UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RONALDO E SILVA VIEIRA

DRAFTING IN COLLECTIBLE CARD GAMES VIA REINFORCEMENT LEARNING

Belo Horizonte
2020

RONALDO E SILVA VIEIRA

CRIANDO ESTRATÉGIAS DE *DRAFT* EM JOGOS DE CARTAS COLECIONÁVEIS VIA APRENDIZADO POR REFORÇO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Luiz Chaimowicz
Coorientador: Anderson Rocha Tavares

Belo Horizonte, MG
Outubro de 2020

RONALDO E SILVA VIEIRA

DRAFTING IN COLLECTIBLE CARD GAMES VIA REINFORCEMENT LEARNING

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Luiz Chaimowicz
Co-Advisor: Anderson Rocha Tavares

Belo Horizonte, MG
October 2020

Vieira, Ronaldo e Silva.

V851d Drafting in collectible card games via reinforcement learning [manuscrito] / Ronaldo e Silva Vieira. – 2020.
xiv, 66 f. il.

Orientador: Luiz Chaimowicz.
Coorientador: Anderson Rocha Tavares.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.49-56
.
1. Computação – Teses. 2. Jogos digitais – Teses. 3. Jogos de cartas colecionáveis – Teses. 4. Aprendizado por reforço – Teses. I. Chaimowicz, Luiz. II. Tavares, Anderson Rocha. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*82.1(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

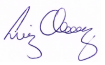
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Drafting in Collectible Card Games via Reinforcement Learning

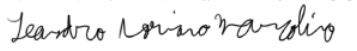
RONALDO E SILVA VIEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:



PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ANDERSON ROCHA TAVARES - Coorientador
Instituto de Informática - UFRGS



PROF. LEANDRO SORIANO MARCOLINO
Lancaster University

PROF. ADRIANO ALONSO VELOSO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 13 de Outubro de 2020.

Aos meus pais, por tudo

Acknowledgments

As most of my acknowledgments are to people who speak Portuguese, the remainder of this section is written in Portuguese.

Apesar de ter meu nome estampado na capa, este trabalho jamais existiria senão pelo suporte, ora técnico, ora emocional, ora financeiro, de um número grande de pessoas. Infelizmente seria necessária uma dissertação à parte se eu pretendesse listar todas elas e suas contribuições, portanto, me limito a uma pequena parte.

Sou grato:

- À minha família, por ter sempre me apoiado incondicionalmente em todas as minhas maluquices acadêmicas.
- Àos amigos que fiz em Belo Horizonte, que antes significava apenas pão de queijo, doce de leite e *uai*, e agora possui um valor sentimental imensurável. Obrigado, amigos do *Patreecios*, *Wisemigos*, dos laboratórios *J*, *VeRLab* e *LaPO*, do *Colab*, do *Futsense*, do campeonato do DCC, dos *Cabráis*, das *Cácias* e dos *Carnavais*, por existirem.
- Ao meu orientador Luiz Chaimowicz e meu coorientador Anderson Tavares, sempre sensatos, por serem tão bons no que fazem. Se hoje eu me considero um pesquisador, a culpa é deles.
- À secretaria do PPGCC, por seu trabalho tão primordial quanto, às vezes, invisível para nós.
- Ao meu cachorro, Kiko, por respirar.
- Aos criadores de música *lo-fi hip hop*.
- Por último mas não menos importante, aos diversos *Ronaldos* do passado, cujo esforço permitiu que o Ronaldo de hoje estivesse onde está. Se algum Ronaldo do passado de alguma forma estiver lendo isso, saiba que você vai conseguir. `:)`

Abstract

Collectible card games (CCGs), such as *Magic: the Gathering* and *Hearthstone*, are played by tens of millions of players worldwide, and their vast state and action spaces, intricate rules and diverse cards make them challenging for humans and artificial intelligence (AI) agents alike. In them, players build a deck using cards that represent creatures, items or spells from a fantasy world and use it to battle other players. Therefore, to win, players must be proficient in two interdependent tasks: deck building and battling. The advent of strong and fast AI players would enable, for instance, thorough playtesting of new cards before they are made available to the public, which is a long-standing problem in the CCG industry.

In this thesis, we present deep reinforcement learning approaches for deck-building in the *arena mode* – an understudied game mode present in most commercial collectible card games. In arena, players build decks immediately before battling by drafting one card at a time from randomly presented candidates. We formulate the problem in a game-agnostic manner and investigate three approaches that differ on how to consider the cards drafted so far in the next choices, using different game state representations and types of neural networks.

We perform experiments on Legends of Code and Magic, a collectible card game designed for AI research. Considering the win rate of the decks when used by fixed battling AIs, the results show that our trained draft agents outperform the best draft agents of the game, and do so by building very different decks. Moreover, a participant of the *Strategy Card Game AI* competition improves from tenth to fourth place when using our best draft agent to build decks. We conclude with a discussion on the results, contributions and limitations of this work as well as directions for future research.

Keywords: Collectible card games, Deck building, Reinforcement learning

Resumo

Jogos de cartas colecionáveis (JCC), como *Magic: the Gathering* e *Hearthstone*, possuem atualmente dezenas de milhões de jogadores pelo mundo. Seus vastos espaços de estados, junto de suas complexas regras e grande quantidade de cartas diferentes fazem com que jogá-los seja uma tarefa desafiadora tanto para humanos quanto para agentes de inteligência artificial (IA). Neles, os jogadores constroem um baralho usando cartas que representam criaturas, itens ou mágicas de algum universo fictício e o usam para batalhar contra outros jogadores. Para vencer, portanto, um jogador precisa ser proficiente em duas tarefas interdependentes: contruir baralhos e batalhar. O advento de IAs que joguem JCCs de forma proficiente e rápida possibilitaria, por exemplo, o *playtest* extensivo de novos conjuntos de cartas antes destes serem disponibilizados para o público, o que é, há muito tempo, um problema em aberto na indústria de JCCs.

