generation*, and this leads to more effective training and better generalization over unseen levels.

Lee et al. [27] try to address the issue of overfitting to high dimensional inputs like images by introducing random perturbations of the state-space, allowing an agent to generalize over unseen visual patterns in the test set. Their method outperforms techniques such as regularization and *data augmentation* for the same purpose.

A more interesting and recent result comes from the work of Farebrother et al. [28]. The authors first propose a test-bed to assess generalization on the ALE and use it to measure the abilities of a standard DQN algorithm, showing how it heavily overfits to the training set and is unable to perform well even with minor differences in the environment. Furthermore, they experiment with regularization and dropout in the network and report that these techniques improve the generalization abilities of an agent, both considering zero-shot performance and fine-tuning, achieved through pre-initialization of the weights with those learned during training.

3.1.1 Hierarchical Reinforcement Learning

The idea behind HRL is inspired by how humans learn to reuse primitive actions to achieve more complex goals, or macro-actions. Our process of decision making involves multiple levels of abstraction, or hierarchies, where we work towards a fixed goal by achieving smaller sub-goals that lead to accomplishing the final one. HRL is similar, having a hierarchy of agents with different goals, working at multiple levels of temporal abstractions from the others. The benefits of the hierarchy are multiple. Firstly, it leads to better exploration, as an agent can explore the environment driven by sub-goals, rather than through random techniques. Secondly, having different levels of temporal abstraction means that high-level behaviors can be learned by taking actions less frequently, thus having a faster training process. Finally, learning low-level behaviors leads to better knowledge transfer. A human that is learning to play Tennis does not need to also learn how to grab an object or move his arms again since it already acquired this knowledge in the past and can transfer it to this new task.

Florensa et al. [29] 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. [30] try to solve the problem of reusing RL modules with different reward scales, without having to linearly sum them. 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. Ideally, this architecture works better with on-policy algorithms, but the authors empirically prove that it also performs well with Q-Learning. The proposed architecture ends up being a hierarchical one, similar to [29].

Nachum et al. [31] conduct a thorough investigation of hierarchical models, analyzing what are the reasons behind the benefits of hierarchical architectures. They find out that the main one is improved exploration, followed to a lesser degree by temporally extended actions. Additionally, they propose two non-hierarchical methods that share the same benefits of hierarchical ones. The first one consists in having multiple policies focused on different goals, and just randomly sampling one every τ timesteps to use in the environment.

The second one involves two agents, one focused on maximizing environmental rewards, and one focused on reaching hand-crafted goals, where the first one is the default choice for exploitation, while the second one is used for exploration in the environment.

Feudal RL

The first hierarchical architecture is proposed by Dayan et al. [32] with the *Feudal* framework. Here, managers learn to give goals to sub-managers at lower levels in the hierarchy, who will in turn learn how to achieve them. The only communication between managers and sub-managers is done through goal assignment, where sub-managers are rewarded for achieving a goal (reward hiding). Each level in the hierarchy only needs to observe the subset of the state-space that is useful for it, without the need to know what happens in the other layers (information hiding). This first version of feudal learning is not guaranteed to converge to an optimal policy.

Vezhnevets et al. [33] make use of the Feudal framework in an architecture called *FeUdal networks*, featuring two layers of hierarchy. A manager picks explicit sub-goals in a latent space, that is itself learned by the manager, and provides them to a worker who tries to achieve them. There is no gradient sharing between manager and worker, but only communication of goals.

Options

The family of architectures based on *options* stems from the work of Sutton et al. [34]. An option represents a temporally extended action, and it can be formally defined as the triple $o =< I_o, \pi_o, \beta_o >$, where:

- $I_o \subseteq S$ is the initiation set, i.e., the set of states where the option is available and can be chosen by the agent.
- $\pi_o : S \times A \to [0, 1]$ is the policy used to choose actions after option o is selected.
- $\beta_o : S^+ \to [0, 1]$ is the termination condition, i.e., the probability distribution over states, indicating when the option ends.

With this framework, a high-level policy-over-options observers the environment, picks options (sub-policies) and runs until termination, while a low-level sub-policy observes the environment, picks primitive actions that are available, and runs until the termination condition is met (or the episode ends).

This formalization extends actions to high-level skills that can be applied to different environments. A traditional formalization with primitive actions, for an agent that has to escape from a house, would include actions like moving North, West, South, or East. A more efficient formalization, using options, allows the same agent to use high-level actions like "going to the hallway" or "leaving the room".

Kulkarni et al. [35] extend the options framework and propose the *h-DQN* model, where a meta-controller receives rewards from the environment and picks high-level goals to provide to a low-level controller, which in turn takes primitive actions until it reaches the goal. The low-level controller follows the idea of intrinsic motivation, where an agent rewards itself for exploring the environment. Goals can be expressed in a versatile way, even as a function of states.

Bacon et al. [36] propose the *option-critic* model, which is a variant of the action-critic algorithms. The authors extend the policy-gradient theorem to options and come up with an end-to-end architecture where options can be learned rather than pre-specified. Here, a manager receives the gradients directly from the workers, unlike in feudal learning.

3.1.2 Reward Shaping

Many RL environments provide an agent with rewards that are too sparse and/or delayed, making it difficult for an algorithm to identify how past actions impact the final outcome, as the reinforcement process can happen long after the action was taken, and with few rewards. Working in such a setting can lead to a very slow learning process, often too time demanding to be useful for real-world applications. A way to approach this is to give more frequent rewards (*intrinsic*) to the agent, other than those given by the environment (*extrinsic*), to guide the learning process and make it faster. This category of solutions falls under the name of *reward shaping*, where the reward function is shaped either manually, by an expert, or through intrinsic motivation, by the agent itself.

Potential-Based

Manually shaping the reward function by giving additional rewards to an agent can lead to undesired behaviors. Randlov et al. [37] craft an experiment where an agent that has to learn to cycle to a goal is given positive rewards when moving towards the goal. This additional reward is exploited by the agent by performing circles on the spot, without ever driving to the goal, as it gains infinite rewards in the loop every time it is facing the destination.

To address the problem, Ng [38] proposes a reward function that achieves policy invariance by assigning potentials to states, and giving rewards based on the difference of potentials between the new state and the previous one, avoiding loops. The author formally proves that shaping the original reward with a *potential-based* intrinsic reward is a necessary and sufficient condition to still achieve the optimal policy. The new Q-learning update becomes:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + F(s,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big] \tag{3.1}$$

where the additional reward $F(s,s')$ is defined as:

$$F(s,s') = \gamma\Phi(s') - \phi(s) \tag{3.2}$$

Devlin et al. [39] build upon Ng's work and make the potential a function of time as well, thus dynamic. They also prove how the new function still achieves the optimal policy, even if it never converges. In this case, F is defined as:

$$F(s,t,s',t') = \gamma\Phi(s',t') - \phi(s,t) \tag{3.3}$$

One flaw of Ng's approach is that the potential does not take into account the action taken, as it only depends on the visited states. Wiewiora et al. [40] propose an extension where the potential is now a function of the full transition from one state to the other, so it is possible to guide an agent by giving it advises on the best action to take in every state. The authors propose two ways of defining the new function F. The first way is called *look-ahead*:

$$F(s,a,s',a') = \gamma\Phi(s',a') - \phi(s,a) \tag{3.4}$$

where a' is chosen according to the learning rule (e.g. greedy selection in Q-learning). To recover the optimal policy, action selection in a state should be biased by adding the potential of the transition. The new policy becomes:

$$\pi^b(s) = \operatorname*{argmax}_a \Big[Q(s,a) + \Phi(s,a)\Big] \tag{3.5}$$

With *look-back* advice, we only need to change the reward function F, which now takes into account the previous transition as opposed to the future one:

$$F(s_t, a_t, s_{t-1}, a_{t-1}) = \Phi(s_t, a_t) - \gamma^{-1}\Phi(s_{t-1}, a_{t-1}) \tag{3.6}$$

Building upon the previous work, Harutyunyan et al. [41] propose a method to use any kind of reward function to shape the original one, while maintaining the same guarantees of potential-based shaping. The method involves learning a new state-action value function $\Phi(s, a)$ that is trained using a reward R^Φ equivalent to the negation of the reward function R^Ψ provided by the expert. In formulas:

$$R^\Phi = -R^\Psi \tag{3.7}$$

$$\Phi_{t+1}(s, a) = \Phi_t(s, a) + \beta_t \delta_t^\Phi \tag{3.8}$$

$$\delta_t^\Phi = r_{t+1}^\Phi + \gamma\Phi_t(s_{t+1}, a_{t+1}) - \Phi_t(s_t, a_t) \tag{3.9}$$

and using $\Phi(s, a)$ as a potential allows to achieve policy invariance and thus convergence to the optimal policy. In fact, the followng equality holds:

$$F(s, a) = \gamma\Phi(s', a') - \Phi(s, a) = R^\Psi(s, a) \tag{3.10}$$

Curiosity-Driven

The idea of *curiosity-driven* learning is presented by Chentanez et al. [42]. The authors let an agent explore a playground where it can learn different skills, and provide it with a reward based on the prediction error: the more novel a state is, the higher the reward given. This form of purely intrinsic reward allows the agent to first learn easy behaviors, which are repeated many times (decreasing the reward for each repetition), and then more complex behaviors as the time goes on. Actions are modeled through the options framework for more expressiveness.

More research followed on curiosity-driven learning. In particular, a thorough study by Burda et al. [43] compares different SotA methods giving intrinsic rewards based on prediction error. An interesting result is their thought experiment, later confirmed empirically, to prove that these methods do not work with highly stochastic environments, as the agent would only take actions with the highest entropy, leading to sub-optimal behavior. This is especially important for a game like CCFS, where an agent would focus only on exploiting the high stochasticity of the transitions instead of learning the optimal policy.

Linear methods

A straightforward way to incorporate relevant events into the reward function is through linear approaches. Lample et al. [44] make use of this idea to train an agent in the *VizDoom* environment, where additional positive rewards are given for events like picking up ammunition or walking in the map, while 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. [45] and consists in decomposing a reward function in multiple ones, learning a value function for each reward stream. Ideally, since each stream is different, the various policies can be trained using the subset of the entire state-space that contains relevant information for that stream, decreasing sample complexity. The architecture can be thought of as a single network having multiple heads, where each head is trained with a different reward stream, but the body can be shared among the heads. Action selection is performed by summing the Q-values of the different policies and picking the action with the highest resulting Q-value. Theoretically, the method works better with an on-policy algorithm, as an off-policy one would update the Q-values of each policy as if the next action was chosen by that policy, while it is not true in this case.

The same concept of multi-head network is presented by Burda et al. [46], where the additional intrinsic rewards are given as an exploration bonus. The combination of intrinsic and extrinsic rewards is performed by learning multiple value functions since the linearity of the returns with respect to the rewards allows to simply sum the two value functions represented by the two heads. In formulas, if V_e is the value function related to the extrinsic rewards, and V_i the one related to the intrinsic rewards, the final value function is $V = V_e + V_i$. This decomposition also allows training an agent with different discount factors, as they are factored into the respective value functions.

Other approaches

An approach similar to Curiosity-Driven Learning is proposed by Jaderberg et al. [47], where an agent tries to maximize more reward functions at the same time, using unsupervised approaches. The model performs additional control tasks like learning to change pixels or network features, and it shares the same convolutional and *LSTM* layers with the base policy. Moreover, the model tries to predict additional quantities like the immediate reward given past states, and the learned weights are shared with the LSTM of the main policy.

An interesting intrinsic-only method is proposed by Justensen et al. [7], where rewards are given using an automated heuristic. An expert first defines a series of events that are relevant for the task at hand, and the agent is rewarded for achieving those events: the more it achieves them, the less reward it will gain. Formally, given a vector of event occurrences x, where x_i is the number of times event i occurred during a timestep t, the reward given to the agent in t is:

$$R_t(x) = \sum_{i=1}^{|x|} x_i \frac{1}{max(\mu_t(\epsilon_i), \tau)} \tag{3.11}$$

where $\mu_t(\epsilon_i)$ is the temporal episodic mean occurrence of ϵ_i at time t, which represents how often the event occurs every episode on average, while τ is a constant to avoid division by zero. 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 hard to define with a manual reward without introducing human bias.

3.2 Playtesting

Silva et al. [48] attempt to find an AI that could master the board game called *Ticket to Ride*. Their experiments with search techniques like *A-Star* [49] and *Monte Carlo tree search (MCTS)* [50] are not successful due to the large state-space of the game and its partial observability, so the authors opt for an heuristic-based approach. They develop four agents, each one based on a strategy commonly accepted by the game community as strong, and test them on different maps. Their discoveries help to find similarities between maps and which strategies are desirable for certain maps, thus increasing the common knowledge on the game. Moreover, the agents discover states that are not covered by the game rules, effectively representing bugs in the rules.

Ariyurek et al. [51] propose a method to employ automated testers to discover bugs in games. They use two types of agents, based either on RL or MCTS and train them using one of two different approaches. The first one uses test goals from game scenarios modified to reward unintended transitions, while the second one uses *Inverse RL* to learn a reward function from human play-testers, such that the agents could imitate their behaviors in finding bugs. The performance of the agents is on par with that of human testers, and the final results highlight the similarity between the RL agents and human behaviors.

Borovikov et al. [52, 53] propose a learning and planning framework to assist developers in building and testing their games. The first approach is based on model-free RL but is discarded due to the computational cost of training DQN-based agents to achieve the same performance of human testers on even short episodes. The authors improve over the first method by using Imitation Learning instead of RL, allowing the agent to learn from expert demonstrations. Their final architecture consists of a Supervised model trained on a data set composed of expert demonstrations augmented with artificial ones generated from a Markov model that is imitating a human player. At any point in time, a game designer can take control of the model by injecting more demonstrations samples to learn from.

A more recent paper by Shin et al. [54] tries to solve play-testing for *Jewels Star Story*, a Match 3 game that is very similar to CCFS. The authors define a set of strategic plays, representing meaningful heuristic strategies that a player can choose to follow, and train an RL agent using A2C to learn which strategy to use on the current board. Their results underline that this method performs better than rule-based agents, that are fully based on heuristics, and close to human performances on some levels. The authors also conclude that their results are not extendable to all levels with different goals, as the features may vary a lot from those used during training, and thus not generalize well.

King

This thesis builds on the work done in the past by researchers at King, who tried to automate and improve playtesting for level designers.

Poromaa [55] develops a general MCTS algorithm to predict the *Average Human Success Rate (AHSR)* on the levels of CCS. His results report more accurate estimations of the AHSR than SotA methods based on heuristics, and the author argues that MCTS can be more accurate than a human playtester, which is limited by the time and number of experiments it can run. However, the results are relevant on the first levels used for testing, while the accuracy over harder levels is worse. Finally, the author underlines the limitations of MCTS in an exhaustive search of the state-space, which is deemed infeasible due to the computational load of the algorithm.

Gudmundsson et al. [1] approach the problem of estimating AHSR by using deep learning from available player data. They train a CNN network to predict the most likely action to be taken in a given board state based on the demonstration data collected from real players on 2150 different levels. The success rate of the CNN bot, measured as the number of wins over the total number of attempts, is then fed into a binomial regression model that predicts the success rate of a human (AHSR). Their results over 200 test levels report stronger prediction accuracy and execution efficiency than the MCTS approach of [55]. Nonetheless, they are limited by the need to retrain the prediction model when new game elements are added, as well as collecting new human data for the training set.

To address the previous points, RL solutions are developed and tested by Fischer [56] and Karnsund [2]. The former implement a version of PPO, a policy-based model-free agent, while the latter experiment with DQN and its variants, namely *Double DQN*, *Dueling DQN* and *Prioritized Experience Replay*. In both cases, the research is scoped down on a subset of levels and one objective in CCFS. This thesis builds directly upon the work of [2], making use of DQN as its base agent and Q-learning as the underlying algorithm. In the experiments, the trained agents fail to get rid of Blockers in the available number of moves and thus have a low overall win rate compared to humans. Moreover, the agents struggle to transfer their knowledge to unseen levels, demonstrating poor generalization abilities.

Chapter 4

Problem Setting

This chapter provides the reader with crucial details needed to understand the CCFS game and its definition as an RL environment. This is a necessary step before diving into the remaining chapters, as the proposed solution is strictly intertwined with how the environment is defined. Section 4.1 explains the mechanics of the game and formalizes it with an MDP. Then, Section 4.2 goes through the implementation of the environment. It covers the encoding of the state and action space, the reward functions used in previous works, and how to keep track of meaningful events such as the creation of Special candies.

4.1 Candy Crush Friends Saga

As mentioned in Section 1.6, the game used in this thesis is CCFS, the latest released game in the Candy franchise. Each level is completed if its objective is reached within the number of available moves. Currently, there are five different types of objectives in the game: *Free the Animals*, *Free the Octopuses*, *Dunk the Cookies*, *Fill the Empty Hearts*, and *Spread the Jam*. To ensure consistency with the baseline [2], the objective used in this thesis is Spread the Jam, so the features introduced in this section are only the ones available in these levels.

The game board is a grid of 9×9 tiles that contains either Regular candies, Special candies, or Blockers (defined below). 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. These are then eliminated from the board and replaced with the candies above them, or with random candies if the action involves the top row.

The behavior of random candies falling from the top is regulated by a seed number, which also determines the candies that are randomly present at the beginning of the level. A player that plays the same level twice, with two different seeds, will have a different game experience: the same move in the same board configuration will lead to a different game state due to the involved randomness.

Candies

Regular candies are the most common type of candies that are present in the game. There are seven types of Regular candies, each one with a distinct color: blue, cyan, green, orange, purple, red, and yellow. 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 five types of Special candies: *Striped* candies, *Wrapped candies*, *Color Bombs*, *Coloring* candies and *Fishes*. A Special candy can be used by swapping it with another Special candy, without necessarily having to match three or more candies of the same color together. The effect of a Special candy is triggered whenever it is part of a match. In particular:

- Striped candies are created by matching four candies in a horizontal line or vertical line. Depending on the direction of the match, the candy can either have horizontal stripes or vertical stripes. When a striped candy is involved in a match, it creates a vertical or horizontal blast (according to its stripe) that clears all candies in that line and is stopped by the first Blocker it encounters.

- Wrapped candies are created by matching five candies in an "L" or "T" shape. When a Wrapped candy is involved in a match, it first clears all the other candies in a 3×3 space, and then drops down and repeats the effect. The explosion damages Blockers.

- Color Bombs are created by matching five candies in a straight line. When a Color Bomb is swapped with another candy, all the candies of the same color of the latter will be cleared from the board.

- Fishes are created by matching four candies in a square. When a Fish is involved in a match, it randomly targets another tile on the board, and either clears the candy in that tile or damages the Blocker in that tile if present.

- Coloring candies are created by matching six or more candies in a straight line. When a Coloring Candy is swapped with another candy, it makes all the other candies on the board of the same color as the latter.

Blockers

Blockers 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 taken out of the board, and the tile is said to be *free*. In jam levels there are four main families of Blockers:

- *Cupcakes* occupy one tile on the board and block the flow of candies, as well as their effects, from going beyond them. Cupcakes can have up to six layers. For instance, if a Cupcake with six layers occupies a tile, it needs to be hit six times to be completely removed from the board.

- *Liquorice Chains* are similar to Cupcakes, blocking candies, and effects from traversing their tile. They can have up to four layers, but when one of them is removed from the board, all the others forming a chain with it are also removed, regardless of their layers.