Nesta dissertação, propomos abordagens de aprendizado por reforço profundo para a tarefa de construir baralhos no *modo arena* – um modo de jogo presente na maioria dos jogos de cartas colecionáveis comerciais. No arena, os jogadores constroem seus baralhos imediatamente antes de batalhar, escolhendo uma carta de cada vez dentre cartas aleatórias apresentadas (processo chamado de *drafting*). Nós formulamos o problema de forma genérica, aplicável a vários JCCs, e investigamos três abordagens que diferem em como considerar as cartas já escolhidas nas próximas escolhas, usando diferentes representações de estados e tipos de redes neurais.

Realizamos experimentos no *Legends of Code and Magic*, um JCC desenvolvido especificamente para pesquisa em IA. Usando como métrica de desempenho a taxa de vitória dos baralhos ao serem usados por IAs em batalhas, os resultados mostram que nossos agentes de *drafting* alcançaram desempenho melhor que as melhores IAs disponíveis para o jogo, e o fizeram construindo baralhos muito diferentes dos construídos por elas. Além disso, uma IA participante da competição *Strategy Card Game AI competition*, realizada na conferência IEEE CoG 2019, subiu do décimo para o quarto lugar na classificação ao usar nosso melhor agente para construir seus baralhos. Concluímos com uma discussão sobre os resultados, contribuições, limitações e possíveis

trabalhos futuros.

Palavras-chave: Jogos de cartas colecionáveis, Construção de baralhos, Aprendizado por reforço.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Games have been increasingly used as benchmarks for artificial intelligence (AI). In the last decade, AI algorithms were found to achieve superhuman performance on traditional games such as Go [Silver et al., 2017] and Poker [Brown and Sandholm, 2019], as well as on complex strategic video games such as Dota 2 [Berner et al., 2019] and StarCraft II [Vinyals et al., 2019]. Many of these solutions led to significant breakthroughs in the field, such as the advent of deep reinforcement learning methods [Mnih et al., 2013].

In turn, the development of strong AI players can also foster new insights on a game and help discover unforeseen strategies.[1] If the AI is fast enough to play thousands or millions of matches in reasonable time, it can aid in game balancing – one of the hardest tasks game designers face when adding new content or mechanics to a game. If the AI's reasoning power is adaptable, it can provide challenging opponents for human players of all levels. However, there are many games that current AI methods are not able to master yet. This is the case of collectible card games (CCGs).

Collectible card games, such as *Magic: the Gathering* and *Hearthstone*, are adversarial turn-taking two-player games that are challenging for humans and AI agents alike [Hoover et al., 2020]. Their digital and in paper versions form a powerful industry, currently hoarding tens of millions of players worldwide. In CCGs, players build a deck from a collection of cards, often representing creatures, items or spells from a fantasy world, and use it to battle each other, playing cards on a board and using them to interact with the opponent's cards. The game revolves around a resource often called mana, which is required to play cards – the stronger the card, the higher its mana cost.

[1] When playing against Lee Sedol, one of the world's best Go players, DeepMind's AI AlphaGo made a highly unexpected move that changed the course of the second contest of their best-of-five match in its favor, leaving many spectators astonished [Metz, 2016].

The game ends when a player successfully reduces their opponent's health points to zero or any of the other (usually many) win conditions is achieved.

Collectible card games require reasoning in very large discrete state spaces with a large number of possible actions. Their set of rules is usually vast, intricate, and can be modified or augmented by some cards during the game. This makes them much more complex than traditional card games such as Bridge or Poker. Furthermore, winning a match requires not only good battling skills but a well constructed deck.

Deck building is done differently depending on the game mode. In the most common modes, often called constructed modes, a fixed, usually large card pool is available for the players to build decks in an offline manner. In draft-based modes, on the other hand, players are required to incrementally build a deck before each match or set of matches, selecting one card at a time from a few randomly presented candidates.

Deck building in the arena mode, a draft-based mode most notably found in *The Elder Scrolls: Legends* and *Hearthstone*, is still an understudied subject: as of the time of writing, only a single published work specifically addresses the problem [Kowalski and Miernik, 2020]. In this thesis, we investigate the use of deep reinforcement learning methods to the task of deck building in the arena draft mode of collectible card games.

1.1 Research objectives

Deep reinforcement learning is responsible for many recent breakthroughs in game AI research. In CCGs, however, its use has not been extensively explored. Our overall research objective is to investigate whether deep reinforcement learning methods can achieve competitive performance in comparison to current state-of-the-art approaches in the task of deck-building in the arena mode. Given a specific battle agent, the ultimate goal is to find a draft strategy that maximizes the win rate of that battler over all possible drafts and battles against all possible opponents. Since evaluating a draft strategy in such scenario is an unattainable task, we resort to a reasonable set of drafts, battles and opponents.

Our specific goals are (i) to encourage further research on the topic, by developing all source-code in a reusable fashion and making it available under an open-source license, as well as enumerating possible extensions and applications of this work; (ii) to advance the state-of-the-art of deck building in the arena mode, by presenting draft strategies that achieve greater win rates than the best known ones; and (iii) to study the differences between draft strategies, by pointing them experimentally and analyzing them.

We propose three approaches that explore different formulations of the task. A state-of-the-art deep reinforcement learning is used to train each of them on various neural network architectures by self-play, using fixed battle agents. The performance of the resulting draft strategies is compared to a set of baseline and state-of-the-art draft strategies. The training and evaluations are made on *Legends of Code and Magic* (LOCM) [Kowalski and Miernik, 2018], a collectible card game designed especially for AI research. To speed-up experiments and facilitate further research, we reimplemented the game engine as OpenAI Gym [Brockman et al., 2016] environments.

1.2 Contributions

To apply reinforcement learning, we formulate the arena drafting problem as a Markov Decision Process (MDP). Then we propose three approaches that differ in state representation and type of neural network used. We train each of them partnered with different battle agents, finding ideal hyperparameters and network architecture via hyperparameter optimization.

We compare them to the most current approaches in literature. These comparisons are made by observing the win rate, mana curves (see Section 2.1) and choice similarity of the approaches in a round-robin tournament while using the same battler. As shown by the experiments, the resulting draft strategies significantly outperformed those of the best game-playing agents by building very different decks. We also show that our best draft strategy could have improved the ranking of a participant bot in the Strategy Card Game AI competition held at the IEEE CoG 2019 conference from the tenth to the fourth position.

In summary, the contributions of this work are:

- A game-agnostic Markov decision process formulation of the task of deck building in the arena mode of collectible card games, including an alternative state representation.
- A deep reinforcement learning methodology for finding policies for the aforementioned Markov decision process;
- A fully-working game engine and OpenAI Gym environments encompassing LOCM's deck building and playing tasks;[2]

[2] `https://github.com/ronaldosvieira/gym-locm`

- A collection of ready-to-use competitive trained draft strategies that can be integrated with any LOCM bot;[3]
- A short paper published in the proceedings of the 18th Brazilian Symposium of Computer Games and Digital Entertainment (SBGames 2019), with some preliminary results [Vieira et al., 2019].
- A full paper published in the proceedings of the 19th Brazilian Symposium of Computer Games and Digital Entertainment (SBGames 2020), describing this research, which was awarded as the best paper in the Computing Track [Vieira et al., 2020].
- A LOCM-playing bot submitted to the Strategy Card Game AI competition, held at the 2020 IEEE Congress on Evolutionary Computation (CEC), featuring our best draft strategies, which achieved the third place.[4]

1.3 Chapter organization

This thesis is organized as follows:

- In Chapter 2, we present the necessary background for this thesis, including a more detailed description of collectible card games and the necessary background on reinforcement learning, neural networks and deep reinforcement learning.
- In Chapter 3, we discuss related literature on collectible card games, addressing work on deck building, battling, and other relevant topics, and positioning our work with regards to them.
- In Chapter 4, we describe our methodology, formulating the problem as a Markov decision process, presenting our deep reinforcement learning approaches, instantiating our methodology on *Legends of Code and Magic* and discussing its instantiation on other CCGs.
- In Chapter 5, we perform and discuss experiments to validate our approaches.
- In Chapter 6, we conclude the thesis by summarizing our methodology, results and contributions, as well as discussing directions for future research.

[3] `https://github.com/ronaldosvieira/gym-locm/tree/1.0.0/gym_locm/trained_models`
[4] `https://github.com/ronaldosvieira/reinforced-greediness`

- In Appendix A, we document the *gym-locm* repository on GitHub, which contains our implementation of *Legends of Code and Magic*, the OpenAI Gym environments, final models and several utility scripts.
- In Attachment A, we describe all hyperparameters we optimize and list their final values.

Chapter 2

Background

In this chapter, we discuss the concepts used throughout the thesis, including collectible card games (Section 2.1), classic reinforcement learning (Section 2.2), neural networks (Section 2.3) and deep reinforcement learning (Section 2.4). This discussion is not meant to be a comprehensive review on each topic, but rather a vertical cut on all subjects that are relevant to the thesis.

2.1 Collectible card games

Collectible card games (CCGs), such as *Magic: the Gathering* and *Hearthstone*, are adversarial two-player games played by tens of millions of players worldwide. Apart their usually large set of rules and challenging learning curve for human players,[1] they also pose many challenging research problems for artificial intelligence (AI). Besides playing the game, AI can help, for instance, in content generation, difficulty scaling and game balancing [Yannakakis and Togelius, 2018; Hoover et al., 2020]. Fueled by the recent successes in other types of games, academic interest in CCGs has increased in the last few years, as measured by the amount of published work on the subject.

In CCGs, the player is required to build a deck from a set of available cards, which often represent creatures, items or spells from a fantasy world, and use it to battle other players. In battles, each player starts with a certain amount of cards in their hand, drawn from their shuffled deck. With their playing order decided, they take turns in which they may play cards (by paying their mana cost), use abilities of played cards or attack with creature cards. As turns pass, players get an increasing amount of mana points, thus being able to play more powerful cards (the stronger the card,

[1] As of the time of writing, the English version of *Magic: the Gathering*'s comprehensive rules book has 242 pages.

the higher its mana cost). Players do not know which cards are in their opponent's hand and deck nor the card ordering of their own shuffled deck. The game ends when a player successfully reduces their opponent's health points to zero or any of the other (usually many) win conditions are achieved.

The turn-based structure in CCGs leaves the second player at disadvantage. At the time the second player have their k-th turn, the first player has had k turns worth of preparation to counter whatever moves are made by their opponent. On the other hand, in the first player's k-th turn, their opponent has had only $k-1$ turns to prepare. CCGs mitigate this imbalance by having the second player draw an additional card and/or get an additional mana point at the beginning of the match. From a strategic standpoint, this makes playing first and second significantly different games.

What is referred as "playing" in CCGs often combine two different tasks: constructing a deck and battling other players. In the remainder of this thesis, we refer to them as separate AI problems: deck building and battling. They are interdependent AI tasks in the sense that the performance of a deck-builder AI depends on the battler's skills and style, while the performance of a battler AI depends on the strategy and overall power of the decks it uses.

Apart from the battler, the quality of a deck can be influenced by several factors, such as:

- *Mana efficiency.* Since mana correlates to card power, and playing more powerful cards than the opponent correlates to a greater probability of winning, it is paramount to maximize the amount of mana spent every turn [Cunningham, 2007a]. In other words, this means observing the distribution of card costs in the deck (i.e., the mana curve) to maximize the probability of having adequate cards to play every turn.

- *Deck consistency.* In CCGs, there are many possible strategies to win battles. However, to optimize the probability of winning over all possible starting hands and card draws, decks usually benefit of being coherent to a single main strategy and having all cards support that strategy [Koska, 2010; DoubleXP, 2014].

- *The metagame.* Decks tend to perform better against some types of decks and worse against others, as CCGs are designed to not have a single dominant deck strategy. Thus, the performance of a deck depends on which decks are used by the opponents. This implicit probability distribution of opponent decks is called the metagame [Cunningham, 2007b].

In most CCGs, following the principle of deck consistency, deck strategies are grouped into three main archetypes: aggro, control and combo. Aggro decks attempt to deal damage to the opponent and win as soon as possible, while control decks try and neutralize the enemy threats until it is able to use powerful, usually slower game finishers, and combo decks exploit a specific interaction between two or more cards that leads to an instant win.