- *Liquorice Locks* are placed on top of a candy and prevent the player from swapping the candy. They have one layer and are removed when the candy under the Lock is involved in a match with candies of the same colors.

- *Liquorice Swirls* are the only type of moving Blocker, as they fall down like other candies and do not have a fixed position. They can also be swapped with a candy, block the flow of candies, and prevent effects from going through. They only have one layer and are removed by making a match on the neighboring tiles.

Characters

Characters, or *champions*, are entities that can assist a player in completing a level. One of them is selected before the level starts, and its effect is triggered once the player makes a certain number of matches that involve the color of that character. Currently, there are five characters in the game: *Tiffy*, *Yeti*, *Nutcracker*, *Odus* and *Misty*. To ensure comparable results with the baseline [2], Tiffy is the character used in all the levels in the experiments. Tiffy's effect is triggered once ten red candies are collected and consists of turning three random candies on the board into three red fishes.

Special candies play an important role in the solution proposed in this thesis. Using a champion with respect to another one might introduce bias in that sense, so ideally one would conduct the research without selecting any at the beginning of the level. However, this is not feasible to implement in the source code in the time given, so the choice falls on the same one used in the previous works.

Spread The Jam

In Spread the Jam levels, the objective is to cover the entire board with jam. At the beginning of the level, some tiles already contain jam as a background. When a match involving a tile with jam is performed, the jam is spread to the other tiles involved in the match. These levels require strategic planning, as spreading the jam all over the board within the move limit is not straightforward. In particular, Blockers prevent a player from performing matches on the board, so the jam cannot be spread on the tiles with Blockers. Unlike other objectives, in jam levels all Blockers must be taken out of the board, and a good approach is to focus on doing that initially, so the tiles are then free of Blockers, and jam can be spread more easily.

4.2 Environment Implementation

The environment was built by King engineers and acts as a medium between the game and the RL agents. The game exposes two endpoints to the environment, one to reset the state and start the level passed as a parameter, and one to perform the action passed as parameter. Both endpoints return an encoded representation of the current game board after the reset or action is performed.

The environment exposes a uniform interface to the RL agent that is the same as Gym's [57], allowing custom implementation of an agent to be used in the future without compatibility issues with the environment. It receives the raw state from the game and transforms it by applying an encoding detailed in this section. Moreover, it exposes two endpoints, one to reset the state and one to take an action, using a second type of encoding. Both endpoints return a reward for the move taken, the new state representation, a signal that indicates if the level is finished, and additional information. As described in this section, the reward function is part of the environment, so defining an intrinsic reward function involves defining a new environment.

The environment receives a raw representation of the game board from the game itself, without information on what happens as a consequence of the action of an agent. Key information such as what Special candies have been created and what Blockers have been damaged are missing in this representation. Since these types of events are crucial for the solution, we decided to implement an event tracker in the environment that serves this purpose. This routine receives the initial board state, the action taken, and the following board state, and follows a best-effort approach to determine the events caused by the action.

Determining precisely the consequences of a game action is impossible, as the game involves randomness due to candies falling from the top to fill up the board, and we cannot replicate it outside of the game. These random candies filling the board can be immediately involved in another match, so multiple matches can be caused by a single action. However, the environment receives the board state after all effects are applied, therefore, it is only possible to track the events as a difference between the board state before the move and after it. For instance, a Special candy may be indirectly activated by the random effect, and it would wrongly appear to the environment as if this was done by the agent, which would be rewarded for it. This noise in the measurements cannot be removed and might impact the performances of some reward functions proposed in the next chapter.

MDP

CCFS is a complex environment for RL. The state-action space is discrete and finite, yet very high dimensional. Independently of the encoding, the number of possible board configurations is exponential with respect to the number of game features. As a consequence, it is highly unlikely that an agent will face the same board configuration twice, and this burden on generalization has to be carried by the function approximator used. The stochasticity of the transition and reward matrices makes it challenging to learn the correlation between an action and its consequences, as random candies falling from the top cause the same action in the same state to lead to different board configurations, if different game seeds are involved. As a consequence, exploration is harder than other popular environments, and the number of episodes required to learn an effective policy grows with it. In this thesis, an episode is a full play on a level, finishing with a win or a loss. To summarize these characteristics, we propose to model the environment of the game through the following MDP:

- Time horizon T: finite.

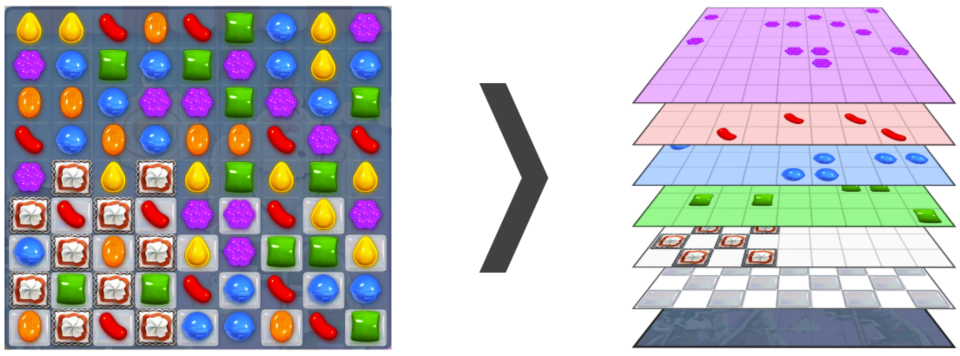
A player knows beforehand how long the game is going to be, given the move limit.

- Set of states S: finite.
  The number of possible combinations of board states is finite and depends on the encoding.
- Set of actions A: finite.
  The number of available actions in a given state is finite and depends on the encoding.
- Matrix P: partially stochastic.
  The transition from a state to another is affected by randomness from the candies falling.
- Matrix R: partially stochastic.
  The reward for a given action is affected by randomness from the candies falling.
- Vector μ: stochastic.
  The initial state distribution is random and depends on the game seed.
- Discount factor γ: undetermined.
  Different levels have different time horizons, so γ is treated as a hyperparameter.

State Space

The idea behind the encoding of the state space is to use a one-hot binary representation for each game feature available. This type of encoding is adopted by [1], and is also used for the state representation of AlphaZero [58] and OpenAI Five [59]. The advantage of a one-hot representation in a board game is that it makes learning much easier compared to using raw pixels. There is no risk of learning patterns from meaningless components like the background color, so the network layers can instead focus on the important features directly, without having to learn how to abstract from them. In this case, feature engineering is an important preprocessing step that speeds up and improves training.

Another work at King [1] represent the state as a 3D grid of $9 \times 9 \times M$ dimensions, where M is the number of layers. This number depends on the Candy game taken into consideration, as different games have different features. In CCS, the authors adopt $M = 102$. In CCFS, [2] uses $M = 193$. A representation of the encoding for CCS is provided in Figure 4.1.



Adopted from [1].

Figure 4.1: One-hot encoding of the board as a $9 \times 9 \times M$ 3D matrix for CCS. The same idea applies to CCFS.

In [2], given the focus on only a subset of jam levels, there is no need to use the entire $9 \times 9 \times 193$ encoding of CCFS. Most of those layers will hold zeros, because only a subset of the features is available in their levels, so the author decides to adopt 27 feature planes.

In this thesis, given the focus on generalization, we opt to make use of a larger number of levels compared to [2], all having Spread the Jam as the game objective. The levels are selected from a pool that is diverse enough to cover all the game features available in these levels. During training, the parameters of the network used will fit every possible feature, so that an agent will be able to be tested and used on all the jam levels, even future ones. An exception is done for *entity spawners*, which are features that create new candies and Blockers on the board in addition to the ones present at the beginning of the level. This group of game features is excluded from the thesis scope, due to some engineering constraints in the game build used. The full list of levels, together with the specific reasoning behind the choice of each of them, is provided in Chapter 7. In total, these cover a feature space with 32 variables: 30 binary ones coming from the raw encoding of the game, and two real ones that are handcrafted. The explanation for the choice of each layer is provided next.

Layers

Layers 0-5 encode the presence of one of the six Special candies on a tile. Layers 6-12 encode the color of the candy in the tile, out of the seven available colors. A Regular candy in position i, j on the board is encoded with 0s in the feature planes of the Special candies, and a 1 in position i, j of the feature plane related to the color of that candy. A Special candy in position i, j is instead encoded both by a 1 in its corresponding feature plane, and a 1 in the feature plane related to its color.

Layers 13-18 encode the presence of one of the six Cupcakes on that tile. For instance, if a Cupcake with two layers is in position i, j, then the 14th state layer, corresponding to cupcakes with two layers, will have a 1 in position i, j. When that Cupcake is hit and remains with one layer, there will be a 1 in position i, j of the 13th state layer, and a zero in position i, j of the 14th state layer. Layers 19-22 represent Liquorice Links, which behave like Cupcakes. Layers 23 represents Liquorice Locks. Layers 24 represents Liquorice Swirls.

Layer 25 indicates if the tile contains jam, so it is related to the objective of the level.

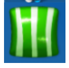
Layer 26 indicates if the tile contains a *portal*: a feature of the game that teleports candies from its entrance to its exit, which is represented by another portal on the board.

Layer 27 is the bias of the network, and it's a 9×9 layer full of 1.

Layer 28 is the *void* layer, and it is used to indicate the true shape of the board, as some levels do not have a full 9×9 board playable, but might have a subset such as a 7×8 board.

Layer 29 is associated with *cannons*, which create candies randomly to fill the board after a match. As mentioned before, in the scope of this thesis we only use levels with cannons that create Regular candies.

The last two layers are handcrafted and contain fractional numbers. Layer 30 is used to indicate gravity, i.e., the direction in which candies move after a match, as in some levels they might not fall downwards but in other directions. Layer 31 contains the number of moves left as a fraction of the initially available moves. This kind of encoding for scalars is redundant, as they are represented as a 9×9 matrix that contains the same value everywhere. A better encoding would use two input sources, one represented as a matrix and one as scalar features. However, this requires creating an ad-hoc layer to merge the two after feature extraction, and we deem it unnecessary for only two scalar features, so we leave it for future work. A concise representation of the adopted feature space is provided in Table 4.1.

Layer	Feature	Format	Render
Special candies			
0	Color Bomb	Binary	
1	Vertical Striped	Binary	

2	Horizontal Striped	Binary	
3	Wrapped	Binary	
4	Coloring Candy	Binary	
5	Fish	Binary	
	Colors		
6	Cyan	Binary	
7	Yellow	Binary	
8	Orange	Binary	
9	Green	Binary	
10	Red	Binary	
11	Purple	Binary	
12	Blue	Binary	
	Blockers		
13	Cupcake_0	Binary	
14	Cupcake_1	Binary	
15	Cupcake_2	Binary	
16	Cupcake_3	Binary	
17	Cupcake_4	Binary	
18	Cupcake_5	Binary	
19	Chainx_1	Binary	
20	Chainx_2	Binary	
21	Chainx_3	Binary	
22	Chainx_4	Binary	
23	Lock	Binary	
24	Swirl	Binary	
	Objective		
25	Jam	Binary	

	Others		
26	Portal	Binary	
27	Ones	Binary	
28	Void	Binary	
29	Cannon	Binary	
30	Gravity	$[0, 1]$	
31	Moves left	$[0, 1]$	

Table 4.1: A summary of the 32 layers used in the thesis, including their indexes, names, data types and renders.

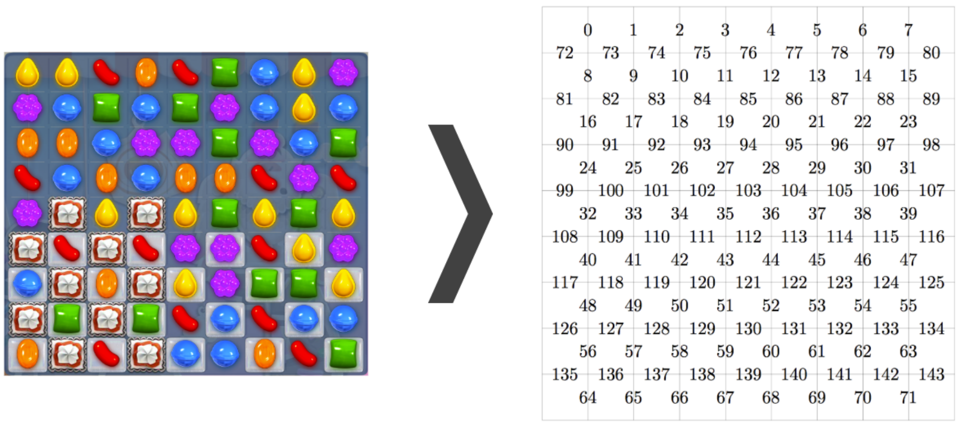
Action Space

The same encoding of the state space applies to the action space. Following the idea of [1], actions are represented by one-hot encoding the edge between the two candies that are swapped, for a total of 144 possible swaps on the board. This encoding does not take into account the direction of the swap, and this sometimes matters in the game and might lead to different results. However, as the authors notice, this is not worth the cost of doubling the number of actions to 288 to also include the direction of the swap, so it left for future improvements. For an RL agent, 144 discrete actions can already be considered high dimensional, compared to many games in the ALE where the actions available are four. A representation of the encoding is provided in Figure 4.2.

At any point in time, only a few candies can be swapped on the board, i.e., those whose swap leads to a successful match. We prevent an agent to choose illegal actions by marking their Q-values as illegal both before selecting an action and when performing a Q-learning step.

Illegal Actions

The main logical adjustment we made to the action space consists in changing the way illegal actions are treated. A legal action is one that leads to a successful match, with either three or more candies of the same color in a row or forming a shape that follows the rules of Special candies. At a given time t, only a subset of all the 144 actions can be made on the board, and these are called legal actions. A common approach in the literature is to reward an agent negatively and bring it back to the same state when it tries to perform an illegal action so that it will learn to take only legal ones. In CCFS, this is unfeasible because of the high dimension of the action space, and due to only



Source: Tech at King, Medium [1]

Figure 4.2: One-hot encoding of the actions as 144 indexes marking the edge between the candies to be swapped.

an extremely small subset of the 144 actions being legal in each state. Trying to teach the agent to distinguish between legal and illegal actions would be a complex RL problem itself, so it is discarded. To address this, in [2], the network architecture is fed with a list of legal actions, and the Q-values of illegal ones are set to a negative threshold by the head of the network. In this way, when an agent performs the $\max$ operation, it only focuses on the actions that have a positive Q-value, i.e., only the legal ones. However, this assumption only holds for certain types of reward functions, and it might be false for other ones that also give negative rewards, for instance when losing. This can make the training process heavily unstable, as incorrect transitions would be saved in the ERB. We propose a simple adjustment that consists of dealing with illegal actions downstream, making the network not responsible for this task. It is now the responsibility of the agent itself, using information obtained from the environment, to perform ϵ-greedy action selection only over the legal actions, masking the illegal ones.

Reward Function

The environment was created by the researchers at King, who also designed the reward functions. In this work, a function that is directly rewarding an agent for pursuing the objective of a level is called *extrinsic*. In this section,

[1]https://medium.com/techking/human-like-playtesting-with-deep-learning-92adafffe921

we present examples of extrinsic reward functions for CCFS designed at King in previous works. On the other hand, a function that rewards an agent for pursuing other goals that are not the one of the environment is called *intrinsic*. This type of reward is used in our solution to teach basic skills and is part of Chapter 5.

Sparse Reward

Sparse rewards are one of the most adopted form of reward functions due to their simplicity. They do not introduce any form of human bias, because they only give a reward when the game is won or lost, and not in between. For instance, they are widely used in the ALE. They can be defined as follows:

$$r(s_t, a_t) = \begin{cases} 0, & s_{t+1} \neq s_{win} \; and \; s_{t+1} \neq s_{lose} \\ 1, & s_{t+1} = s_{win} \\ -1, & s_{t+1} = s_{lose} \end{cases} \tag{4.1}$$

$$r(s_t, a_t) = \begin{cases} 0, & s_{t+1} \neq s_{win} \\ 1, & s_{t+1} = s_{win} \end{cases} \tag{4.2}$$

according to whether we penalize a losing state (4.1) or not (4.2). These functions can be used interchangeably for different level objectives, as they are not related to jam in any way. Of course, different environments need to define when a state is marked as winning or losing, but that is the only difference between them.

The main issue with these functions is sparsity. Learning an optimal policy requires a much higher number of episodes than with a non-sparse (dense) reward function, because the reward is given at the end of the episode, and needs to be backpropagated to every action that was involved in achieving it. With DQN, that is based on TD(0) methods, the reward is backpropagated one episode at a time, so the process is very slow. Moreover, distinguishing between actions that contribute to a win and those that did not is hard, as the only information that an agent gets is if it won or lost.

From the experiments of [2] and [56], we observe that the sparse reward functions have poor performance in CCFS compared to the dense ones. We claim that these types of rewards are not a good choice for stochastic environments, where the effect of an action is impacted by randomness in such a way that it becomes difficult to assign the correct credits. Instead, the approach that should be followed is to use a dense function, which rewards every action for its direct contribution. A final issue that we identify with these rewards is related to low win rates. Some levels in CCFS are designed to be really hard and have an extremely low win rate as a consequence, as low as 5% for human players. In this situation, a random policy exhibits a win rate of 0%. This makes the learning process even harder than it normally is, as the winning signal would be unobtainable at the beginning of the learning process, where the agent is effectively playing randomly.

Dense Reward

Dense reward functions address the problem of sparsity by giving more frequent rewards to the agent, whenever it progresses towards the objective to reach. In the context of CCFS, [2] and [56] propose two different dense functions, directly related to jam levels. The idea behind them is to give more frequent rewards to the agent, whenever it progresses towards covering the board with jam.

The first one is *Delta Jam*, and gives a reward for an action that is equal to the number j of new tiles covered in jam, normalized by the total number B of tiles on the board. In formulas:

$$r(s_t, a_t) = \begin{cases} \frac{\Delta j}{B}, & s_{t+1} \neq s_{win} \; and \; s_{t+1} \neq s_{lose} \\ 1, & s_{t+1} = s_{win} \\ -1, & s_{t+1} = s_{lose} \end{cases} \tag{4.3}$$

One of the main points in favor of this reward design is that it is normalized to 1. The total amount of reward that an agent can accumulate in an episode is 1, disregarding the end game rewards. This property is desirable if the agent is going to be trained on multiple levels at the same time, since it is independent of any scale related to the specific level.

Despite the normalization properties, Delta Jam performs worse than another alternative, *Progressive Jam (PJ)*. In this case, 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 in jam at that moment, normalized by the total number B of tiles on the board. In formulas:

$$r(s_t, a_t) = \begin{cases} \frac{J}{B}, & \Delta j > 0 \\ 0, & \Delta j = 0 \end{cases} \tag{4.4}$$

The immediate reward that the agent receives is always less or equal to 1 (when a winning move is performed), but the accumulated reward over an episode depends on the particular level that is being used. This reward is denser than the previous one and results show that it allows us to cover the last few remaining tiles on the board that did not have jam yet. In [2] and [56], PJ is the reward that allows achieving the best experimental results, so it is the default one for the environment. Therefore, in this thesis, we choose to use PJ as the extrinsic reward.