Deck building is done differently depending on the game mode. In the most common ones, often called constructed modes, players build decks using any combination of cards from a large predetermined list of permitted cards, usually with a limit on the maximum amount allowed of copies of the same card. In these modes, decks are built in an offline manner, that is, at the time of play, decks are already built.

In the arena mode, on the other hand, players are required to build a deck by *drafting*, that is, selecting one card at a time from a few randomly presented candidates. After a deck is built, the player uses it to battle different opponents until a predetermined amount of losses is reached, and is rewarded proportionally to the amount of wins obtained. Since the players have limited card options and usually limited time to choose, decks are less optimal than in constructed modes.

Drafting can be viewed as a set of sequential decisions that results in a noisy outcome (amount of wins), which the player intends to maximize, and every decision and combination of decisions affect the outcome to some unknown degree. These features characterize the type of problems that are typically tackled by *reinforcement learning*.

2.2 Reinforcement learning

Reinforcement learning (RL) is a framework that models sequential decision problems, which are solved by refining a decision-making strategy (i.e., a policy) in a trial-and-error fashion, using a numerical reward signal as feedback on each of its decisions [Sutton and Barto, 2018]. Its roots can be traced back to the Law of Effect, from the studies of learning in animals, which states that behavior succeeded by satisfactory outcomes (i.e., positive rewards) become more likely to be repeated, given the same situation, while ones succeeded by discomforting outcomes (i.e., negative rewards) become less likely to be repeated [Thorndike, 1911].

RL requires problems to be formulated as a Markov decision process (MDP). In MDPs, a decision-maker *agent* interact with its environment (i.e., the decision problem) by observing its current *state* and performing an *action* based on it, which results in

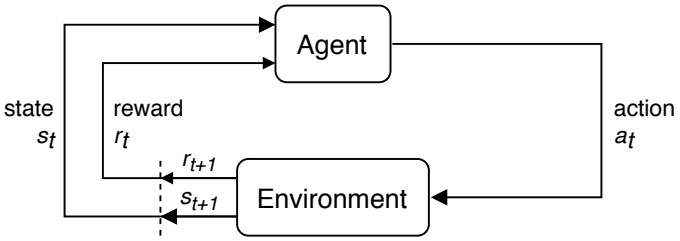


Figure 2.1: The agent-environment interaction in a Markov decision process. At time step t, the agent receives a reward r_t and a new state s_t from the environment, and, based on them, performs the action a_t. This leads to it receiving a new reward r_{t+1} and a new state s_{t+1} at time step $t+1$. Adapted from Sutton and Barto [2018].

a *reward* signal, indicating the immediate feedback of performing that action, and the resulting state of the environment, which is then used by the agent to act again. Considering the *experience tuple* containing a current state s, an action a, a reward r and a next state s' for each time step in the process, this interaction loop, depicted by Figure 2.1, results in a sequence $s_0, a_0, r_1, s_1, a_1, ..., a_{N-1}, r_N, s_N$. This sequence is called an *episode*, and starts in a *initial state* s_0 and ends whenever the agent reaches a *terminal state* s_N of the environment.[2]

Formally, a MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$ in which $\mathcal{S}$ is the set of possible states of the environment can be in, $\mathcal{A}$ is the set of available actions the agent can perform, $T(s'|s,a)$ is a function that describes the probability of getting to state $s' \in \mathcal{S}$ as a consequence of the agent performing action $a \in \mathcal{A}$ while in state $s \in \mathcal{S}$, and $\mathcal{R}(s)$ is a function that describes the reward obtained at every state $s \in \mathcal{S}$.[3] The states in a MDP should satisfy the Markov property, that is, every state should contain all information necessary to determine what the probability distribution over next states is, given an action. In other words, it should be true that

$$T(s_{t+1}|s_t, a_t) = T(s_{t+1}|s_0, s_1, ..., s_{t-1}, s_t, a_t), \forall t \in [0, N].$$

The goal of the agent is then to learn a *policy* $\pi(a|s)$ that maps every state to a probability distribution over all possible actions, such that it maximizes the expected reward obtained in an episode.

When learning a policy, RL agents face the *exploration versus exploitation* dilemma: within a finite amount of timesteps, it needs to balance choosing the actions

[2]In this thesis, we focus on episodic tasks. However, there can be infinite tasks, for which the concepts of episode and terminal states do not apply.

[3]In literature, the reward function is sometimes also formulated as $\mathcal{R}(s,a)$. The two formulations are equivalent, because the expected reward of an action is the reward of the possible successor states, weighted by the probabilities of reaching them.

it considers the best, to maximize rewards (exploiting), and choosing unknown actions that may lead to even better rewards (exploring). RL algorithms usually tackle this by employing some kind of noise to its policy when interacting with the environment, to ensure every state has a nonzero probability of being visited.

Traditional RL algorithms, such as Q-Learning [Watkins, 1989], store their policies and their estimated expected rewards of all states in memory. While this is possible for smaller problems, when dealing with larger ones such as drafting in collectible card games, with too many states to store in memory, the most frequent solution is to use function approximators, such as neural networks, to represent the policies and estimates of expected rewards.

2.3 Neural networks

Artificial neural networks (ANN), or just neural networks, are a machine learning technique that alludes to the functioning of a brain. Their first appearance on literature was in the form of a mathematical model of a single neuron, named *perceptron* [Rosenblatt, 1958]. Then, following developments led to the stacking of interconnected layers of perceptrons, that is, multilayer perceptrons (MLP) [Pal and Mitra, 1992], as a way to solve harder problems than those a single neuron could solve.[4]

Formally, an individual perceptron in a MLP works by receiving k numerical signals $\mathbf{x} = \{x_1, x_2, ..., x_k\}$ as input (that may come from other neurons), performing a weighted sum of these signals using different *weights* $\mathbf{w} = \{w_1, w_2, ..., w_k\}$ for each of them, and passing the result plus a *bias* term b to an *activation function* f, that usually performs a non-linear operation on it. The output, y, is then passed as input to other neurons or used as output of the network. Treating the inputs and weights as vectors, the entire computation can be described as $y = f(\mathbf{x} \cdot \mathbf{w} + b)$. Figure 2.2 shows the common depiction of a perceptron.

The entire MLP consists of n layers of multiple perceptrons, each layer containing m_i perceptrons. It works by receiving k numerical signals $\mathbf{x} = \{x_1, x_2, ..., x_k\}$ as input, which are fed to every neuron in the first layer. The m_1 neurons in the layer produce $\mathbf{x}_{(1)} = \{x_1, x_2, ..., x_{m_1}\}$ as output, which is then fed to every neuron in the next layer, repeating the process until the last layer is reached. The output layer then produces $\mathbf{x}_{(n)} = \mathbf{y} = \{y_0, y_1, ..., y_{m_n}\}$, the final output. Treating the inputs of a layer i as a vector

[4]The most notable limitation of the perceptron is that it could only solve linearly separable problems, that is, problems that are solvable by splitting the two-dimensional problem space with a single line (or with a hyperplane, if it has more than two dimensions). Simulating an XOR logic gate, for instance, is not a linearly separable problem, and, therefore, is not solvable by perceptrons.

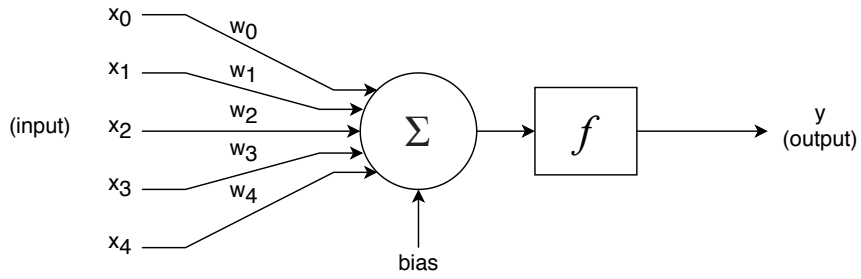


Figure 2.2: Typical structure of a neuron in a neural network.

$\mathbf{x}$ of m_{i-1} elements and the weights and bias terms of all neurons in that layer as a $m_{i-1} \times m_i$ matrix $\mathbf{W}$ and a vector $\mathbf{b}$ of m_i elements, respectively, the entire computation of a layer can be described as $\mathbf{y} = f(\mathbf{x} \cdot \mathbf{W} + \mathbf{b})$, where f applies the neurons' activation function element-wise and $\mathbf{y}$ is the resulting output vector, containing m_i elements.

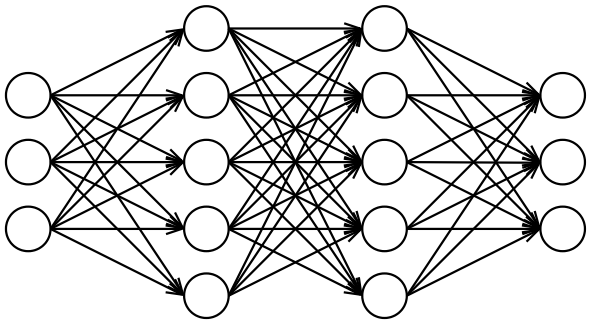


Figure 2.3: Typical structure of a neural network with $n = 4$ layers. Circles depict neurons, and arrows depict the output of neurons being passed as input to others. The leftmost layer is the input layer. While depicted with circles, they represent only the input values and are not neurons. The rightmost layer is called the output layer, and the remaining ones between input and output are called hidden layers.

As described above, the final output of a neural network is determined by their neurons' weights and biases, as well as its architecture (e.g., number of layers and amount of neurons) and activation functions. Although there are many ways to train a neural network, the most widely used form of training is by *gradient descent*, that is, by iteratively updating the weights and biases as to reduce the error between the expected output and the actual output of the network (given by some *loss function*, such as the mean squared error [Sammut and Webb, 2010]). The trainable units (weights and biases) are often referred to as the network's *parameters*, while the remaining factors that influence the training process (such as the activation functions, the network architecture and the loss function) are called the network's *hyperparameters*.

The architecture of a network has a heavy influence on the network's capacity to solve problems. As layers of perceptrons were preferred over a single perceptron, deeper

and wider networks[5] are also preferred over shallower and narrower ones when the complexity of the problem being tackled is higher. However, using architectures that are too deep and/or wide or training for too long makes the network prone to *overfitting*, that is, be extremely accurate on the inputs given during training yet not achieving such accuracy on inputs not seen during training.[6] Training deep networks also has other unique problems and characteristics, which are studied in the recent *deep learning* field.

Some kinds of problems are best solved with different types of network architectures. For instance, the advent of convolutional neural networks (CNN) [Fukushima and Miyake, 1982] enabled better performance on tasks involving spatial data, such as digital images, and that of recurrent neural networks (RNN) [Rumelhart et al., 1985] provided better suited techniques to dealing with temporal sequences, such as text and voice. Differently from MLPs, RNNs are stateful – the output of a network is determined by both the current input and the previous inputs. This is achieved by using recurrent neurons which keep an internal memory and learn to encode the last inputs received on it.

Long short-term memory (LSTM) is a notable type of recurrent neuron proposed by Hochreiter and Schmidhuber [1997] which overcame the problem of vanishing gradients present on other recurrent neuron types and became the state-of-the-art on tasks featuring some form of temporal dependency. In contrast to the simplicity of a perceptron's weighted sum, LSTMs perform many operations to the input data to decide which information is going to be saved in or removed from its memory.

The allied power of neural networks (of all types of architectures), especially deep neural networks, with the reinforcement learning paradigm enabled several advancements in the state-of-the-art of game AI research, giving birth to the *deep reinforcement learning* field.

2.4 Deep reinforcement learning

Deep reinforcement learning (DRL) can be understood as the adaptation of the successful deep learning techniques to reinforcement learning settings, in which the classic RL methodologies do not suffice or present limited performance. A large part of the recent milestones in game-playing AI were achieved using DRL methods [Silver et al.,

[5] Deeper and wider in the sense of having more layers and more neurons per layer, respectively.