Chapter 5

Skill Learning

This chapter introduces the reader to the first part of the solution, related to learning reusable skills. As explained in Chapter 1, the idea behind the overall solution is to imitate the behavior of human players during their learning curve in the game. Section 5.1 presents the detailed motivation behind this first part of the solution. In this thesis, the focus is on learning three skills: getting rid of Blockers, creating Special candies, and using Special candies by matching them with other candies. Reward functions related to Blockers are presented in Section 5.2 and the ones related to creating and using Special candies are introduced in Section 5.3. The network architecture is shared by both families of policies and is illustrated in Section 5.4. Finally, Section 5.5 goes over the different training methods proposed to improve generalization.

5.1 Motivation

The skills presented in this section are just a few examples of the ones learned while progressing in the game. As a new player progresses through the levels in CCFS, he/she learns the wide range impact that Special candies can have, and will tend to create and use them more frequently to reach the level objective. This involves recognizing specific board configurations from which Special candies can be created and making the right moves to make that happen. Also, as the player learns the game, he/she understand the importance of triggering the character's ability and will start to play around it more. For instance, if Tiffy is the selected character, it is usually better to collect more red candies to activate its effect. Moreover, when dealing with jam levels specifically, the player will immediately recognize the importance of getting rid of Blockers on the board as quickly as possible, such that it can then spread the jam more easily and win. Learning how to get rid of them can also help the player in levels that have different objectives.

The aforementioned skills are not directly related to achieving the goal of the level, so it is harder to learn them by giving rewards for spreading jam. Nonetheless, these skills enable the player to complete most levels. This first part of the solution deals with exploiting intrinsic motivation, in the form of rewards that are 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 getting rid of Blockers, creating Special Candies, and using them. Using the character's ability is out of the scope of this thesis, and is left for future work. We select a candidate function for each skill to learn out of several different ones that are tested, based on their performances. The goal is to obtain policies that can retain a good amount of skill proficiency in unseen levels.

To the best of our knowledge, there is only one other recently published research on Match-3 games following the same idea and was developed independently of this thesis. In their work, Shin et al. [54] recognize the potential of skill-based learning, and try to teach an agent a series of skills relevant for the game at hand. However, they do this only through handcrafted heuristics, and not via RL. Instead, they use RL to train an agent that can choose between those heuristics, and their methodology is not meant to be scaled across levels, as it is presented in this chapter. In this thesis, the focus is on using RL to also perform skill learning, followed by finding the best way to combine them effectively, as explained in the next chapter.

5.2 Blockers

As explained in Chapter 4, there are different types of Blockers available in jam levels, each one with its characteristics. Different levels can contain different types of Blockers, in various quantities, so defining a reward function that can work across multiple levels and generalize to new ones is not trivial. Two approaches are followed in this part: (i) *Damaging Blockers*, which consists of rewarding agents for every Blocker they damage, regardless of the type of Blockers, and (ii) *Freeing Tiles*, which is driven by the motivation behind learning to get rid of Blockers to obtain a free board. Rewards are given whenever there is a new tile without Blockers on the board. In both cases, the agent will learn the mechanics behind how Blockers are stripped of layers, and how to remove them completely from the board. Throughout this section, we will use level 65 as an example to make the reward functions more clear. Further details about the levels can be found in Section 7.1.

5.2.1 Damaging Blockers

We propose to reward an agent whenever a Blocker loses a layer, without distinguishing between different types of Blockers. This reward design is denser than the one for freeing tiles because the agent is rewarded every time it takes a layer from the Blocker on a specific tile, not only when it completely removes it. To this end, we propose two reward functions: *Blockers v1 (Bv1)* and *Blockers v2 (Bv2)*. The difference between them lies in how the normalization is performed. Some levels have more Blockers than others, and the same level can have more Blockers of a type than another one. For instance, in level 65 there are nine Cupcake_0, six Cupcake_1, four Cupcake_2, two Cupcake_3, four Cupcake_4, two Cupcake_5, and three Locks (defined in Table 4.1). When a Cupcake_1 is hit, it turns into a Cupcake_0, and this has to be factored in the initial counters when normalizing the reward function. Other types of normalization have been designed but were discarded before the experimental phase after preliminary tests highlighted their poor performances.

Blockers v1

The first function rewards an agent for each Blocker x damaged, normalized by the initial amount x_0 of Blockers of that type. In formulas:

$$r(s_t, a_t) = \sum_{x \in X} \frac{d_{x,t}}{x_0} \tag{5.1}$$

where X is the set of all Blockers and $d_{x,t}$ 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 gets rid of it completely. However, the total reward accumulated in an episode is at most equal to the number of different Blocker types in that level.

Following the aforementioned example, if on level 65 the agent makes a move that removes 1 layer from a Cupcake_5, it will be rewarded with a value of $\frac{1}{2}$, where 2 is the initial number of Cupcake_5 on the board.

Blockers v2

The second function addresses the normalization problem by dividing by the initial number of different Blockers, making the accumulated reward in an episode at most 1. In formulas:

$$r(s_t, a_t) = \sum_{x \in X} \frac{d_{x,t}}{x_0 * U} \tag{5.2}$$

where U is the number of distinct type of Blockers in the level.

Considering the example of level 65, if an agent makes a move that removes 1 layer from a Cupcake_5, it will be rewarded with a value of $\frac{1}{2*7}$, where 7 is the number of distinct type of Blockers in level 65.

5.2.2 Freeing Tiles

In this case, we propose to reward an agent whenever it completely removes Blockers from a tile, making it free. This reward design is more sparse than the previous one, but introduces less bias as the agent is only rewarded for reaching the end goal, that is freeing up the board. Additionally, it should perform better if new Blockers were introduced in the game. Again, two different reward functions are proposed: *Delta Tiles (DT)* and *Progressive Tiles (PT)*.

Delta Tiles

The first function is straightforward: whenever a new tile is freed from Blockers, the agent is rewarded by $\frac{1}{B}$, where B is the board size. In formulas:

$$r(s_t, a_t) = \frac{\delta f}{B} \tag{5.3}$$

where δf is the number of new free tiles. The function is normalized by the total board size B, so the accumulated reward in an episode is at most equal to the number of initial tiles with Blockers, and is less than 1.

Considering level 65, if the agent makes a move that completely removes the last two Blockers on the board, it will be rewarded with $\frac{2}{64}$, where 64 is the total board size.

Progressive Tiles

The second function is inspired from PJ and is more dense than the one defined above. The idea is to reward 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 entire number of free tiles F at the time, whenever a new one is freed. In formulas:

$$r(s_t, a_t) = \begin{cases} \frac{F}{B}, & \delta f > 0 \\ 0, & \delta f = 0 \end{cases} \tag{5.4}$$

Considering the same example of level 65, if the agent makes the same move that removes the last two Blockers on the board, it will be rewarded with

$\frac{64}{64}$, since all the tiles are now free. Clearly, the agent is more prone to free the board completely given the higher reward compared to DT.

5.3 Special Candies

Special candies can be created multiple times when playing a level. Some levels, due to the spacial structure of their board, are better suited to create more Special candies than others. However, it is not possible to predict beforehand how many candies can be created in the best-case scenario, or how many will be used by the player. As a consequence, normalizing the reward to keep it constrained in a fixed range is not possible, so techniques like *reward clipping* [5] cannot be used. Additionally, some Special candies can be created more often than others, so rewarding an agent simply 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.

The limitations presented in Chapter 4, related to event tracking, are a major factor to take into account here since the proposed methods rely on knowing exactly when an agent creates or uses a Special candy. The idea is to keep the same type of function both to learn how to create Special candies and to use them. The first one will give a reward when a candy is created, while the second one when it is used on the board.

5.3.1 Creation

An effective way to deal with features that have different frequencies is to use the frequency itself as a way to balance the weight given to each feature and constrain the reward scale. We test two types of frequency normalization: *Candy Creation v1 (CCv1)* and *Candy Creation v2 (CCv2)*. The first one is adapted from a proven method proposed in the literature, while the second one is designed to solve some of the issues with the first method.

Candy Creation v1

The first type of frequency normalization is adapted from the work of Justensen et al. [7]. In their paper, the authors perform skill learning using intrinsic motivation for the game of Doom. They define a set of events (skills) that should be learned, and reward an agent more when it uses a rarer skill. In CCFS, we define an event for the creation of each type of Special candy, and weight it using its *mean episodic frequency* of creation, which is the average of using that skill over the last k episodes, where k is a hyperparameter. The mean episodic frequency is taken reversed since the purpose is to give more rewards to candies that are created less frequently. In formulas, the function is:

$$r(s_t, a_t) = \sum_{x \in X} c_t^{(x)} \times [\frac{1}{\max(\tau, \mu_t^{(x)})}] \tag{5.5}$$

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 τ is a hyperparameter that represents the initial value of the frequency of each skill, used when no previous data is available (cold start problem), and also identifies the scale of the weights by setting a higher bound.

As an example, if in episode 1 an agent creates three Fishes, and in episode 2 it creates five Fishes, the mean occurrence of Fish creation is four. If in episode 3 the agent makes a move that creates two Fishes at the same time, the reward will be $2 \times \frac{1}{4}$, where 4 is the new mean episodic occurrence, and 2 is the number of Fishes created in that action.

The idea of Equation 5.5 is to reward an agent for exploring new parts of the environment, giving smaller rewards to skills that have already been observed. Even though the method is designed to have an expected cumulative reward for each skill of one, if the mean episodic frequency of a skill is smaller than one, its weight, and thus immediate reward, will be higher than one. In fact, if the environment presents some skills that are seldom used in the last k episodes, their mean episodic frequency will be smaller than one, which leads to rewards with potentially different orders of magnitude, making gradient updates highly unstable.

Candy Creation v2

To address the problem of an unstable reward scale, we propose to normalize the reward function by taking into account the proportion of occurrence of a skill with respect to all the others, such that the weights will always be in the range $[0, 1]$. Our function is defined as follows:

$$r(s_t, a_t) = \sum_{x \in X} c_t^{(x)} \times [1 - \frac{\mu_t^{(x)}}{\sum_{x' \in X} \mu_t^{(x')}}] \tag{5.6}$$

where the denominator of the weight is the sum of all the frequencies of creation of the Special candies. Similarly to the CCv1, our method does not reward agents for winning a level or spreading jam, but it does so for using novel skills. However, unlike the previous one, it does not suffer from the cold start problem, as all the weights are initially set to one. Moreover, it does not require a hyperparameter like τ, as the upper bound for the weights is one. Finally, CCv2 is robust in environments with very unbalanced skills like CCFS, as the frequency is taken in relative terms. Similarly to the previous example, if the mean frequency of fish creation is 4, and the one for a color bomb is 0.4, then the reward for creating one color bomb will be $1 \times \left(1 - \frac{0.4}{0.4+4}\right)$.

5.3.2 Usage

The idea is the same used for Special candy creation, with the difference that the reward is given when a Special candy is involved in a match on the board. When combined with the previous skill, an agent will learn how to create Special candies and how to make proper use of them by understanding their effects. The two types of normalization are the same.

Candy Usage v1

The first type, called *Candy Usage v1 (CUv1)*, is inspired from the work of Justensen et al. [7], and it is not constrained:

$$r(s_t, a_t) = \sum_{x \in X} u_t^{(x)} \times \left(\frac{1}{\max(\tau, \mu_t^{(x)})}\right) \tag{5.7}$$

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

Candy Usage v2

Again, we propose a second type of frequency normalization, called *Candy Usage v2 (CUv2)*, to tackle the issues of the first one. In this case, the function is constrained by taking into account other Special candies when normalizing. In formulas:

$$r(s_t, a_t) = \sum_{x \in X} u_t^{(x)} \times \left(1 - \frac{\mu_t^{(x)}}{\sum_{x' \in X} \mu_t^{(x')}}\right) \tag{5.8}$$

with the same benefits discussed for Special candy creation.

5.4 Network Architecture

We use DQN [17] as the learning algorithm, but the solution can be implemented using any other RL algorithm, even on-policy ones. The deep neural

Layer	Type	Size	Filters	Act. Func.	Strides	Padding
Input	State	$9 \times 9 \times 32$	-	-	-	-
Conv1	Conv.	3×3	35	ELU	1	Same
Conv2	Conv.	3×3	35	ELU	1	Same
Conv3	Conv.	3×3	35	ELU	1	Same
Conv4	Conv.	3×3	35	ELU	1	Same
Conv5	Conv.	3×3	35	ELU	1	Same
FCL1	FCL	999	-	ELU	-	-
FCL2	FCL	144	-	ELU	-	-
Output	-	144	-	-	-	-

Adapted from [2].

Table 5.1: Overview of the parameters for the SSN used.

networks are based on the work of [2] and are used as non-linear function approximators for the target network and prediction network to stabilize the learning algorithm.

The body of the network uses the same design of the supervised approach of [1], optimized via hyperparameter tuning to extract features from the same state encoding described in Chapter 4, while keeping the overall complexity low. It consists of five identical convolutional layers, each having 35 filters with 3×3 kernels, zero padding and a stride of 1. The weights of the kernels are initialized using Xavier initialization [60], and the activation function is the *Exponential Linear Unit (ELU)*:

$$f(x, \alpha) = \begin{cases} x, & x > 1 \\ \alpha(e^x - 1), & x \leq 1 \end{cases} \tag{5.9}$$

The head of the network is made of two *Fully-Connected layer (FCL)s*. Both are initialized using Xavier initialization and use ELU as the activation function. The first one has 999 neurons. The second one is the output layer, so it consists of 144 nodes, each representing the Q-value of one of the 144 actions. The hyperparameters of the head are the same used in [2], who performed tuning for them. The overall network, called *Single Stream network (SSN)*, is summarized in Table 5.1.

5.5 Generalization

Learning skills that can be transferred from one level to the other is not straightforward. Theoretically, an agent must be able to experiment in multiple environments before it can show a sufficient level of performance in a new one. An effective way to do that would be to train it on different levels at the same time. However, given the properties of the MDP used to model CCFS, this cannot be taken for granted and should be proven empirically. In this section, we propose two different training pipelines: *Multi Level Training (MLT)* and *Single Level Training (SLT)*. These can then be used to discover if the current multi-level setup is suited for generalization, or if needs improvements.

5.5.1 Multi Level Training

MLT is inspired by recent results in the literature regarding generalization in RL. Both Cobbe et al. [25] and Justesen et al. [26] argue that training an agent on multiple levels of the same game always improves its generalization abilities because it prevents its parameters from overfitting to a single level. We adapt their technique to CCFS, and leverage on the idea of curriculum learning [61] to build a curriculum of distinct levels for each skill to learn, where an agent can be trained on. This set of levels should comprise as many game features as possible, and display a heterogeneous win rate distribution, including both easy and hard levels. According to the authors of [25], the more variations the agent is subject to, the better its generalization abilities will be.

Unlike [26], we do not have a way to generate infinite levels procedurally, so we focus on a subset of levels in CCFS that contain all the game features available, and only use those to train the agent. The training pipeline features a different level every episode, extracted randomly from the curriculum set to avoid ordering bias. However, the agent never sees the same level twice, as every episode is associated with a different game seed, involving different random behaviors and board configurations. The resulting pipeline highlighted in Figure 5.1, theoretically sacrifices training accuracy to improve generalization, as the agent is unlikely to be in the same state twice, and will be forced to adapt its parameters to all the different configurations it meets.

A possible drawback behind this technique is that the current network might not be expressive enough to capture the unlimited variations caused by pairing a constantly changing seed with a multitude of levels in the same training run. The agent might thus not have enough time to adapt its parameters to each combination and end up not learning anything meaningful as a consequence. To draw a comparison, this pipeline is equivalent to training an SL model using only one epoch, since the model never sees the same training data twice. We will verify whether this is true with empirical evidence in Chapter 7.

5.5.2 Single Level Training

To address the possible drawbacks of MLT, we designed another pipeline to compare the results with. SLT is the method commonly adopted in the literature when using the ALE and simply consists of training each skill on a set of different levels, separately from one another. As in MLT, an agent never plays the same game seed twice, so it will likely never face the same board configuration. However, having the same level removes one layer of complexity in the generalization process, as the board structure and the game features remain the same, so the agent might not be limited by the network complexity in its process of generalization. The pipeline is highlighted in Figure 5.2

Different agents are trained on different levels, and the test levels are the same between them. By testing the agent on common unseen levels, it is possible to shed light on which levels are better suited to teach the agent how to play the game. In fact, some training levels might comprise a larger set of game features, and allow the agent to have higher performances on the test levels.

This technique has two possible drawbacks. The first one, similar to the one of MLT, is that never playing the same seed twice might still lead to a computationally infeasible learning process, where the agent would take too long to learn anything meaningful. This is especially true for levels where the win rate is low, where only playing each seed once means that the agent is not able to win with that seed, as it cannot explore different options by playing it again. The second one is that it cannot be used in production since one level is not enough to represent all the thousand different ones in CCFS. As a consequence, it is only used to study whether the current setup and architecture are expressive enough to capture the differences in the levels, or if something needs to be adapted to account for the stochasticity and heterogeneity of the game.

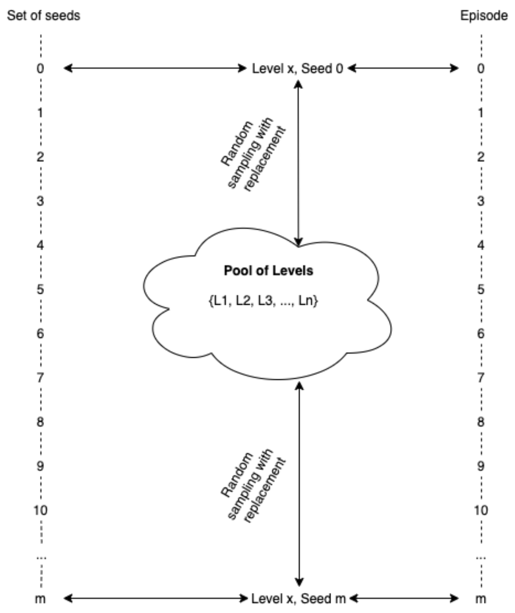


Figure 5.1: Training pipeline with m Episodes, using m distinct seeds and only one level. At each episode, the level is associated to a new seed, forming a unique $\langle Level, Seed\rangle$ pair.

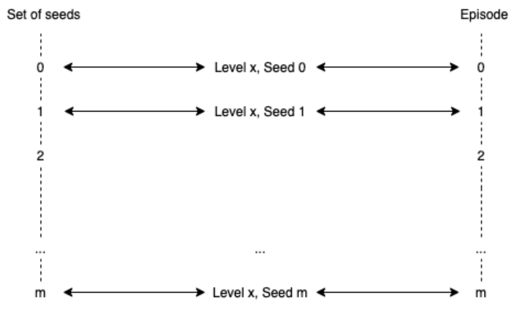


Figure 5.2: Training pipeline with m Episodes, using m distinct seeds and n distinct levels. At each episode, a level is sampled from the pool and associated to a new seed, forming a unique $\langle Level, Seed\rangle$ pair.

Chapter 6

Hybrid Architectures

Once a set of skills is available in the form of different policies, the question becomes how to put them together to improve the general win rate of an agent. Humans have a growing pool of skills to sample from, and they learn which one is the best in a given situation. For an RL agent, there are different ways to replicate this process. In this chapter, two distinct approaches are proposed. The first one, presented in Section 6.1, is an ensemble model that can be thought of as a form of *bagging* [62]. Action selection is performed by taking into account the advice of all policies together. The second one, presented in Section 6.2, is a Hierarchical RL model that learns which policy to select an action from at every timestep.

6.1 Bagging

The first architecture takes inspiration from the literature on bagging, where multiple powerful learners are used together to decrease variance, and adapts the idea to RL. According to how the problem is framed, there are two ways to perform bagging: *Majority Bagging* and *Average Bagging*.