[6] A common metaphor to an overfit network is a student that memorized the answer to all her math exercises without understanding them, and failed her math exam because it contained different yet very similar exercises to the ones she memorized.

2017; Berner et al., 2019], and it has been also present in other domains of research on games such as content generation [Khalifa et al., 2020; López et al., 2020].

The ascension of DRL started with the introduction of the Deep Q-Network (DQN) [Mnih et al., 2013], a variation of the classic Q-Learning approach that uses a deep neural network to learn Q-values (quality values) for the available actions in each visited state and to generalize over unvisited states. It was used to play several Atari 2600 games using raw pixel matrices as inputs, and outperformed all previous approaches on six out of the seven games they played. In a later publication, the DQN was shown to outperform a professional human game tester in 29 out of 49 Atari 2600 games [Mnih et al., 2015].

To attenuate the learning instability displayed by previous attempts on DRL, the DQN used experience replay [Adam et al., 2012] – the generated experience tuples were stored in a limited-size buffer, from which batches of random tuples would be sampled to participate in training. With time, several extensions and variations to the original DQN algorithm were proposed [van Hasselt et al., 2016; Schaul et al., 2016; Wang et al., 2016], some of which were later combined in the Rainbow DQN [Hessel et al., 2018].

Policy gradient methods are a parallel thread of DRL algorithms that output a probability distribution over actions (i.e., the policy) instead of Q-values. The Trust Region Policy Optimization (TRPO) [Schulman et al., 2015] algorithm is an approximation to an iterative method that is mathematically guaranteed to yield better or equally-performing policies at each iteration. The guarantee relies on not allowing too large updates in a way that could collapse the policy's performance, implemented by constraining the maximum KL-divergence[7] between the old and new policies in each iteration.

The Proximal Policy Optimization (PPO) algorithms [Schulman et al., 2017] propose a simpler and sample-efficient approach: instead of constraining updates by the KL-divergence, either (i) use the KL-divergence to penalize too large updates, or (ii) clip too large updates up to a maximum absolute value (defined as a hyperparameter). For its easier implementation and sample-efficiency, PPO is currently the go-to DRL algorithm and is available in the most used reinforcement learning libraries,[8] often along with recurrent, parallel and asynchronous variations.

[7]Kullback-Leibler divergence, or KL-divergence, is a metric of divergence between two probability distributions [Kullback and Leibler, 1951].

[8]OpenAI Baselines - `https://github.com/openai/baselines/tree/master/baselines/ppo2`, Stable Baselines - `https://stable-baselines.readthedocs.io/en/v2.10.0/modules/ppo2.html`, RLlib - `https://docs.ray.io/en/releases-1.0.1/rllib-algorithms.html#ppo`

Chapter 3

Related work

In this chapter, we review the available literature on the tasks of building decks (Section 3.1) and battling (Section 3.2) in collectible card games (CCGs), as well as other work on AI on the topic (Section 3.3). Lastly, we summarize the reviewed approaches and situate our work in comparison to the state of the art regarding the problem of drafting in arena mode and of CCG deck building in general (Section 3.4).

3.1 Deck building

Deck building in all game modes of CCGs is dominated by evolutionary approaches. In those, each individual represents a fixed-length deck, with each gene serving as a card slot. Fitness is based on either the win rate [García-Sánchez et al., 2016, 2018; Bjørke and Fludal, 2017] or the health difference [Bhatt et al., 2018] of a fixed battle agent using the respective deck either in a round-robin tournament with the entire population or against a selection of established decks. Most notably, Fontaine et al. [2019] propose a novel variation of the MAP-elites [Mouret and Clune, 2015] algorithm to evolve high-performance decks while also ensuring diversity among them, preventing the common outcome of evolutionary algorithms which converge to a single deck archetype.

A reinforcement learning approach is proposed by Chen et al. [2018], that models the deck building problem as a Markov decision process whose states represent complete decks, and actions represent the replacement of a card in the deck. Starting from a random deck, the Q-learning algorithm is used together with a neural network to approximate the optimal policy of card substitutions, obtaining competitive results. A different approach, proposed by Góes et al. [2016], aims to find creative card combos rather than building a full deck. They use the honing theory framework, which relies on the distributed nature of human memory and the analytic-associative dichotomy

of reasoning to explain creativity. Card concepts and interactions between them are mapped into a graph, which is traversed in a way that simulates the process of creative thought to find sets of cards that interact positively with each other (i.e., card combos).

Given a card pool, the aforementioned deck-building approaches search the space of all possible decks looking for well-performing ones. Such approaches are not suitable for the arena mode, where the space of possible decks is not known beforehand. Instead, research on arena focuses on finding a more general solution, that is, a draft strategy.

To the best of our knowledge, there is a single work that specifically addresses drafting in arena mode: Kowalski and Miernik [2020] tackle the problem with an evolutionary algorithm, where each individual represents a ranking of all available cards instead of a deck, with each of its genes carrying the priority value of a specific card. The resulting draft strategy is to choose the card with the highest priority value on each draft round. The fitness of an individual is represented by the win rate of its draft strategy in a tournament containing either the entire population or selected draft strategies. The standard uniform cross-over and mutation operators are used, but only modify genes whose respective cards appeared in at least one match in the fitness calculation, as these cards are the ones responsible for the resulting fitness.

Other unpublished work on arena drafting involve either estimating rankings of cards by analysis of thousands of match replays, picking cards to pursue an optimal distribution of resource costs or a combination of both.[1] The Hearth Arena website[2] provides an arena draft helper, guiding human players to draft cards according to card rankings and synergy lists maintained by the *Hearthstone* community.

3.2 Battling

Currently, the best approaches for battling in collectible card games use tree-search methods. Monte Carlo tree search (MCTS) [Browne et al., 2012] is the preferred technique, followed by alpha-beta search. In the game tree built by these battle agents, the nodes represent game states while edges link all possible actions of a state (e.g., playing a card, attacking, or passing the turn) to their respective resulting states. An exception is made by Cowling et al. [2012], which reduces playing *Magic: the Gathering* to a series of binary yes-no decisions before applying the MCTS algorithm.