6.1.1 Majority Bagging

In the first one, the output of each policy is considered to be the action it selects, so it is similar to a classification problem with 144 classes, where the choice of each learner falls on the class (action) with the highest Q-value. In this case, bagging can be performed by majority voting: each policy selects an action, and the one that is selected by the highest number of policies is then performed in the environment. In SL, this approach is based on the concept of *wisdom of the crowd*, where the more voters are used, the more accurate the result will be. If the task on which the learners are trained on is the same, the total variance of the ensemble will be reduced. However, in the context of this thesis, only one policy is trained to be goal-oriented, so its vote would be insignificant compared to the sum of all the others, and the final agent would never reach an optimal policy. A final drawback of this method is that it might also require a way to break ties in the voting, depending on the number of sub-policies involved.

6.1.2 Average Bagging

The second approach we propose, called *AB*, is meant to partially address some of the issues of the previous one. In this case, the output of each policy is considered to be the set of 144 Q-values for all the actions, and bagging is performed by selecting the action with the highest average Q-value among all the learners. Since this approach deals with real numbers, the probability of encountering a tie is null, so a tie-breaking mechanism is not required. For this architecture, we propose to adopt three extensions to improve its performances: normalization, weighted averaging, and summation. An overview of the architecture, including all optional extensions, is highlighted in Figure 6.1.

One of the benefits of this architecture is that co-training the policies together can bring an improvement in the final performance, as described in [45]. Each policy could adapt and learn from the others, optimizing its weights to work in conjunction with them. Additionally, to further impose this form of co-operation, we could have one shared convolutional layers between all the policies, and perform a joint optimization of its weights through a loss that is the sum of the losses of each policy. This architecture can be thought of as a network with one common body, and multiple heads, one per policy. Sharing the body can improve generalization, as the parameters of the kernels should not differ between the policies, since the state representation used is the same, and so is the information to be extracted from the board. The difference between each policy should instead lie in the FC layers downstream, which are optimized separately. However, training all the policies together in the same run requires a computational effort the goes beyond the scope of this research, as the burden of optimization is carried out by the same GPU, and from initial experiments, this makes training computationally infeasible. As a consequence, we decided to optimize the architecture by pre-training the policies separately. The ensemble averages the Q-values from networks whose weights have already been optimized, so the burden of computation is limited to the forward

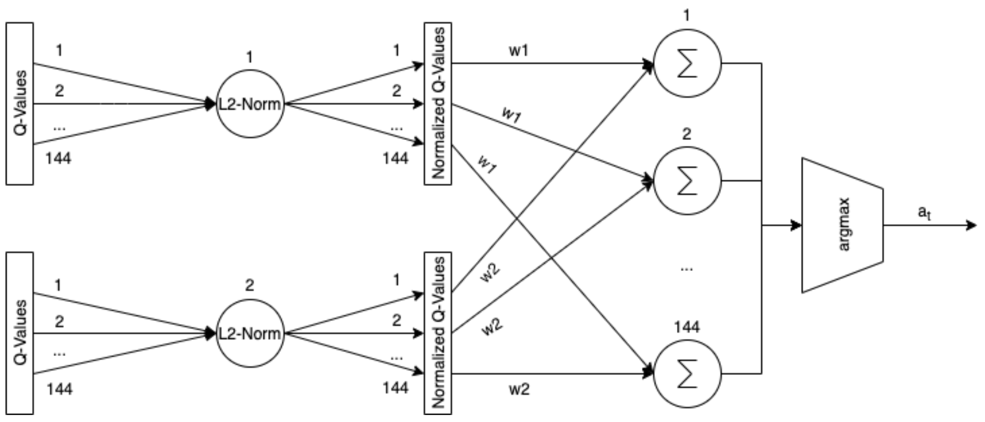


Figure 6.1: Action selection in AB. Two policies are displayed in the example. The first step is normalization of the Q-values for each policy. Then, each value of the first policy is summed to the corresponding one of the second policy, using weights $w1$ and $w2$. Finally, the argmax operation is performed to select the action a_t with the highest sum.

passes on the networks to calculate these values. As a result, the efficiency mentioned in Research Question 2.1. translates to computational efficiency, at the cost of possibly sacrificing win rate performance.

Normalization

When dealing with different reward scales, working with Q-values directly can be problematic, as a policy trained with higher rewards might overshadow the others. Consequently, the effect of averaging 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 a form of normalization to bring all the Q-values in the range $[0, 1]$. For the scope of this thesis, the choice falls on L2-normalization. Given a vector $q = [q_1, q_2, \ldots, q_{144}]$, representing the Q-values of a given policy, the output of L2-normalization is a vector $y = [y_1, y_2, \ldots, y_{144}]$ of the same dimension, where the n-th component y_n is calculated as:

$$y_n = \frac{q_n}{\sqrt{q^T \cdot q}} \tag{6.1}$$

Other types of normalization, such as L1-normalization or L_∞-normalization could be used instead, but testing their effectiveness is left for future work. Here, only L2-normalization is compared against not using any form of normalization, to assess whether it brings any considerable benefits.

Weighted Averaging

In this architecture, the impact of a single vote from a policy is strictly related to the scale of the Q-values of that policy. Even though normalizing everything in the range $[0, 1]$ brings fairness to the averaging mechanism, we are still affected by the problem of under-representing some policies that might be more meaningful for the context at hand. If we are playing a level where removing all Blockers is crucial, the Q-values of the policy that is trained to get rid of them might be lost when averaging between all the other policies, and this would negatively impact performances. The architecture should allow us to express this form of prioritization between policies, and we propose to do it through a weighted average. Each policy is associated with a weight, which is a hyperparameter found through parameter search, and the Q-values of that policy are multiplied by that weight before averaging. A good set of weights can prioritize the right policies considering their real utility in winning the level. Given m policies, where the i-th policy is associated to a set of 144 Q-values $y = [y_{i,1}, y_{i,2}, \ldots, y_{i,144}]$ and a weight w_i, the n-th component t_n of the resulting vector $t = [t_1, t_2, \ldots, t_{144}]$ is computed as follows:

$$t_n = \frac{1}{m} \sum_{i=1}^{m} y_{1,i} \times w_i \tag{6.2}$$

and the action selected is:

$$a_n = \operatorname*{argmax}_n t_n \tag{6.3}$$

In the scope of this thesis, this improvement is not considered in the experimental phase as it requires an extensive search for each weight of the policies. To obtain a useful result, this hyperparameter search should be conducted over as many levels as possible to find a set that works independently of the specific level, but this is unfeasible in the time given. However, it remains a promising extension to be validated with future research.

Summation

The actual implementation of the architecture uses a trick to speed-up training, inspired by the work of [45]. 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 m. All the 144 components of the final vector are scaled by m, which does not change the index of the maximum value selected by the argmax operation. However, in practice, summing instead of averaging is computationally faster, so it is the preferred option. Considering weights, the ensemble becomes a weighted summation, where the n-th component t_n of the resulting vector $t = [t_1, t_2, \ldots, t_{144}]$ is computed as follows:

$$t_n = \sum_{i=1}^{m} y_{1,i} \times w_i \tag{6.4}$$

and the action selected remains the same, that is is:

$$a_n = \operatorname*{argmax}_n t_n \tag{6.5}$$

6.2 High-Speed Hierarchy

The second architecture we propose is called *HSH*, and uses RL to learn which policy to choose from at every timestep t. In this case, the inspiration comes from Hierarchical RL architectures, where different agents interact at different levels of hierarchy.

More than one hierarchical method was considered when designing HSH. Options and Feudal architectures, presented in Chapter 3, have been discarded due to the inability to effectively convey goals from managers to sub-policies using specific states to reach. Whereas in environments like mazes it is easy to specify sub-goals like reaching a particular point, in CCFS there is no clear distinction between which state might be better to win, and specifying it manually would introduce unnecessary human bias. As a consequence, the focus shifted on architectures where the hierarchies do not interact with each other through goals but are independent of one another. In particular, we draw inspiration from [29] and [30], where a *master* chooses one among many *slaves* which one to use in the environment. In our case, the slaves are related to skills and goal-oriented behaviors, while the master should learn which one is better to use in a given situation. An overview of the architecture is provided in Figure 6.2.

As for the AB architecture, this one also theoretically benefits from co-training. In fact, in this case, it would be even more beneficial if we could train the slaves together with the master, such that the former can adapt its parameters to work better in conjunction with the latter, and vice versa. This joint optimization would lead to higher win rates, as described in [29] and

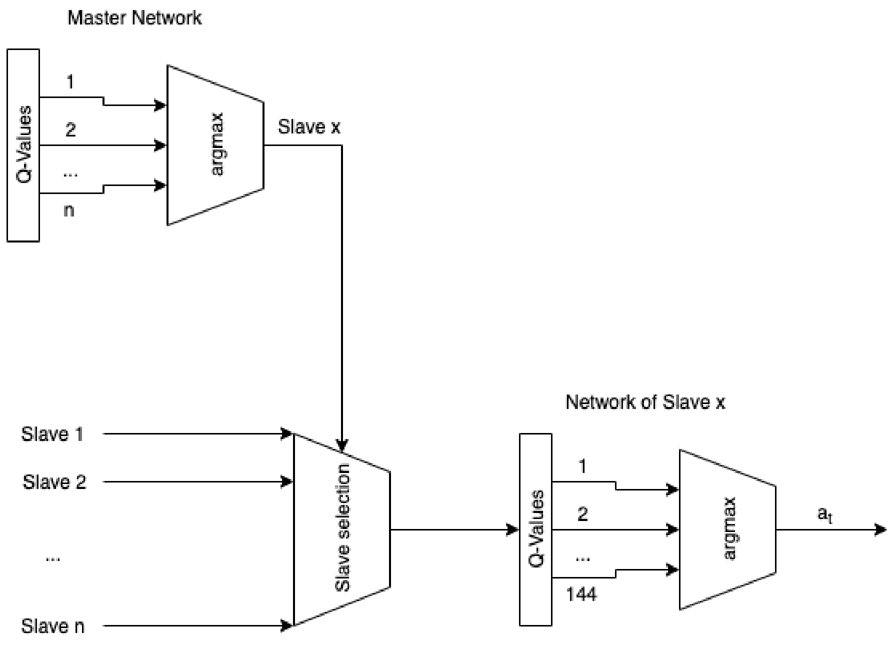


Figure 6.2: Action selection in HSH. First, the master outputs the index of one of its n slaves. Then, the n-th slave performs a forward pass through its network to select the action to take in the environment at time t.

[30]. Sharing convolutional layers could be extended to the master to alleviate the burden of learning weights that should in principle be very similar to understand the same input representation. In practice, training the slaves at the same time as the master requires more computational power than available, for the same reason of AB. In addition to that, this algorithm requires the slaves to converge to a nearly-optimal policy before the master can do it, so this adds even more training time to the total required. As a consequence, like before, we pre-train the slaves separately and then reuse their models as if they were performing inference. Only the master is trained during the learning process, using actions from policies whose training is already converged. Like for AB, the efficiency mentioned in Research Question 2.1. translates to computational efficiency in the implementation, sacrificing win rate performance in favor of training speed.

Slaves

Slaves are agents that have been trained to learn policies related to various objectives. In the architecture we propose here, each slave knows how to perform a skill that can either be goal-oriented or not. One main slave has been trained using PJ as reward function and learns how to spread the jam to win the given level. The other slaves are loaded from those that performed best among the ones presented in Chapter 5, and are focused on removing all Blockers as well as creating and using Special Candies.

Each slave has its own prediction network to approximate the Q-values of a given state s_t and is used to recommend the action to take at time t. Since the slaves are not being trained, the action selection is purely greedy, always choosing the one with the highest Q-value. The architecture of the prediction network is the same presented in Section 5.4, as it is retrieved from the models trained there.

Master

The master is an RL agent itself. It observes the same state space of the slaves but has an action-space with dimension equal to the number of slaves. Its actions correspond to choosing one of the slaves to use in the environment. Unlike [29], we let the master select a slave at every timestep, and only take one action from that particular one. The reason behind this choice is that episodes in CCFS are quite short. For instance, level 61 allows the player to take at most 25 moves, so the master should adapt the strategy move by move. In our case, the result would be similar to the one proposed in the aforementioned paper with $\tau = 1$. The environment provides a reward to the master based on how good the action performed is with respect to winning the current level, which is the final goal of this architecture.

The master has a pair of prediction and target networks, according to the DQN architecture. The network structure is the same as for the slaves, except for the output layer, that has dimension n, equal to the number of different slaves used. The overall network, called *master network*, is summarized in Table 6.1.

Training

The master is using DQN as the learning algorithm, so it can be trained off-policy from experiences collected in the past, even by different agents. We made this choice to rely on the optimal parameters found in [2], and only partially modify the network architecture. However, the learning algorithm is not a constraint for this ensemble to work and can be changed accordingly to the needs.

Layer	**Type**	**Size**	**Filters**	**Act. Func.**	**Strides**	**Padding**
Input	State	$9 \times 9 \times 32$	-	-	-	-
Conv1	Conv.	3×3	35	ELU	1	Same
Conv2	Conv.	3×3	35	ELU	1	Same
Conv3	Conv.	3×3	35	ELU	1	Same
Conv4	Conv.	3×3	35	ELU	1	Same
Conv5	Conv.	3×3	35	ELU	1	Same
FCL1	FCL	999	-	ELU	-	-
FCL2	FCL	n	-	ELU	-	-
Output	-	n	-	-	-	-

Table 6.1: Overview of the parameters for the master network. n is the number of slaves.

The master uses a standard ERB, whose size is to be decided based on how decoupled the experiences should be. This buffer stores samples of experiences in the form of $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$, where a_t is the identifier of one of the slaves that is selected at time t, and r_{t+1} is the reward given by the environment in response to the action taken by that slave.

The loss that is minimized is the standard one for Q-learning, i.e., the difference between the Q-values given by the prediction network and those of the target network. The update formula for the weights of the prediction network, using any form of gradient descent, is the following:

$$\Delta \boldsymbol{w} = \alpha(r_{t+1} + \gamma \max_{a' \in A} \hat{Q}(s_{t+1}, a', \boldsymbol{w}^-) - \hat{Q}(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a, \boldsymbol{w}) \quad (6.6)$$

that is the standard update rule for vanilla DQN. These parameters are copied to the target network every τ timesteps, which is an hyperparameter of the algorithm.

Weight Initialization

The weights of the head of the master network are initialized in the same way as for the slaves, relying on Xavier initialization. For the body, consisting of the convolutional layers, we try to adapt the idea of sharing these parameters with the pre-trained slaves. All the parameters of these body layers are initialized by copying them from one of the slaves, such that the master can extract better features from the input representation straight away. The choice of which slave to use falls on the goal-oriented one, trained with PJ as reward function, as we believe that the master is more akin to the goal of that slave over the others, so it should use its parameters to begin with. This might not be the best approach, but the validation of this hypothesis is out of the scope of this thesis.

Reward Function

The reward function for the master is crucial to ensure a successful learning process for the whole ensemble, as explained in [30]. Ideally, the optimal choice should fall on a reward that represents the purpose of this architecture, which for CCFS means winning a level. As a consequence, we use the sparse function described in 4.2, which only rewards the master for winning a level. Other types of rewards would bias the selection of the master towards one of its slaves, so they are not considered here. However, as a downside, we should take into account that the drawbacks of a sparse reward might apply to this architecture as well, making the learning process slow and possibly affected by the stochasticity in the game. This will be validated with the experiments.

Action Selection

The process of selecting an action to take in the environment involves both the master and the slaves. The master uses an ϵ-greedy strategy, with fixed exploration rate. At every timestep, it either selects a random slave with probability ϵ, or it picks the one according to the greedy selection. In particular, the latter consists of performing a forward pass through the master's prediction network and selecting the slave associated with the highest Q-value from those in the output of the head. From that point, only the chosen slave is used to predict what primitive action to take in the environment. Since the slaves are using a pure greedy approach, a forward pass is performed on the network of the selected slave, and the primitive action with the highest Q-value is chosen and performed in the environment. The action selection process is optimized for speed, as pre-calculating the output of each slave ends up taking more time on a single GPU. However, if more than one GPU was available, the architecture could be scaled by precomputing the action suggested by each slave while also computing the slave chosen by the master, without waiting sequentially.

Chapter 7

Results

This chapter contains all the experiments conducted in the thesis. Firstly, Section 7.1 explains the common experimental setup adopted, including resources, levels, and baselines. Sections 7.2 and 7.3 present the results for the first part of the thesis, i.e., skill learning. Sections 7.4 and 7.5 present the results for the second part, i.e., hybrid architectures. Each section contains a set of hypotheses formulated to answer the research questions that guide this work and the metrics used to evaluate the agents. When reporting the results, we follow the notation presented in Chapter 5 to abbreviate names.

7.1 Experimental Setup

We follow the best practices from the literature regarding how to properly evaluate RL models [25]. Unlike results published using the ALE, we rely on the division in levels of CCFS to train an agent on a set of levels and test it on different ones. Using the training set, we can keep track of metrics such as convergence speed of the algorithm and training loss, which are evaluated when the model is learning and are a good indicator of the stability of the process. We also evaluate the agent on the same levels used during training, but on different seeds, calculating the final training performance on these levels. On the other hand, the test set contains levels that the agent has never seen before, and allows to measure goal-related metrics such as win rate. In particular, the test set is used as a proxy to measure generalization, which is primarily what we are interested in. Since the training and test sets have different statistical properties, in particular related to the win rate, it is important to **avoid** any type of direct comparison between the performance of a model on the train levels and on the test levels. Instead, to measure generalization, we can analyze the difference in performance of the same model on the two sets, and compare it to the difference of another model on the same two sets. If we want to focus on the performance on a single level, we should instead compare the results of two different models on that level.

The selection of levels to include in the training and test set is performed in a preliminary study by taking into account heterogeneity in the game features, board structures, and human win rate, identifying a representative sample of the rest of the levels in CCFS. According to the task we are trying to complete, the levels selected differ. For instance, the training set for policies that learn how to get rid of Blockers is different from the one to create Special candies, since we are trying to cover different game features. However, we try to also have some common levels between them, which can later be used for comparisons. All levels mentioned in this chapter are displayed in the Appendix A, respectively divided in training set in Figure A.1 and test set in Figure A.2. The train and test sets used for each experiment are reported in the following sections.

Model Runs

We start by clarifying the terminology used to avoid misunderstandings. *Testing* and *inference* are used as synonyms. One model is always associated with one RL agent, and *running* it means either training or testing that agent. If we run the exact same model multiple times, we call each run a *trial*. Given the high stochasticity in the game, and the randomness involved when running a model, we want to perform multiple trials to account for the variance in the results. However, we are limited in the number of trials for each model by the time available for the thesis, so a trade-off is required. On a final note, the name of a reward function is used to identify an agent trained with that specific reward function. For instance, with PJ we usually refer to an agent trained using PJ as reward function.

Training

We train every model for a total number of 80 000 episodes, where each episode is one full play on a level, either finishing with a win or a loss. Assuming an average number of 25 maximum moves for the levels used, each training run amounts to two million steps in the environment, the same order of magnitude of other works in the literature. As an example, if we consider level 61, which is the fastest computationally, each training run would take at least five days of continued execution if no other run is performed in parallel. If we instead consider an average between all the levels, the training time would be approximately of seven consecutive days.