The hidden information present in CCGs is not handled by tree-search methods by default. In the CCG literature, most approaches ignore it, but some efforts were

[1] `https://www.codingame.com/forum/t/legends-of-code-magic-cc05-feedback-strategies/50996/63`

[2] https://www.heartharena.com

made to determinize[3] [Cowling et al., 2012] or predict [Dockhorn et al., 2018] it. Alpha-beta search and some MCTS approaches use a state evaluation function that frequently consists of hand-made heuristics, but can also be machine-learned [Swiechowski et al., 2018; Wang and Moh, 2019] or optimized by evolutionary algorithms [Le, 2019].

Many other unpublished battle agents are part of non-official implementations of CCGs, usually made by the players community. They serve as sandboxes to players, where all cards and modes are available, and the battle agents may be used as adversaries to them or other AIs (such as in Fireplace,[4] Sabberstone[5] and Magarena[6]), or as playtesters for human-made decks (such as in Firemind[7]).

3.3 Other work on CCGs

Research on CCGs has also tackled problems other than deck building and battling. For instance, Bursztein [2016] achieves up to 95% of accuracy in predicting the next move of opponents in *Hearthstone* by exploiting the decks in the metagame (see Section 2.1). It shows that opponent prediction might be accurate enough to be integrated in deck building and playing approaches, if the metagame is stable enough.

De Mesentier Silva et al. [2019] use evolutionary algorithms to help in game balancing. They aim to find the fewest changes in card attributes that would balance an unbalanced metagame, that is, that would bring the average win rate of a set of decks closer to 50% when playing against themselves. Work in this direction can reveal what combination of attributes are more influential in the cards' power. On the inverse direction, Zuin and Veloso [2019] use machine learning to predict the mana cost of a card given its attributes. While this also aids CCG designers in balancing the game, the methodology of feature extraction for *Magic: the Gathering* cards introduced by the authors may also be useful in further work on deck building and game playing for the game.

A broader review of the many artificial intelligence challenges provided by *Hearthstone* (that are also present in most CCGs) and their current literature is presented in the work of Hoover et al. [2020].

[3] In this context, determinization means: before each iteration of the tree-search algorithm, assigning random values to all unknown information from the set of possible values they can assume, and proceeding the iteration as if the problem was deterministic and had complete information.

[4] `https://hearthsim.info/fireplace`

[5] `https://github.com/HearthSim/SabberStone`

[6] `https://magarena.github.io`

[7] `https://www.firemind.ch`

3.4 Summary

Despite the decades-long success of collectible card games, most work using artificial intelligence in the genre dates to the last five years. The appearance of digital CCGs that provide more accessible test environments than their traditional in-paper counterparts may be one of the causes. Another possibility is that the recent advances in the field and in computational power may have encouraged work on games of otherwise prohibitive complexity, such as CCGs.

Although the published work on deck-building can improve deck building, a comparison between them is yet to be done. This is not a trivial task since each approach optimizes decks for different battle agents and is biased to the battlers' capacities and playing styles.

Most of the work on LOCM's drafting and battling phases were developed for the past LOCM-based AI competitions. Two competitions used the CodinGame platform,[8] while the others were held by the CEC and COG conferences. Although none of the approaches were published, a discussion of some of them can be found in the CodinGame competitions post-mortem forum,[9] alongside valuable insights on the game.

As mentioned above, only a single work regarding the arena draft mode of CCGs has been published so far. We aim to explore this subject further. Our research differ from the current literature on the following points:

- Evolutionary approaches are currently the preferred methods for deck building in CCGs. While they suit the problem and yield good results, we investigate whether deep reinforcement learning algorithms can produce competitive decks, following their previous successes in other domains.
- Our methodology is formulated in a game-agnostic manner to make it easier to port to other CCGs. Current literature usually commits to a single game.
- To the best of our knowledge, no approaches on drafting (either published or unpublished) explicitly consider synergies with previously picked cards when picking a new card, usually relying solely on card power. We propose approaches that try to exploit synergies between cards (as well as one approach that does not).
- Our approaches operate using the features of the cards rather than their IDs, learning what a good card is like instead of which cards are good. We expect this to yield more robust and reusable drafters.

[8] `https://www.codingame.com`

[9] `https://www.codingame.com/forum/t/legends-of-code-magic-cc05-feedback-strategies/50996/63`

Chapter 4

Methodology

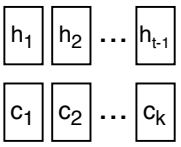
In the arena mode of collectible card games (CCGs), the player participates in a draft through which they build the deck they will use to play the subsequent set of matches. In this thesis, we propose a game-agnostic deep reinforcement learning methodology to find draft strategies that maximize the win rate of a specific fixed battle agent. In this chapter, we provide a formulation of the deck building task in the arena mode as a Markov Decision Process (MDP) (Section 4.1) and describe our three proposed agents to tackle it (Section 4.2). Then, we instantiate our methodology on *Legends of Code and Magic* (Section 4.3) and finish with a discussion how to use it on other CCGs (Section 4.4).

4.1 Problem formulation

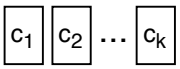
Let $\mathcal{C}$ be the set of available cards in the arena mode of a collectible card game. In that arena mode, the player builds their deck incrementally in n turns. At each turn, the game presents k cards to the player, sampled without replacement from $\mathcal{C}$ under some probability distribution defined by the game itself (usually a uniform distribution). From the sample of k cards, the player must pick a card to add to their deck. The player does not know which cards will appear in the future, and, in addition, we assume that the player does not know the cards presented to or picked by their opponents. At the end of n draft rounds, the complete n-cards deck is used in a set of battles, which is used to assess its performance. We define this problem as a (k, n)-*draft*.