For each model, we run five parallel trials. Each trial uses a different value to initialize the seeds of all Python packages relying on randomness, such as Random, Numpy, and TensorFlow. Every operation that involves randomness is regulated by this value, which makes the results completely reproducible. At the end of training, we have five trained versions of the same agent. For instance, when we train the performance of an agent using a reward function like PJ, we launch five training runs with all the parameters fixed, except for the random seed.

Inference

We first test the performance of a trained model on the levels used during training, and then on those belonging to 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. As a consequence, 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 the aggregated values are reported in this chapter. We always measure the inference performance through zero-shot evaluation, meaning that we leave the weights of all models untouched after training, and avoid fine-tuning them on the test levels.

Baselines

We use three different baselines throughout the experiments. The first one is an agent playing randomly, with no learning involved, and it is fundamental to assess the progress of our agents. This baseline is not trained, and we use it by running five inference trials to account for the variance involved when selecting random actions. The second one is a RL agent trained using PJ as reward function and represents the performances of a behavior focused only on the level objective. We train the agent using five trials, following the same process of the other RL agents and then average the results obtained during inference. The third and final baseline is the available human data on those levels, from which we infer the human win rate. This data is not available for each specific seed, but it is aggregated over all the seeds encountered by

Parameter	Value
Buffer size	50 000 (experiences)
Batch size	32 (experiences)
Learning rate, α	$5 \cdot 10^{-4}$
Epsilon, ϵ	0.01
Discount factor, γ	0.5
Prediction network update	2 (steps)
Target network update	100 (steps)

Table 7.1: The common hyperparameters and settings used in all experiments

the players, ignoring boosters and using Tiffy. For legal reasons, data coming from human players is always scaled.

The set of experiments related to skill learning makes use of only the first two baselines since we can track the skill performances exclusively using RL agents, including the one playing randomly. On the other hand, the experiments related to hybrid architectures also use human data.

Hyperparameteres

Some hyperparameters related to the DQN algorithm are shared between all the experiments presented in this chapter. These settings are mostly taken from the original DQN paper [5]. Some key ones, such as the *discount factor* γ, the *target network update* and the *prediction network update*, are instead adopted from [2], who performed hyperparameter search on the CCFS environment for them. In particular, the *prediction network update* refers to how frequently the prediction network performs a training step, and it is measured in environment steps, while the *target network update* refers to how frequently the weights from the prediction network are copied to the target network, and uses the same measurement unit. An overview of all the shared hyperparameters is provided in Table 7.1.

Hardware Resources

The hardware used to run the experiments consists of a Debian server inside King's premises. The server is equipped with a 48 cores Intel® Xeon® CPU, running at 2.60 GHz of clock speed, 792 GB of RAM and one Nvidia Tesla P100 GPU with 16 GB of Memory. With this setup, given that an RL agent occupies at least 780 MB of memory on the GPU, we can run at most 20 agents in parallel. However, since each additional agent slows down all the others, we decided to never launch more than 15, as the benefits from having five additional agents running would be lost with the decrease in computational speed.

During the final phase of experiments, three Virtual Machines (VM) on Google Cloud Platform were available, each running an eight cores Intel® Xeon® CPU and an Nvidia K80 GPU with 12 GB of memory. These VMs can run at most three agents in parallel, otherwise seeing performance drops.

7.2 Blockers

When evaluating policies to remove Blockers, the primary metric used is called *clearing percentage*. It represents the percentage of Blockers removed during an episode with respect to the initial number of available ones, for each type of Blocker on a given level. This measure is averaged over the last 100 episodes using a running mean. Even though these policies should be independent of the goal of the given level, we also keep an eye on another metric, called *win rate*, which is computed by dividing the number of total wins achieved during the inference run by the total number of episodes played, which is 10 000. This value is always scaled for legal reasons, and the same applies to all win rates reported in the results. In the case of two policies that perform similarly as far as clearing percentage, we take the win rate into account to break the tie.

Level Selection

The philosophy used to select levels for policies trained to remove Blockers consists in using the minimum number of levels that cover all the existing types of Blockers for Spread the Jam levels, while including board structures that differ from one another. As mentioned in Chapter 4, the types of Blockers available are Cupcakes, Liquorice Locks, Liquorice Swirls, and Liquorice Links.

The training set is composed of three levels: 65, 82, and 103. The first one has a simple board structure, contains all the types of Cupcakes and Liquorice Locks, and has the lowest human win rate among the three. The second one starts with less free tiles in the beginning, and covers a subset of Cupcakes and Liquorice Swirls, with the latter changing position on the board during the game. The final one has a board structure that is akin to the first one, covers all Liquorice Links and as a subset of Cupcakes, and has a much higher human win rate than the first two.

The test set is composed of three levels as well: 136, 147, and 163. The first one has a peculiar board structure, quite different from all the others. It features a few Cupcakes, many Liquorice Swirls, and some Liquorice Locks, and has a win rate comparable to level 65. The second one has a standard board structure but introduces portals, never seen during training. This characteristic allows us to assess how to agent responds to a feature for which it did not learn network parameters. The level covers Cupcakes, Liquorice locks, and Liquorice Swirls, so it can be used to measure the skills learned for all Blockers while having a lower win rate than the levels of the training set. The final one is almost fully covered with Blockers, including Cupcakes, Liquorice Locks, and Liquorice Links, and has the lowest human win rate among all the levels seen so far.

Hypotheses

We formulate a first set of hypotheses to answer research questions 1.1. and 1.2., which are related to the ability of the reward functions presented in Section 5.2 to learn how to get rid of Blockers. The second set of hypotheses is meant to answer research question 1.3., and are related to the generalization ability of the reward functions with respect to the training pipelines presented in Section 5.5. The hypotheses are:

Hypothesis 1 (H1): *There exists at least one reward function that has a clearing percentage greater than both baselines, for each type of Blocker on every test level.*

Hypothesis 2 (H2): *The reward function with the highest clearing percentage score is either Bv1 or Bv2, given the higher rewards density.*

Hypothesis 3 (H3): *The agents trained with reward functions focused on removing Blockers have a higher win rate than both baselines.*

Hypothesis 4 (H4): *The performance of an agent trained with MLT is higher than the performance of the same agent trained with SLT if both are evaluated on the same test level.*

Experiments

In SLT, each reward function is trained on each level from the training set, separately. For testing, we first test the trained agents on the same levels of the training set, using different seeds. Each agent is only tested on the level where it was trained, not on the other two. Then, we test the agents on the levels from the test set. We first aggregate the results of the five trials of each agent. Then, for each level in the test set, we further aggregate the performances from the same agent trained on different levels. This process is better illustrated in Figure A.3 of the Appendix. We report the win rate and the clearing percentage of all Blockers together, both on the train and test set. The results are summarized in Table 7.2. The raw data for each level is instead reported in tables A.5 and A.6 of the Appendix.

After collecting and analyzing the results of SLT, we make a pruning step before launching MLT experiments, discarding Bv2 and DT. In general, their performances were clearly sub-par on all levels, and preliminary tests on MLT indicate that the same applies when training on multiple levels at the same time too. For this reason, we decide to move on only with PT and Bv1 as candidate functions.

In MLT, we train each agent on all the levels from the training set at the same time, according to the procedure presented in Section 5.5. Firstly, we test the trained agents on each of the levels of the training set separately. This allows us to understand if the agents trained on multiple levels meet a decrease in performance on a specific level of the training set, compared to one trained on that level only. We then test the agents on the levels from the test set. In this case, we do not aggregate the results over different levels, as each agent is trained on all levels together. We use the same metrics used for SLT. The results are summarized in Table 7.3, while the raw data is reported in tables A.1 and A.2 of the Appendix.

7.3 Special Candies

The metrics used to compare candidate functions to create and use Special candies are the *creation probability*, the *usage probability*, and the *match-3 probability*. The first one measures the probability that an action of an agent will create a Special candy of a given type, while the second one measures the probability that the action of that agent involves at least one Special candy of a given type. The third one measures the probability that an action of an agent does not create any Special candies, which is something that we want to minimize. They are calculated by measuring how many Special candies of each type are created and used during an episode, divided by the number of moves taken in that episode, to account for the differences between levels. These measures are averaged over the last 100 episodes through a running

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**
DT	1.32	0.31	69.54	69.31
Bv1	1.95	0.34	75.82	71.35
Bv2	1.31	0.27	73.67	69.64

Table 7.2: Aggregated results in the training and test levels for models trained with SLT. Random policy and PJ baselines are included.

Reward	**Win Rate**		**Clearing (%)**	
	Train	Test	Train	Test
Random	0.21	0.05	39.39	63.47
PJ	1.04	0.38	57.38	69.65
PT	**2.25**	**0.69**	**68.38**	**74.11**
Bv1	1.12	0.43	64.25	71.84

Table 7.3: Aggregated results in the training and test levels for models trained with MLT. Random policy and PJ baselines are included.

mean. Specifically, when evaluating functions to create Special candies, the primary metric considered is the creation probability, while the usage probability is used to break ties when two functions perform similarly. The inverse applies for functions that use Special candies. Finally, the win rate of the agent is also used here to understand the relationship between the considered skills and the ability to win the levels.

Level Selection

For policies trained to create and use Special candies, we focus on selecting levels where the possibility to create them is different. Some levels have board configurations and game features that allow creating Special candies easier, while others are more constrained by the presence of Blockers, which prevents their creation. In fact, creating and using Special candies is much easier in a level without Blockers, as the entire board can be exploited to make the needed matches.

The training set is composed of three levels: 61, 82, and 151. Level 61 is the first level in CCFS that introduces Spread the Jam as objective, so it very simple and does not contain Blockers. On average, humans always win the level, and we expect to be able to generate Special candies more easily than in other levels. Level 82 is shared with Blockers policies, and has a much smaller part of the board without Blockers, so Special candies will be harder to create and use. Level 151 only has four columns initially empty where an agent can create Special candies, while the rest is filled with Blockers. This level also features a portal and has an extremely low human win rate, comparable to 163.

The test set is composed of three levels as well: 62, 147, and 163. The first one does not contain Blockers and is quite similar to level 61. It features multiple portals, and its human win rate is halved with respect to 61. The other two levels are shared with Blockers policies for comparisons and are both quite challenging as far as creating Special candies since most of the initial board is filled with Blockers.

Hypotheses

The set of hypotheses related to generalization is the same formulated for Blockers. In addition to those, we formulate new hypotheses related to the ability of the reward functions presented in Section 5.3 to learn how to create and use Special candies. These are:

Hypothesis 5 (H5): *There exist two rewards function that respectively have a higher usage probability and creation probability than both baselines, for each type of Special candy on every test level.*

Hypothesis 6 (H6): *The reward functions with the highest usage probability and creation probability are respectively CUv2 and CCv2, due to a better reward normalization.*

Hypothesis 7 (H7): *The reward functions trained to optimize the usage probability also learn how to create more Special candies than the baselines, but not* vice versa.

Experiments

The setup for SLT is the same as in the previous section. Each reward function is trained on each level from the training set, separately. We first test the trained agents on the same levels of the training set and then on those from the test set. The results on the test set follow the same idea illustrated in Figure A.3 of the Appendix. We report the win rate, the match-3 probability, the usage percentage, and the creation percentage, aggregating the results for all Special candies together, both on the train and test set. The results are summarized in Table 7.4, while the raw data can be found in Tables A.7 and A.8 of the Appendix.

We decide to perform a similar pruning step after analyzing the results of SLT. The reward functions using the first type of normalization show lower performances overall and are discarded. We move on with CUv2 and CCv2 as candidates for MLT.

The setup for MLT is also the same as in the previous section. Each reward function is trained on all levels from the training set together. We first test the trained agents on the levels of the training set, and then on those from the test set. The metrics reported are the same used for SLT. The results are summarized in Table 7.5. The raw data is reported in Tables A.3 and A.4 of the Appendix.

7.4 Average Bagging

The main metric used to evaluate this method is the win rate since the overall purpose of the architecture is to win the game. To provide a fair comparison against the baselines, we also report the *skill rate* ρ, which is the increase of the win rate w of our agent over the random policy, divided by the increase of another baseline over the random agent. This metric provides a benchmark of our agent with respect to a given baseline, which is here taken to be human data. In formulas:

$$\rho = \frac{w_{agent} - w_{random}}{w_{baseline} - w_{random}} \tag{7.1}$$

Level Selection

In AB there is no training involved, so all the levels can effectively be used for testing. However, we use level 82, which is shared by all the models trained on the various skills, to find the best combination of skills for the hybrid. Then, if the first step gives promising results, we use all the other test levels to assess the

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
CUv1	6.76	3.33	82.15	80.98	9.73	10.43	1.72	1.76
CUv2	6.54	3.44	81.30	79.89	10.40	10.83	1.70	1.90
CCv1	4.90	1.48	80.93	83.94	7.88	7.58	2.94	2.83
CCv2	7.54	**4.03**	**48.24**	**62.89**	**11.93**	**10.91**	**9.06**	**6.71**

Table 7.4: Aggregated results in the training and test levels for models trained with SLT. Random policy and PJ baselines are included.

Reward	Win Rate		Match-3 (%)		Usage (%)		Creation (%)	
	Train	Test	Train	Test	Train	Test	Train	Test
Random	4	1.77	90.33	89.33	6.95	7.36	1.93	2.06
PJ	**7.31**	3.73	83.93	83	8.57	9.11	1.73	1.84
CUv2	6.78	**3.90**	81.33	78.93	**8.73**	**11.05**	1.72	1.78
CCv2	5.07	2.17	**74.33**	**77.20**	8.73	8.67	**5.01**	**4.02**

Table 7.5: Aggregated results on the training and test levels for models trained with MLT. Random policy and PJ baselines are included.

performance of this architecture, and whether or not it generalizes better than the baselines. As a result, there is no training set, and the test set is composed of levels 62, 136, 147, and 163.

Hypotheses

We formulate a set of hypotheses to answer research questions 2.1. and 2.2., related to the ability of AB to mix skills together, independently of generalization. Then, we formulate additional hypotheses to answer research question 3.1., and assess the generalization ability of the method. The hypotheses are:

Hypothesis 8 (H8): *There exists at least one set of skills, combined with AB, which performs better than an agent trained using only extrinsic rewards, both* *on train and test levels.*

Hypothesis 9 (H9): *Using L2-normalization improves the performances of the architecture as opposed to not normalizing the Q-values.*

Hypothesis 10 (H10): *The combination of skills that performs best is the one using all the available skills, including spreading the jam.*

Hypothesis 11 (H11): *Using sub-policies pre-trained with MLT leads to a higher win rate in the test levels than the ones pre-trained with SLT.*

Experiments

As a first step, we only use the agents pre-trained with SLT on level 82 to have access to both skills for Blockers and Special candies. AB does not require a training phase, so all the skills can be combined without further tuning on the levels used for testing. 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 have not been trained and assess the overall generalization capability of the architecture. We average the results over the five training trials, and report the performance of the two best performing combinations, on both train and test levels, in Table 7.6. The full results of the training experiments, where we try every combination on level 82, are reported in Table A.9 of the Appendix.

As a second step, we focus on the models pre-trained with MLT on multiple levels, which theoretically have better generalization capabilities. Again, we use level 82 to find the two best combinations of skills to use for further testing. We then test the selected candidates on the levels where the sub-policies have not been trained and assess generalization. As for SLT, we report the partial results in Table 7.7, while the full training experiments can be found in Table A.10 of the Appendix. Finally, Table 7.8 presents the win rate of the best SLT and MLT combination on each level, compared to the performance of the baselines.

7.5 High Speed Hierarchy

Since the purpose of HSH is to improve the overall win rate of an agent, we use the same metrics adopted for AB to evaluate it: win rate and skill rate against human data. To assess the training performance, we also keep track of

Level	Combination	Win Rate		Skill Rate	
		L2	None	L2	None
82	PJ+PT+CCv2	7.02	7.40	0.73	0.77
	PJ+PT+Bv1+CCv2	7.33	**8.08**	0.76	**0.84**
62	PJ+CCv2	16.44	16.25	0.67	0.66
	PJ+CCv2+CUv2	**17.37**	15.83	**0.73**	0.64
136	PJ+PT+CCv2	3.84	4.12	1.25	1.34
	PJ+PT+Bv1+CCv2	3.51	**4.41**	1.14	**1.44**
147	PJ+PT+CCv2	2.39	2.65	0.34	0.38
	PJ+PT+Bv1+CCv2	2.45	**3.01**	0.35	**0.43**
163	PJ+PT+CCv2	0.1	0.12	0.1	0.11
	PJ+PT+Bv1+CCv2	0.09	**0.14**	0.09	**0.14**

Table 7.6: Win rate and skill rate of the two best performing combinations of AB The sub-policies are trained on level 82 with SLT. Results are grouped by whether L2-normalization was used (**Left**) or not (**Right**).

Level	Combination	Win Rate		Skill Rate	
		L2	None	L2	None
82	PJ+PT+CCv2	2.46	3.27	0.25	0.33
	PJ+PT+Bv1+CCv2	2.84	**3.32**	0.29	**0.34**
62	PJ+CCv2	10.78	11.55	0.33	0.38
	PJ+CCv2+CUv2	**13.34**	12.75	**0.49**	0.45
136	PJ+PT+CCv2	1.31	1.57	0.41	0.5
	PJ+PT+Bv1+CCv2	1.68	**1.79**	0.53	**0.57**
147	PJ+PT+CCv2	0.91	1.15	0.12	0.16
	PJ+PT+Bv1+CCv2	1.19	**1.41**	0.16	**0.2**
163	PJ+PT+CCv2	0.01	0.03	0.01	0.03
	PJ+PT+Bv1+CCv2	0.03	**0.05**	0.024	**0.043**

Table 7.7: Win rate and skill rate of the two best performing combinations of AB The sub-policies are trained with MLT. The results are grouped by whether L2-normalization was used (**Left**) or not (**Right**).

Level	Humans	AB (SLT)	AB (MLT)	PJ	Random
82	**9.60**	8.08	3.32	1.33	0.13
62	**21.92**	17.37	13.24	10.21	5.22
136	3.10	**4.41**	1.79	0.81	0.08
147	**6.90**	3.01	1.41	0.55	0.08
163	**1.03**	0.14	0.05	0.01	0

Table 7.8: Win rate of the best AB SLT and MLT combinations measured on both train and test levels. All baselines are included.

the percentage of times that each slave is used during an episode and call this *selection frequency*. Theoretically, if the master were not able to distinguish between the slaves, each one would have a selection frequency of $1/n$, where n is the number of slaves used in that particular combination.

Level Selection

Similarly to AB, we use level 82 to assess whether the architecture works, and what are the best skill combinations. If the first step gives promising results, we then use the models trained on level 82 on all the other test levels to assess the performance of this architecture, and whether it generalizes better than the baselines. As a result, the training set is composed of level 82, and the test set of levels 62, 136, 147, and 163.

Hypotheses

We only add two additional hypotheses to the ones formulated for the AB architecture, related to the ability of HSH to use the best slave in a given state. H11 and H10 are shared with AB. The new hypotheses are:

Hypothesis 12 (H12): *The master avoids selecting a random agent if it is included in the slaves.*

Hypothesis 13 (H13): *There exists at least one set of skills, combined with HSH, which performs better than an agent trained using only extrinsic rewards, both on train and test levels.*

Experiments

As a first step, we only use the skills pre-trained with SLT on level 82, similarly to the experiments for AB. For each combination, we launch five training trials on level 82. We then use different game seeds on the same level to find the two best combinations of skills to use for further testing in unseen levels.

After the first batch of tests on level 82, we notice that the master struggles to learn how to distinguish the best slave to use at every timestep. The training results on level 82 are clearly far off the ones obtained with AB, so we decide to validate whether this is a problem with the level, or if it is more common. We train HSH on level 61, where we focus on distinguishing between policies to create Special candies, and then on level 65, where we focus on distinguishing between policies to remove Blockers. The full results for all these training runs are presented in Table 7.9. We report the win rate of the three best skill combinations after averaging the performance of the five trials to account for the variance. The skill rate is not reported on training levels.

The same negative results of level 82 are confirmed also for level 61 and level 65, where the master is not able to distinguish between the slaves in a meaningful way. The selection frequency is roughly $\frac{1}{n}$ for all slaves, slightly biased towards PJ, which we use to initialize the weights of the network. We report the selection frequency of the slaves for all training runs in Figure 7.3, where the slaves follow the ordering of Table 7.9. To further test the ability of the master, we run an additional training run on level 61, where we use a slave trained with PJ and one that is playing randomly. The hypothesis is that the master can select the first slave with a frequency close to 1, as the other one has a much lower win rate. However, as reported in 7.4, the selection frequency of the first slave is 52%, meaning that the master is almost selecting randomly. This test discourages us from running this architecture further, and focus on AB exclusively. Our perspective on these results is presented in the next chapter.

Level	**Combination**	**Win Rate**
82	PJ	1.33
	PJ+CCv2	2.69
	PJ+PT	2.06
	PJ+PT+CCv2	**2.8**
61	PJ	20.7
	PJ+CCv2	**22.7**
	PJ+CUv2	20.6
	PJ+CCv2+CUv2	22.15
65	PJ	0.75
	PJ+PT	**1.55**
	PJ+B1	1.09
	PJ+PT+B1	1.57

Table 7.9: Win rate of the three best combinations of HSH tested on the same levels where the slaves are trained with SLT and compared against PJ as a baseline.

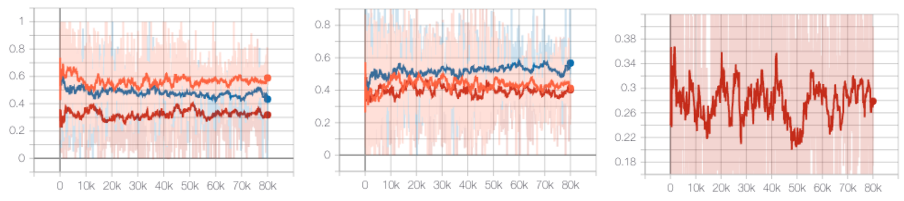


Figure 7.1: Selection frequency on level 61, following the ordering of slaves in Table 7.9, from left to right. (**Orange**) PJ+CCv2, (**Blue**) PJ+PT, (**Red**) PJ+PT+CCv2.

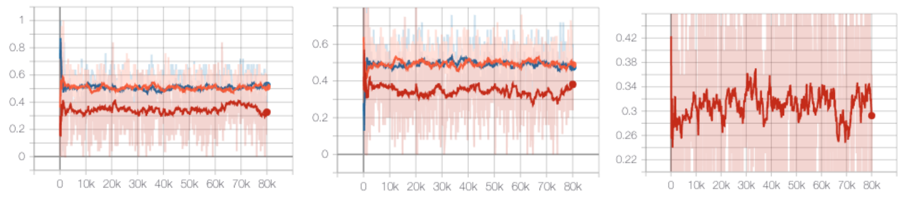


Figure 7.2: Selection frequency on level 65, following the ordering of slaves in Table 7.9, from left to right. (**Orange**) PJ+PT, (**Blue**) PJ+B1, (**Red**) PJ+PT+B1.

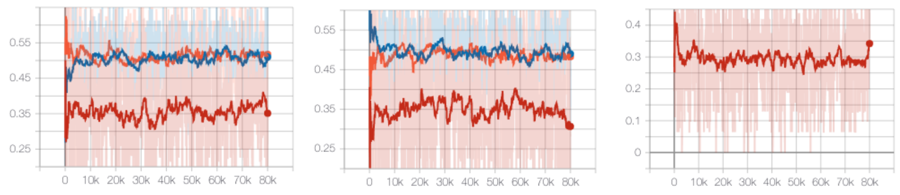


Figure 7.3: Selection frequency on level 82, following the ordering of slaves in Table 7.9, from left to right. (**Orange**) PJ+CCv2, (**Blue**) PJ+PT, (**Red**) PJ+PT+CCv2.

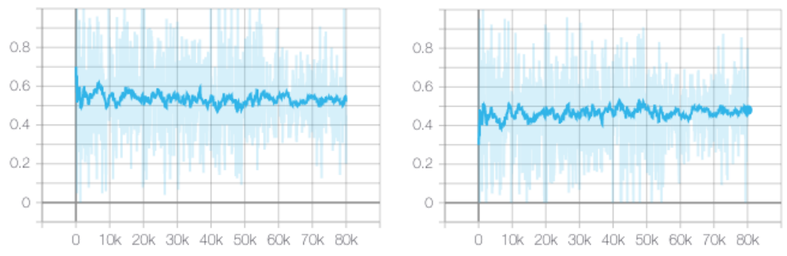


Figure 7.4: Selection frequency on level 61. (**Left**) PJ slave, (**Right**) Random agent slave

Chapter 8

Discussion

The empirical results presented in the previous chapter are promising, and leave plenty of room for discussion. However, they do not come free of limitations. This chapter first provides our perspective on how the results relate to the hypotheses made in Chapter 7. Then, it highlights some key limitations with our methods and proposes different ways to tackle them, leaving room for future work. In particular, Section 8.1 addresses skill learning, going over the correlation found between skills and win rate. Then, Section 8.2 wraps up the experiments by discussing the possible reasons behind the success of AB and the failure of HSH.

8.1 Skill Learning

Given the results of the previous chapter, we can positively answer to research question 1. As highlighted in this section, we found at least one successful candidate function for each skill to be learned. These candidates are more proficient than the baselines we defined and can transfer their ability to the test levels as well. Additionally, they show a higher win rate than the baselines, proving that these skills are essential to winning levels in CCFS. For each candidate identified, we discuss its limitations and how it can be improved with future research. Moreover, we validate the same hypotheses by comparing results with SLT and MLT and call for further experiments to find the best training pipeline.

8.1.1 Blockers

Starting from the results of SLT, we can immediately validate H1. All the proposed functions exhibit a higher clearing percentage than both baselines on the train levels. This means that the reward functions can teach an agent the desired skills. However, only two of those, namely PT and Bv1, show the same behavior over test levels, while the remaining two have a worse clearing percentage than PJ. DT and Bv2 are in fact not able to replicate the same performances over unseen levels, and due to their worse generalization ability are not used for MLT.

H2 is invalidated by the results, as the function with the highest clearing percentage on both train and test levels is PT and not Bv1, despite the lower reward density. We attribute this to a lower degree of bias, as in the former we only declare the goal that we want to achieve, while in the latter we also try to specify how to do it. The result is that Bv1 still exploits the reward by focusing on Blockers that are easier to remove, despite the normalization that we use, while PT is much more balanced overall.

H3 comes from the intuition that PJ fails on harder levels due to its inability to deal with features like Blockers, and this is fully confirmed by the empirical results. An agent trained with a reward function like PT, which is never rewarded for spreading jam or winning a level, still achieves a win rate about twice as high as the one of PJ, both on train and test levels. This is an extremely interesting finding, and partially proves the correlation between skills and game objectives. Moreover, it indicates that this skill is an essential component to use in the hybrid architectures, and this is also confirmed by the experiments of sections 7.4 and 7.5.

MLT

Considering MLT, the discussion above still holds, as the same relative behavior is observed when training on multiple levels. Figure 8.1 presents a comparison between models trained with SLT and with MLT. It is clear that the performance of MLT is worse on the train levels, as the models are less able to fit the training environment since they are trained on multiple levels together and not only on the one where they are then evaluated. However, this translates to superior test performance, especially considering the win rate, meaning that MLT seems to be better suited for achieving generalization than SLT, confirming H4. In the figure, the clearing percentage of MLT PJ on the test levels is lower than SLT PJ, though this is not unexpected, as the model is not trained to deal with Blockers, so we do not expect any form of improved

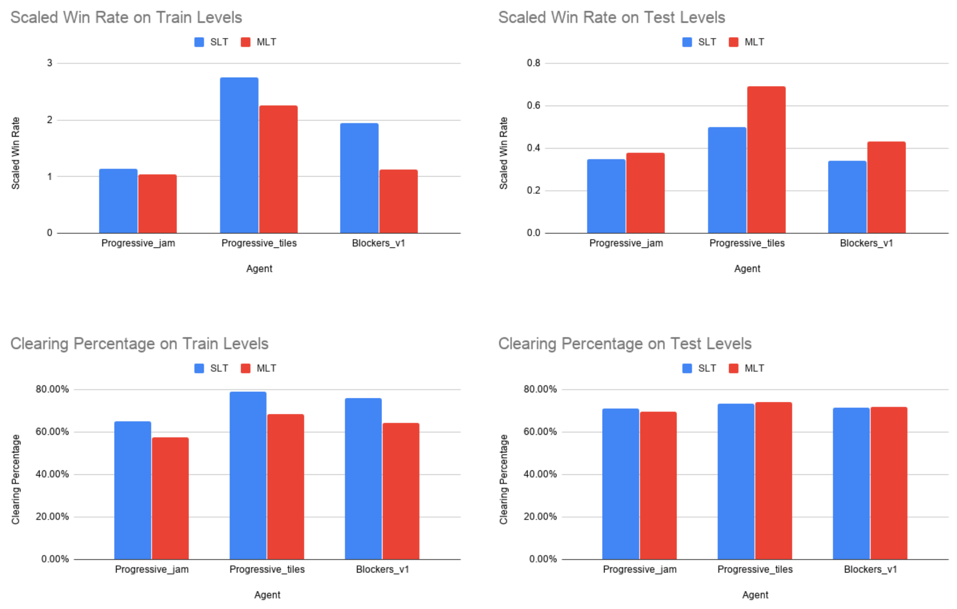


Figure 8.1: Performance comparison of models trained to remove Blockers, divided in SLT and MLT. (**Top row**) Win rate. (**Bottom row**) Clearing percentage.

generalization with MLT. A final take away from these results is that PT comes out as the overall best candidate to train an agent that deals with Blockers, and it is the reason why it is used so much in the hybrid architectures.

Limitations

The main limitation with the above results is related to what we can conclude on MLT. The number of experiments conducted, and the results obtained, do not clearly indicate that MLT is superior to SLT for improving generalization. In fact, the performance drop on train levels is bigger than the gain on test levels, so this is a cause of concern. We need to conduct more experiments to draw a conclusion that is strongly backed by empirical evidence, even extending MLT to more levels at the same time. In case we obtained negative results, we could experiment with other training pipelines. For instance, we could show the same level and same seed to an agent more than once, so it would better adapt its parameter towards that combination. Most importantly, we believe that the current network architecture used is not deep enough to support such a high level of generalization over so many different features and levels. Using a better architecture, such as ResNet [63], should probably be enough to interpolate between all the different variations, at the cost of training speed. We hypothesize that ResNet, paired with MLT, would allow us to obtain performances clearly superior to a SLT pipeline.

8.1.2 Special Candies

Considering SLT, H5 is validated by the results reported in Table 7.4. Both functions designed to use Special candies have a higher usage probability than the baselines, and both functions designed to create Special candies have a higher creation probability than the baselines, considering train and test levels. In particular, the latter also show a lower match-3 percentage than the other models, confirming the fact that they tend to create more Special candies and make less simple match-3s.

H6 is also confirmed by looking at the results. CUv2 has a slightly lower scaled win rate than CUv1 but presents a higher usage probability on both train and test levels, so it the only function focused on using Special candies that we train with MLT. CCv2 qualifies as the overall 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 train levels. This means that, on average, an action every two creates a Special candy. Given these results, we decided not to use CCv1 for MLT, and only focus on the reward normalization proposed by us, which seems to be much more suited for an environment like CCFS, where events have such low and unbalanced frequencies. In Figure 8.2, we highlight this finding by showing the different training behaviours of (I) our method, applied to CCv2 (blue curve), (II) the method proposed in [7], applied to CCv1 (red curve), and (III) PJ, used as a baseline (orange curve).

On the same note, the previous results also invalidate H7, as the opposite seems to be true: functions trained to create Special candies also learn how to use them, and not *vice versa*. This comes as a surprise, and we justify it by observing that the functions that use Special candies focus much more on the Fish, which is the only candy generated by Tiffy in our experiments. As a result, they exploit what is already on the board rather than learning how to create the candies themselves. On the other hand, functions that create Special candies are much more balanced in the usage probability exhibited.

MLT

For what concerns MLT, in Table 7.5 we can observe different results than the one discussed for Blockers. As also shown in Figure 8.3, PJ, and CUv2 present the same relative behavior, with the MLT models having higher test level performances on the metrics of interest for each function, i.e. win rate for both, and usage probability for the second one. However, MLT CCv2 is clearly worse than its SLT counterpart, despite having a higher creation probability than the other MLT models. This deficit in performance likely comes down to the training speed of the model, which is much slower than the others, as depicted in Figure 8.2. On MLT, the models may require more training time, as the parameters have to adjust to different environments. As a consequence, CCv2, which was already the slowest model in SLT, did not have enough training time to get to the same level of performance.

Limitations

A first major limitation with our experiments is related to the way we track the creation and usage of Special candies, as mentioned in Chapter 4. Since we do not receive the events from the game itself, we had to design a best-effort approach to estimate the number of Special candies affected by the actions of an agent, and this estimate is very noisy and error-prone. We believe that we could obtain significantly better results if the game environment provided us with precise counters on all the events that happen after a move, which would also relieve us from the computational burden of calculating them ourselves.

We observe promising results using the normalization technique proposed by us. However, we do not have time to tune a crucial hyperparameter, which is the number of episodes used to estimate the mean episodic frequency of the events. This impacts the balance between the rewards, as if we averaged over the full history we would give more importance to events that were less frequent in the beginning, and give less value to the progress over time. A comparison between a running mean approach and a finite number of episodes should be conducted to understand how this changes the performances of the first type of normalization, and whether it is impacted by the number of levels we train on.

Regarding functions designed to use Special candies, we observe that they tend to exploit the candies that are already present on the board, rather than creating them on their own. This is heavily affected by the bias introduced by the character that we use, Tiffy. We expect to solve this issue when testing without using a specific character so that the functions would not be biased towards already existing candies.

Finally, for what concerns functions designed to create Special candies, we need to look into why MLT CCv2 shows much worse performance than SLT CCv2. We hypothesize that this is due to a training process that has not converged, so the first set of experiments should be focused on training the function for more than 80 000 episodes. We exclude the hypothesis that the problem lies within the normalization used, as CUv2 does not exhibit the same issues when training with MLT, so we expect that longer training times should solve this issue.

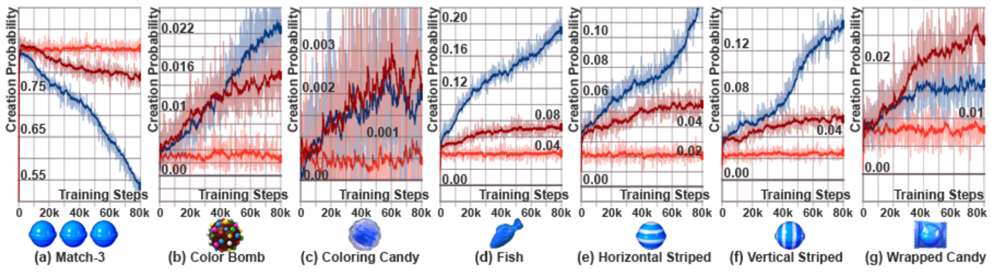


Figure 8.2: Training performance on level 61 of CCv2 (**Blue**), CCv1 (**Red**) and PJ (**Orange**), measured in terms of (**a**) Match-3 percentage and (**b-g**) Creation probability for every type of Special candy.

8.2 Hybrid Architectures

Coming to models meant to combine the skills learned in the first step of the solution, we can also positively answer research questions 2 and 3. Both architectures show a better performance than an agent trained only to reach the objective of a level. However, whereas AB is clearly superior to both baselines, and gets close to human performances, HSH fails to distinguish between its slaves, so it is not used on test levels. In this section, we first discuss why AB is successful, and then provide our hypotheses on the failure of HSH.

8.2.1 Average Bagging

Considering SLT, H8 is fully validated from the experimental results. The majority of AB combinations, including those using fewer sub-policies, always have a higher win rate than both an agent trained with extrinsic rewards and one playing randomly. Moreover, the best models also have a significantly

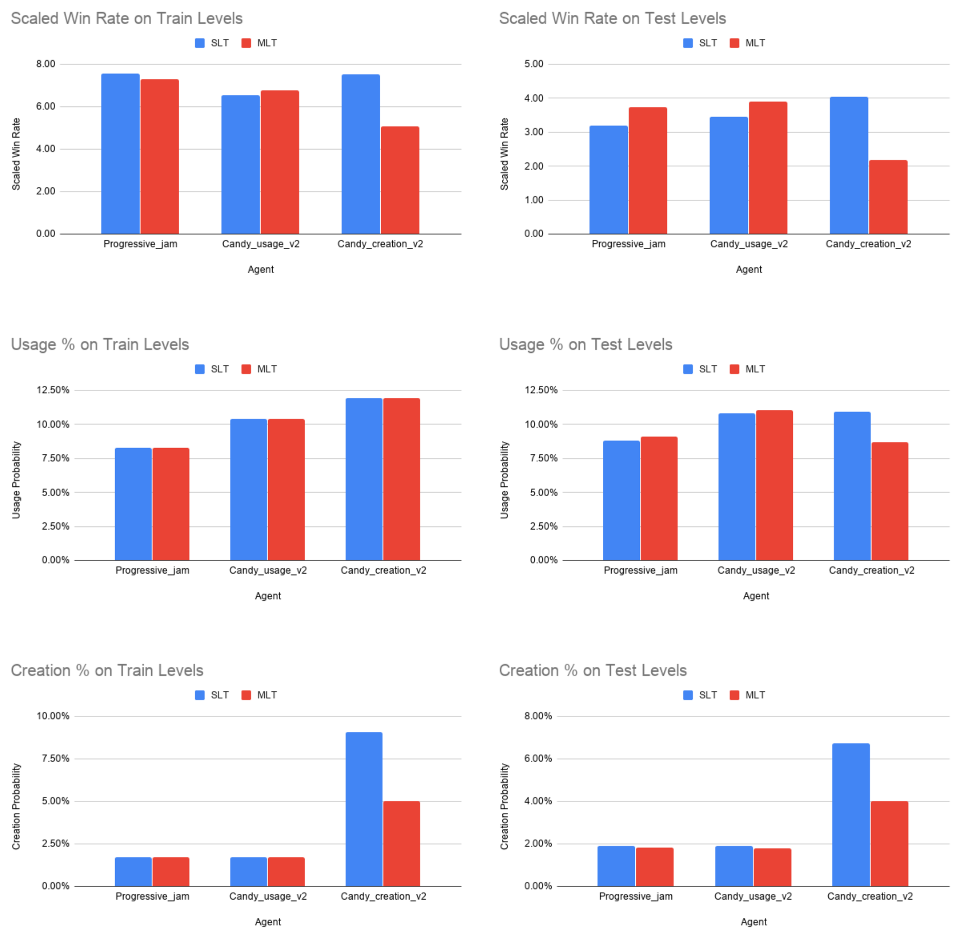


Figure 8.3: Performance comparison of models trained to create and use Special candies, divided in SLT and MLT. (**Top row**) Win rate. (**Middle row**) Usage probability. (**Bottom row**) Creation probability.

higher win rate than the best performing sub-policy used in the combination, meaning that the bagging technique gives the expected improvements.

On the other hand, H9 cannot be confirmed nor invalidated by looking at the experimental results, as the effect of normalization seems to be strictly dependant on the skills used in the combination. For instance, including CUv2 works better with L2-normalization, whereas models that do not use that skill perform better without normalization, except on level 163, where the performance is comparable due to a very low win rate. We guess that CUv2 overshadows the other sub-policies that have a better win rate, due to a higher reward scale, and thus only works with normalization. In the context of this thesis, we claim that L2-normalization does not bring any significant improvements to the combinations that are tested. However, if this architecture is to be extended with more models, which could present win rates on a different scale, using L2-normalization might turn out to be a safer choice.

H10 is also invalidated by experimental results, as the best performing model is only using a subset of all the skills. In particular, PJ, PT, and CCv2 are the three fundamental skills that significantly improve the win rate of the model. Then, Bv1 seems to provide an additional benefit to the hybrid, bringing its performance closer to humans', as reported in Table 8.1. Adding CUv2 to the hybrid instead worsens the performance, proving that using all the sub-policies does not seem to be the best choice. We attribute this last observation to the fact that sub-policies focused on removing Blockers and creating Special candies have also learned to use Special candies as a side effect. However, they in turn have a much higher win rate than CUv2, so adding the latter only makes it worse.

MLT

For what concerns MLT, the above conclusions still hold. There is at least a model that performs better than an extrinsic agent, and the performance of L2-normalization depends on the sub-policies used. Also, the best performing model does not include all the available skills, as CUv2 is once again worsening the performance of the hybrid. However, there is a clear difference between SLT and MLT, as shown in Table 7.8, which invalidates H11. This is not unexpected, as we already mentioned that one of the crucial skills in the architecture is creating Special candies, and MLT CCv2 shows much worse performances than SLT CCv2, due to the training process not converging. As a consequence, even though the best model includes the same skills as in SLT AB, the overall win rate is sometimes even halved, due to a crucial component underperforming.

Nonetheless, we are surprised by the full results on test levels reported in Table A.10 of the Appendix if we compare them to the ones of Table A.9. Even models that do not include CCv2 as sub-policy are performing worse than their SLT counterparts, despite a higher win rate of every single component, as discussed in the previous section. This is better clarified in Figure 8.4, where the performance of models not using CCv2 is compared between SLT and MLT.

Limitations

Given the close performance of SLT and MLT on skill learning, and since we are not able to explain the difference between them in the hybrid architectures, we call for a more thorough study on the best way to train an agent on multiple levels at the same time. First of all, we need to run further tests to prove if each sub-policy trained with MLT is actually better than training it with SLT. If that is not the case, we would need to design a new method for multi-level training, as this is a key step to obtain an agent that can be used across levels in production. However, if we were sure that MLT is better than SLT for each sub-policy, we could narrow down the issue to this architecture, and focus on fixing it instead. We do not see a reason for why AB should work worse when using MLT sub-policies, so we believe that the problem lies within the pipeline itself.

Secondly, more experiments are required to draw a sound conclusion on whether normalizing the Q-values is better. The results presented here point out that this can be the case when using models on different reward scales, which was the initial reason for having a normalization step. However, if the best hybrid architecture is confirmed to include the current skills, normalization would actually worsen the performances and should be avoided. We still believe that normalization is the best choice to ensure that any component can be added without worrying about scale problems.

Thirdly, we did not have time to experiment with a weighted average of the skills. As mentioned in Chapter 6, finding a good set of weights might improve the performances of the hybrid by giving more prominence to some skills. However, performing a hyperparameter search is costly and not scalable when training across levels, as the optimal weight set might change. An interesting extension could be to learn the optimal weights using RL itself, creating an architecture somewhat similar to HSH, where the agent would output the weights for each sub-policy.

Level	AB (SLT)	AB (MLT)	PJ	Random
82	**-15.8**	-65.4	-87.9	-98.8
62	**-20.79**	-39.5	-49.6	-74.2
136	**+42.2**	-42.2	-93.6	-98.7
147	**-56.3**	-79.5	-93.8	-99.8
163	**-86.4**	-95.1	-99.7	-100

Table 8.1: Percentage difference of win rate of the best SLT and MLT AB combinations, compared to human players. Results are measured on both train and test levels. All baselines are included.

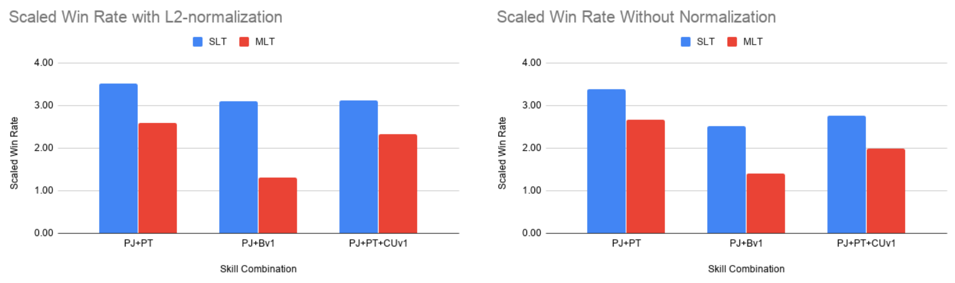


Figure 8.4: Performance comparison of AB combinations not using CCv2 on level 82, divided in SLT and MLT. (**Left**) L2-normalization. (**right**) No normalization.

Finally, we did not have a chance to test all the possible combinations on all the levels, so we cannot conclude that the one presented here is always the best. Further comparisons are needed before concluding which skill combination is better to use.

8.2.2 High-Speed Hierarchy

As seen from the experimental results, H13 is not fully backed by empirical evidence, like in AB. Even though HSH shows better performance than an agent trained with extrinsic rewards, at least on the training levels, it fails to meet its purpose of distinguishing between the best slave to use in the environment.

We test the architecture further, even formulating H12 for this reason, and discover that the master fails to even avoid using the random policy in favor of an agent with a much better win rate. In all training runs, we observe flat curves for the selection frequency of all slaves and thus avoid using the architecture for further testing.

Limitations

We attribute the failure of HSH to three reasons. The first one is stochasticity. When the master selects a slave that is theoretically worse, like a random agent, the action taken might have an overall positive outcome, due to the heavy stochasticity involved in the game from the random effects that follow a move. As a consequence, since the random policy is never used for a whole episode, the master fails to understand that the actions coming from that slave are a worse option.

The second reason is the value of the discount factor γ. Here, we do not have time to perform a hyperparameter search for every architecture, so we keep a fixed value of 0.5, which is likely too low when paired with a sparse reward. To distinguish between the slaves, the master should be able to look all the way to the end of the episode, but this is only possible with a discount factor closer to 1. As a consequence, we call for a hyperparameter search to find the best value of the discount factor for the architecture.

The final reason, and probably the most impactful one, is the choice of the master's reward. As seen for the agents in the work of [2], a sparse reward once again proves to be not suited for an environment like CCFS. Only rewarding the master for winning a level does not seem to provide enough information to understand which slave contributed the most to the final outcome, especially in harder levels like 65 where the reward is very sparse. Ideally, longer training time should improve this, as the master could learn better correlations between each slave and the game outcome. However, we believe that a better improvement worth investigating consists in designing a more dense reward function, still related to the level objective. In this sense, we could try PJ, which provides a more dense feedback signal and should allow the master to better select the slaves, knowing however that it would be biased towards the one that was also trained with PJ itself.

Chapter 9

Conclusions

This closing chapter provides the reader with a summary of the whole thesis. Firstly, Section 9.1 briefly discusses some key issues related to ethics and sustainability that were faced when conducting the research. Then, Section 9.2 provides a summary of the followed methodology, the setup used to run the experiments, and the results obtained. To conclude the chapter, Section 9.3 discusses two extensions worth exploring in future work.

9.1 Ethics and Sustainability

The solution proposed in this research can be seen as an improvement over the supervised approach with respect to privacy. Since RL can learn by unsupervised self-play, it does not need the collection and tracking of user data like the supervised bot did. Moreover, it is not biased by observing human actions and can be perceived as more neutral. Another ethical issue that may arise is related to automation. One could argue that automating this kind of playtesting process might substitute human labor, and in the long term even replace level designers, once it can be used for content generation. We believe that instead of substituting humans, these approaches will work alongside them to improve their workflows. The need for more repetitive manual labor will decrease, so the effort of level designers and testing companies will be spent in more efficient ways. A final ethical issue is related to transparency and explainability. Relying on automated agents to perform content balancing is efficient as long as an explanation for their results can be clearly given to the level designers. If the latter are unable to understand why the agent is performing in such a way on a level, they might fail to balance that level accordingly.

Regarding sustainability, the main issue we identify is power consumption.

As for all ML models, and in particular for RL agents, the training phase requires an impressive amount of resources, from storage to computation and networking. We argue that the path towards generalization might alleviate these problems, as an agent will not have to be retrained multiple times, but will be able to transfer knowledge effectively and require only one training session. In this research, training has been performed using servers on-premise at King and on Google Cloud Platform, both adhering to common sustainability principles. All unnecessary computation has been minimized to save as much energy as possible.

9.2 Summary

This thesis directly builds upon the work of Karnsund [2], with the goal of developing an RL agent that can be used to playtest King's games, focusing on CCFS in particular. We analyzed the results of previous work and observed that an agent fails to generalize over the test levels if it is trained using extrinsic rewards. We thus decided to tackle the challenge of improving generalization by drawing inspiration from human behaviors. Human players learn strategies and patterns as they progress through the game, and are able to reuse them when they are in a suitable board state. These strategies are not necessarily related to the objective of a level, but their execution can indirectly help the player towards achieving it. Following the input of our game designers, we identified a set of valuable skills that a player should execute to improve its win rate. In particular, we focused on removing Blockers from the board, as well as creating and using Special candies. We designed a set of candidate intrinsic reward functions to learn the given skills, and trained an agent for each function to find the best candidate to learn that skill. All agents have been trained in two ways, the first one using multiple levels in the training set, and the second one using just one level, with the goal of proving whether the former improves the generalization ability of the agents over the latter. After obtaining a suitable set of functions, we designed two architectures that allow a new agent to make use of the skills it learned. The first one, inspired by bagging in SL, simply averages the Q-values of the sub-policies associated with each skill and picks the action with the highest average value. The second one, which is a Hierarchical RL model, trains a master agent using RL to pick the best slave in a given board state.

Our experimental setup is designed to provide reproducible and reliable results, taking into account the high stochasticity of the game. We trained each agent for a number of steps comparable to SotA results, using five trials with different random seeds, which allowed us to average five test results for each model. We always compared our results to those of an agent playing randomly and to one trained using extrinsic rewards. For hybrid architectures, we also drew comparisons to human results. We chose a set of levels on which to train our agents, and a different set to test them. The two sets have very different characteristics, so we never compared the results of the same model between them, but rather between different models tested on the same level.

Results

We observed promising results for what concerns learning skills. Considering the clearing percentage of Blockers, our chosen candidate function showed an average improvement of 2% over an agent trained with extrinsic rewards on test levels, and 13% on train levels. Most importantly, even though the candidate function was not trained to win a level and does not know the concept of jam, it still managed 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. The candidate function for Special candies is selected from the ones trained to create them and showed even more promising results than for Blockers. In fact, compared to an agent using extrinsic rewards, the average probability of creating a Special candy is improved from 1.90% to 6.71% on the test levels, and from 1.60% to 9.06% on the training levels. Moreover, the win rate on the test levels is 4.03, compared to 3.20 of the aforementioned baseline. More interestingly, the probability of not creating any Special candy with a move is down to 63%, from 83% of the baseline.

For hybrid architectures, we obtained mixed results. HSH did not show any meaningful learning process, as the master was not able to distinguish between even just a random slave and one with a much higher win rate. The average frequency of selection for each slave is around $\frac{1}{n}$, meaning that the master is basically selecting at random. However, the model still showed higher performances than one trained only using extrinsic rewards, but we decided to not run further tests with it. On the other hand, AB showed encouraging results. Looking at the overall performance of the best model, we believe to have found a promising approach to tackle the issue of generalization. Compared to an agent trained with only extrinsic rewards, the average win rate on the test levels is improved from 2.89 to 6.23. With these results, the difference in performance from humans is down to -30.3%, compared to -84.1% of the aforementioned baseline, even surpassing the human win rate on level 136. The generalization improvement is especially clear on harder levels like 136, 147, and 163, where careful planning is required to win, and an agent focused on spreading the jam exhibited close to random performances. As we said, this is caused by the narrow behavior learned via extrinsic rewards, which we avoided by teaching an agent different skills that can be used based on the situation.

9.3 Future Work

As mentioned in the discussion, there are a number of extensions to this work that are worth exploring in the future. Here, we would like to mention two of them. AB has proven to be an extremely speed efficient and modular architecture, as it effectively does not require a training process. Generalization to different level objectives comes almost for free since it can directly reuse the sub-policies learned during skill learning, and only train one sub-policy with an extrinsic reward for the objective at hand. Training times are heavily reduced, and we can integrate new game features by re-training or fine-tuning the sub-policies. Given these reasons, we propose to extend the architecture by making use of a weighted average and learn the set of weights with RL. This should allow us to find a set of weights that works across levels and improves generalization on the test levels even more, hopefully closing the gap with human performance. For this purpose, we believe that keeping L2-normalization is the right way to move forward.

Secondly, we call for an extended study on the best way to train an agent on multiple levels, since our results do not clearly indicate that MLT is better than SLT. In this context, we also believe that using a deeper and more complex network architecture like ResNet might solve some of the problems highlighted in the previous chapter, and ensure a better multi-level generalization.

Bibliography

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

[2] Alice Karnsund. *DQN Tackling the Game of Candy Crush Friends Saga: A Reinforcement Learning Approach*. 2019.

[3] Richard Bellman. "Dynamic programming and stochastic control processes". In: *Information and control* 1.3 (1958), pp. 228–239.

[4] Ian H Witten. "An adaptive optimal controller for discrete-time Markov environments". In: *Information and control* 34.4 (1977), pp. 286–295.

[5] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[6] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[7] Niels Justesen and Sebastian Risi. "Automated curriculum learning by rewarding temporally rare events". In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2018, pp. 1–8.

[8] Tom M Mitchell et al. *Machine learning*. 1997.

[9] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[10] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[11] Richard S Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1 (1988), pp. 9–44.

[12] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[13] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[14] Jack Kiefer, Jacob Wolfowitz, et al. “Stochastic estimation of the maximum of a regression function”. In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[15] Yoshua Bengio. “Rmsprop and equilibrated adaptive learning rates for nonconvex optimization”. In: *corr abs/1502.04390* (2015).

[16] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[17] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[18] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[20] Matteo Hessel et al. “Rainbow: Combining improvements in deep reinforcement learning”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[21] Dan Horgan et al. “Distributed prioritized experience replay”. In: *arXiv preprint arXiv:1803.00933* (2018).

[22] Adrià Puigdomènech Badia et al. “Agent57: Outperforming the Atari Human Benchmark”. In: *arXiv preprint arXiv:2003.13350* (2020).

[23] Marc G Bellemare et al. “The arcade learning environment: An evaluation platform for general agents”. In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[24] Charles Packer et al. “Assessing generalization in deep reinforcement learning”. In: *arXiv preprint arXiv:1810.12282* (2018).

[25] Karl Cobbe et al. “Quantifying generalization in reinforcement learning”. In: *arXiv preprint arXiv:1812.02341* (2018).

[26] Niels Justesen et al. “Illuminating generalization in deep reinforcement learning through procedural level generation”. In: *arXiv preprint arXiv:1806.10729* (2018).

[27] Kimin Lee et al. “A Simple Randomization Technique for Generalization in Deep Reinforcement Learning”. In: *arXiv preprint arXiv:1910.05396* (2019).

[28] Jesse Farebrother, Marlos C Machado, and Michael Bowling. “Generalization and regularization in DQN”. In: *arXiv preprint arXiv:1810.00123* (2018).

[29] Carlos Florensa, Yan Duan, and Pieter Abbeel. “Stochastic neural networks for hierarchical reinforcement learning”. In: *arXiv preprint arXiv:1704.03012* (2017).

[30] Christopher Simpkins and Charles Isbell. “Composable modular reinforcement learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 4975–4982.

[31] Ofir Nachum et al. “Why Does Hierarchy (Sometimes) Work So Well in Reinforcement Learning?” In: *arXiv preprint arXiv:1909.10618* (2019).

[32] Peter Dayan and Geoffrey E Hinton. “Feudal reinforcement learning”. In: *Advances in neural information processing systems*. 1993, pp. 271–278.

[33] Alexander Sasha Vezhnevets et al. “Feudal networks for hierarchical reinforcement learning”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3540–3549.

[34] Richard S Sutton, Doina Precup, and Satinder Singh. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[35] Tejas D Kulkarni et al. “Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation”. In: *Advances in neural information processing systems*. 2016, pp. 3675–3683.

[36] Pierre-Luc Bacon, Jean Harb, and Doina Precup. “The option-critic architecture”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[37] Jette Randløv and Preben Alstrøm. “Learning to Drive a Bicycle Using Reinforcement Learning and Shaping.” In: *ICML*. Vol. 98. Citeseer. 1998, pp. 463–471.

[38] Andrew Y Ng. “Shaping and policy search in reinforcement learning”. PhD thesis. University of California, Berkeley Berkeley, 2003.

[39] Sam Michael Devlin and Daniel Kudenko. “Dynamic potential-based reward shaping”. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS. 2012, pp. 433–440.

[40] Eric Wiewiora, Garrison W Cottrell, and Charles Elkan. “Principled methods for advising reinforcement learning agents”. In: *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 2003, pp. 792–799.

[41] Anna Harutyunyan et al. “Expressing arbitrary reward functions as potential-based advice”. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[42] Nuttapong Chentanez, Andrew G Barto, and Satinder P Singh. “Intrinsically motivated reinforcement learning”. In: *Advances in neural information processing systems*. 2005, pp. 1281–1288.

[43] Yuri Burda et al. “Large-scale study of curiosity-driven learning”. In: *arXiv preprint arXiv:1808.04355* (2018).

[44] Guillaume Lample and Devendra Singh Chaplot. “Playing FPS games with deep reinforcement learning”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[45] Harm Van Seijen et al. “Hybrid reward architecture for reinforcement learning”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 5392–5402.

[46] Yuri Burda et al. “Exploration by random network distillation”. In: *arXiv preprint arXiv:1810.12894* (2018).

[47] Max Jaderberg et al. “Reinforcement learning with unsupervised auxiliary tasks”. In: *arXiv preprint arXiv:1611.05397* (2016).

[48] Fernando de Mesentier Silva et al. “AI-based playtesting of contemporary board games”. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 2017, pp. 1–10.

[49] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[50] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[51] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. “Automated Video Game Testing Using Synthetic and Human-Like Agents”. In: *IEEE Transactions on Games* (2019).

[52] Igor Borovikov et al. “Towards interactive training of non-player characters in video games”. In: *arXiv preprint arXiv:1906.00535* (2019).

[53] Yunqi Zhao et al. “Winning Isn’t Everything: Enhancing Game Development with Intelligent Agents”. In: *arXiv preprint arXiv:1903.10545* (2019).

[54] Yuchul Shin et al. “Playtesting in Match 3 Game Using Strategic Plays via Reinforcement Learning”. In: *IEEE Access* 8 (2020), pp. 51593–51600.

[55] Erik Ragnar Poromaa. *Crushing candy crush: predicting human success rate in a mobile game using Monte-Carlo tree search*. 2017.

[56] Max Fischer. *Using Reinforcement Learning for Games with Nondeterministic State Transitions*. 2019.

[57] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[58] David Silver et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

[59] Christopher Berner et al. “Dota 2 with Large Scale Deep Reinforcement Learning”. In: *arXiv preprint arXiv:1912.06680* (2019).

[60] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[61] Yoshua Bengio et al. “Curriculum learning”. In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[62] Leo Breiman. “Bagging predictors”. In: *Machine learning* 24.2 (1996), pp. 123–140.

[63] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

Appendix A

Appendix

A.1 Supplement Levels for Experiments

A.2 Supplement Results of Experiments

A.2.1 Skill Learning

A.2.2 Hybrid Architectures

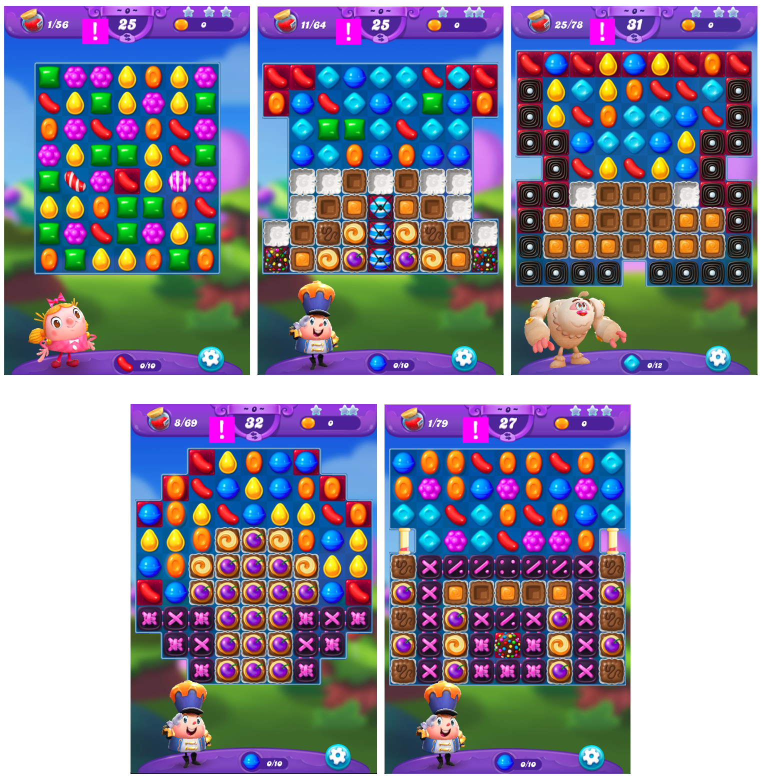


Figure A.1: Example of start board configurations for the levels used in the training sets. (**Top left**) Level 61, (**Top center**) level 65, (**Top right**) level 82, (**Bottom left**) level 103, (**Bottom right**) level 151.

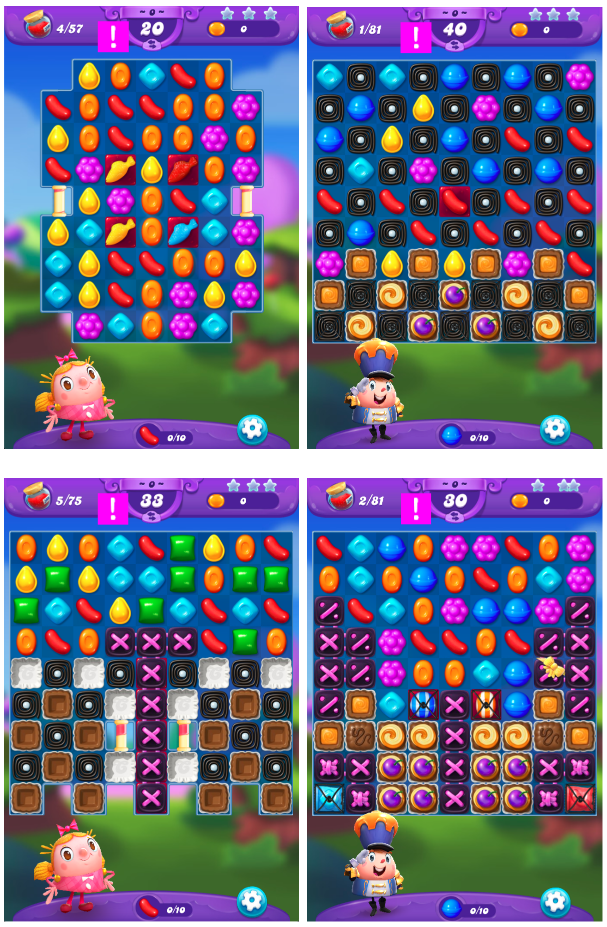


Figure A.2: Example of start board configurations for the levels used in the training sets. (**Top left**) Level 62, (**Top right**) level 136, (**Bottom left**) level 147, (**Bottom right**) level 163.

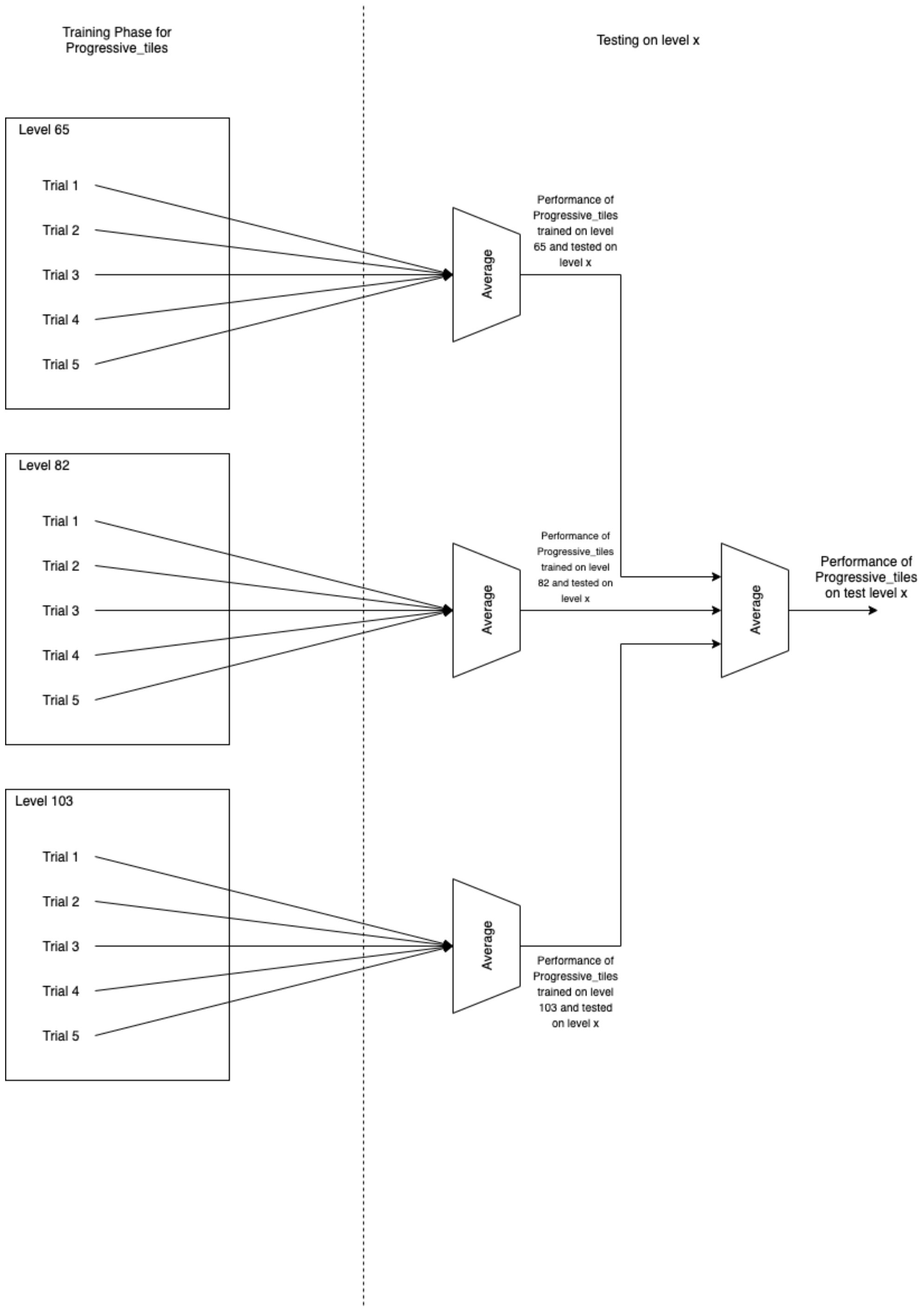

Training Phase for Progressive_tiles
Testing on level x
Level 65
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average
Performance of Progressive_tiles trained on level 65 and tested on level x
Level 82
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average
Performance of Progressive_tiles trained on level 82 and tested on level x
Average
Performance of Progressive_tiles on test level x
Level 103
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average
Performance of Progressive_tiles trained on level 103 and tested on level x

Figure A.3: Example of how results are computed for a PT model trained on the three levels from the training set separately, and tested on a generic level x.

Level	**Reward**	**Win Rate**	**Clearing (%)**			
			Links	Cupcakes	Swirls	Locks
65	Random	0.13	/	40.5	/	45
	PJ	0.78	/	50.5	/	50
	PT	**2.02**	/	**66.5**	/	**71**
	Bv1	1.18	/	60.8	/	69
82	Random	0.12	/	32.7	60	/
	PJ	1.09	/	39.7	70	/
	PT	**2.39**	/	**44.1**	**80**	/
	Bv1	0.6	/	36.8	65	/
103	Random	0.37	56.5	58.8	/	/
	PJ	1.26	67.8	66.3	/	/
	PT	**2.33**	**76.5**	**72.2**	/	/
	Bv1	1.59	83	70.8	/	/

Table A.1: Aggregated results for Blockers models trained with MLT and tested on levels from the training set.

Level	**Reward**	**Win Rate**	**Clearing (%)**			
			Links	Cupcakes	Swirls	Locks
136	Random	0.08	/	53.5	81.4	57
	PJ	0.62	/	66	85	68
	PT	**1.27**	/	**73.8**	**88.6**	**75**
	Bv1	0.83	/	69.7	86.4	71.2
147	Random	0.06	100	61.5	69	/
	PJ	0.53	100	67	76	/
	PT	**0.79**	**100**	**74.7**	**82.6**	/
	Bv1	0.44	100	69.4	76.4	/
163	Random	0	57.7	31.1	/	59
	PJ	0.01	61	37.8	/	66
	PT	**0.02**	**62.9**	42.2	/	**67.2**
	Bv1	0.01	62.6	**43.2**	/	57.8

Table A.2: Aggregated results for Blockers models trained with MLT and tested on levels from the test set.

Level	Reward	Win Rate	Match-3 (%)	Usage (%)	Creation (%)
61	Random	11.97	90	8.3	1.9
	PJ	**20.77**	84.8	10.3	1.6
	CUv2	19.48	81	**12.4**	1.7
	CCv2	14.98	**74**	9.6	**5.1**
82	Random	0.02	90	6.58	1.8
	PJ	**1.12**	83	8.4	1.7
	CUv2	0.83	81	**9.9**	1.7
	CCv2	0.22	**73**	9	**5.2**
151	Random	0	91	5.9	1.9
	PJ	0.03	84	6.9	1.8
	CUv2	**0.04**	82	3.8	1.7
	CCv2	0	**76**	**7.5**	**4.6**

Table A.3: Aggregated results for candies models trained with MLT and tested on levels from the training set.

Level	Reward	Win Rate	Match-3 (%)	Usage (%)	Creation (%)
62	Random	5.2	88	9.2	2.3
	PJ	10.7	81	11.5	1.9
	CUv2	**11.1**	75.6	**14.1**	1.9
	CCv2	6.4	**74.6**	10.2	**4.9**
147	Random	0.07	91	5.9	1.9
	PJ	0.5	85	7	1.7
	CUv2	**0.59**	81.6	**8.9**	1.7
	CCv2	0.09	**81**	7	**3.6**
163	Random	0	89	6.9	2
	PJ	0.02	83	8.7	1.8
	CUv2	**0.02**	79.6	**10.1**	1.8
	CCv2	0	**76**	8.71	**3.5**

Table A.4: Aggregated results for candies models trained with MLT and tested on levels from the test set.

Level	**Reward**	**Win Rate**	**Clearing (%)**			
			Links	Cupcakes	Swirls	Locks
65	Random	0.13	/	42.2	/	45
	PJ	0.75	/	51	/	51
	PT	**2.83**	/	**74.5**	/	**79**
	DT	1.33	/	62.3	/	68
	Bv1	1.18	/	64.8	/	70
	Bv2	1.29	/	64	/	71
82	Random	0.12	/	65.3	60	/
	PJ	1.33	/	80	71	/
	PT	**2.88**	/	**89.7**	**81**	/
	DT	1.23	/	80.7	70	/
	Bv1	2.37	/	86.7	77	/
	Bv2	1.47	/	86.3	78	/
103	Random	0.37	56.5	59.8	/	/
	PJ	1.34	70.5	67	/	/
	PT	**2.55**	75	**74.3**	/	/
	DT	1.4	70.2	66	/	/
	Bv1	2.31	**85.2**	71.2	/	/
	Bv2	1.18	76.5	66.2	/	/

Table A.5: Aggregated results for Blockers models trained with SLT and tested on levels from the training set.

Level	**Reward**	**Win Rate**	**Clearing (%)**			
			Links	Cupcakes	Swirls	Locks
136	Random	0.08	/	54.2	80.3	59.3
	PJ	0.61	/	69.6	87	69.3
	PT	**0.85**	/	**71.3**	**87.7**	**72.3**
	DT	0.49	/	63.7	85.3	66.7
	Bv1	0.63	/	70.1	87	72
	Bv2	0.47	/	63.3	87	69.3
147	Random	0.08	100	60.7	69.3	/
	PJ	0.44	100	67.8	77	/
	PT	**0.62**	**100**	**73.8**	**81.7**	/
	DT	0.42	100	69.3	78.3	/
	Bv1	0.38	100	68.2	76.7	/
	Bv2	0.34	100	67.8	76.7	/
163	Random	0	56.7	32	/	60
	PJ	0.01	60.7	39.7	/	**69.3**
	PT	**0.02**	**61.9**	**43.9**	/	67
	DT	0.01	60.1	38.3	/	62
	Bv1	0.01	61.6	40.9	/	65.7
	Bv2	0.01	59.9	39.1	/	63.7

Table A.6: Aggregated results for Blockers models trained with SLT and tested on levels from the test set.

Level	Reward	Win Rate	Match-3 (%)	Usage (%)	Creation (%)
61	Random	11.97	90	8.3	1.9
	PJ	**21.33**	86	9.6	1.6
	CUv2	18.64	82.5	**12.4**	1.6
	CUv1	19.68	81.6	12.3	1.5
	CCv2	20.41	**52.5**	11.9	**8.7**
	CCv1	14.54	77	9.6	4
82	Random	0.13	89.5	6.8	1.9
	PJ	1.33	82.9	8.7	1.7
	CUv2	0.96	81.4	9.5	1.6
	CUv1	0.56	82.8	8.4	1.9
	CCv2	**2.13**	**46.2**	**12.1**	**9.1**
	CCv1	0.17	82.8	7.4	3
151	Random	0	91	5.9	1.9
	PJ	0.02	86	6.6	1.8
	CUv2	0.04	80	9.3	1.9
	CUv1	0.03	82	8.5	1.7
	CCv2	**0.07**	**46**	**11.8**	**9.2**
	CCv1	0	83	6.6	2.9

Table A.7: Aggregated results for candies models trained with SLT and tested on levels from the training set.

Level	**Reward**	**Win Rate**	**Match-3 (%)**	**Usage (%)**	**Creation (%)**
62	Random	5.22	88.7	9	2.2
	PJ	9.24	80	11.1	2.2
	CUv2	9.84	76.7	13.6	1.9
	CUv1	9.51	78.7	12.7	2
	CCv2	**11.7**	**57.3**	**12.7**	**7.6**
	CCv1	4.4	81.9	9.3	3.4
147	Random	0.08	90	6	1.9
	PJ	0.36	85.4	6.9	1.8
	CUv2	**0.48**	82.3	8.8	1.8
	CUv1	0.46	82.9	8.7	1.6
	CCv2	0.38	**69.7**	**8.9**	**5.8**
	CCv1	0.06	86	6.1	2.6
163	Random	0	89.2	6.8	2.1
	PJ	0.01	83.7	8.4	1.8
	CUv2	0.02	80.7	10.1	1.9
	CUv1	0.01	81.3	9.84	1.7
	CCv2	**0.02**	**61.7**	**11**	**6.7**
	CCv1	0	83.7	7.3	2.5

Table A.8: Aggregated results for candies models trained with SLT and tested on levels from the test set.

Combination	**Win Rate**		**Skill Rate**	
	L2	None	L2	None
PJ+CCv2	5.03	5.37	0.52	0.55
PJ+PT	3.51	3.39	0.36	0.34
PT+CCv2	6.62	6.61	0.69	0.68
PJ+Bv1	3.10	2.52	0.31	0.25
PJ+PT+CCv2	7.02	7.40	0.73	0.77
PJ+B+CCv2	6.21	6.89	0.64	0.71
PJ+PT+CUv2	3.11	2.76	0.32	0.28
PJ+PT+Bv1+CCv2	7.33	**8.08**	0.76	**0.84**
PJ+PT+CCv2+CUv2	6.18	5.66	0.64	0.58
PJ+PT+Bv1+CCv2+CUv2	6.61	5.96	0.68	0.62

Table A.9: Win rate and skill rate of the ten best combinations of AB, measured on level 82. The sub-policies are trained on level 82 with SLT. Results are grouped by whether L2-normalization was used (**Left**) or not (**Right**).

Combination	**Win Rate**		**Skill Rate**	
	L2	None	L2	None
PJ+CCv2	1	1.4	0.09	0.13
PJ+PT	2.6	2.67	0.26	0.27
PT+CCv2	1.63	2.28	0.16	0.23
PJ+Bv1	1.3	1.4	0.12	0.13
PJ+PT+CCv2	2.46	3.27	0.25	0.33
PJ+Bv1+CCv2	1.39	1.78	0.13	0.18
PJ+PT+CUv2	2.32	1.98	0.23	0.2
PJ+PT+Bv1+CCv2	2.84	**3.32**	0.29	**0.34**
PJ+PT+CCv2+CUv2	2.93	2.86	0.3	0.29
PJ+PT+Bv1+CCv2+CUv2	2.85	2.84	0.29	0.29

Table A.10: Win rate of the ten best combinations of AB, measured on level 82. The sub-policies are trained with MLT. Results are grouped by whether L2-normalization was used (**Left**) or not (**Right**).