To apply reinforcement learning, we formulate the (k, n)-draft problem as a Markov decision process, taking the point of view of a player in a draft. The resulting MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$ whose elements are defined as follows:

- **Set of states.** Every state represents a possible turn in the draft, and consist of the features of every card present in that turn. We define two different forms of state representation which differs in how many cards are considered as present in a turn. A history-aware state representation includes the current k card alternatives in the current turn as well as all previously chosen cards by the player in the previous turns, that is, the deck built so far. A simpler history-oblivious state representation considers only the current card choices, disregarding the past picks. We denote their resulting set of states $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. Figure 4.1 shows the difference between the states in each of those sets.



(a) A state in $\mathcal{S}_1$.



(b) A state in $\mathcal{S}_2$.

Figure 4.1: States in each of our state representations at turn t of a draft. On the left, states in $\mathcal{S}_1$ contain the k current card choices $(c_1, c_2, ..., c_k)$ as well as the $t-1$ cards picked in the previous turns $(h_1, h_2, ..., h_{t-1})$. On the right, states in $\mathcal{S}_2$ contain only the k current card choices. Each card is represented by a set of features extracted from it.

In the history-aware representation, the resulting set of states, $\mathcal{S}_1$, is large: at turn t, the agent has already picked $t-1$ cards in the previous turns, yielding $|\mathcal{C}|^{t-1}$ possible combinations, and has $\binom{|\mathcal{C}|}{k}$ possible combinations for the current choices. Considering n turns, therefore,

$$|\mathcal{S}_1| = \prod_{t=1}^{n} \binom{|\mathcal{C}|}{k} |\mathcal{C}|^{t-1}.$$

The subset of starting states in $\mathcal{S}_1$ contains any state in which there are no past picks, while the set of terminal states contains any states in which there are no current choices and n cards have already been picked.

The history-oblivious representation disregards the past picks, resulting in a much smaller state space:

$$|\mathcal{S}_2| = \binom{|\mathcal{C}|}{k}.$$

However, this reduction comes at the cost of not being able to consider synergies with previously picked cards and, therefore, choosing only by card power. In $\mathcal{S}_2$,

since past picks are never considered, all states can be starting states. The set of terminal states contains only the single state in which there are no current choices.

- **Set of actions.** The set of possible actions is the same for all states, and consists in choosing one of the k presented cards, which are represented by their indexes. Formally,

$$\mathcal{A} = \{1, ..., k\}.$$

- **Transition function.** The transition function $T(s'|s, a)$ follows the logic of the drafting process: in s', a new sample of k cards is presented if s was not the last draft round (less than n draft rounds have passed). If $\mathcal{S}_1$ is being used, s' also contains the $t - 1$ previously picked cards. Figure 4.2 describe a transition between turns.

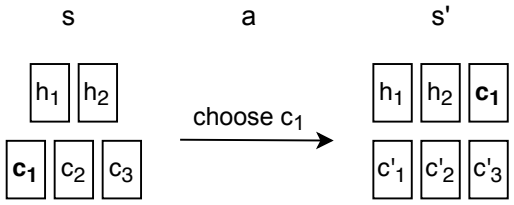


Figure 4.2: In a draft with $k = 3$, when s is a state with two cards picked so far and a is an action that chooses card c_1, the resulting state s' should contain a new sample of k cards, and, if using $\mathcal{S}_1$, the previously picked cards of s plus c_1.

- **Reward function.** We reward the agent proportionally to the win rate of the built deck in battles played by a fixed battle agent. Since in non-terminal states the deck is not complete yet, the reward in these states is 0. In terminal states, when the draft has ended and the deck is complete, a nonzero reward is given. Formally,

$$\mathcal{R}(s) = \begin{cases} 0, & \text{if } s \text{ is not terminal,} \\ \frac{2w}{b} - 1, & \text{if } s \text{ is terminal and } w \text{ out of } b \text{ battles were won.} \end{cases}$$

  This way, a sequence of draft choices that yields a win rate of 100% would be followed by a reward of 1 in the terminal state, whereas one which results in a win rate of 0% would be followed by a reward of -1.

- **Discount factor.** A discount factor is not used (i.e., $\gamma = 1$), as earlier or later choices of cards may have the same influence on winning or losing the succeeding

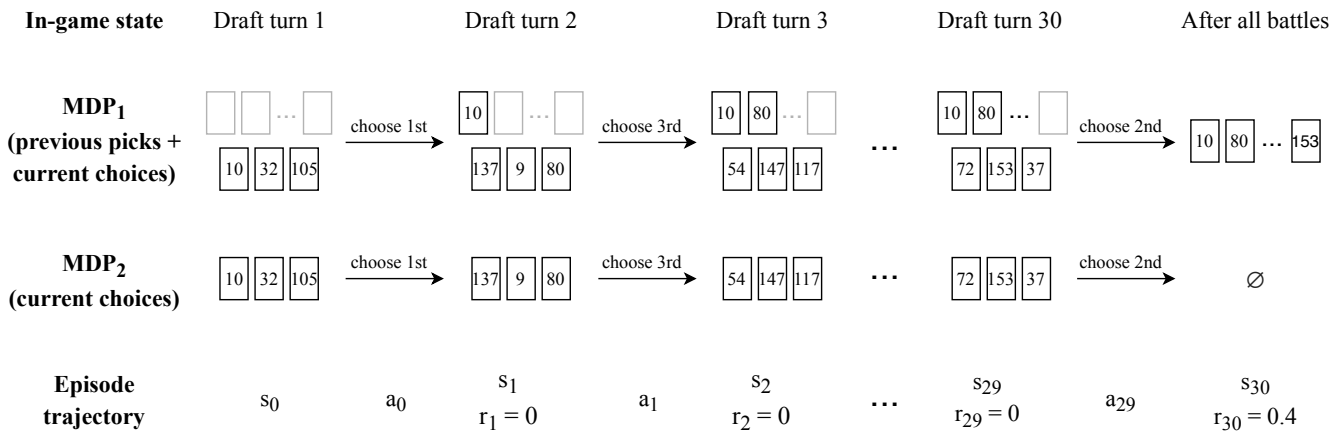


Figure 4.3: A sample episode in our MDPs when modeling a $(3, 30)$-draft. Each numbered rectangle represents a different card in the game. Each empty rectangle represents an empty card slot. Each state holds the three current choices of cards, plus all previously picked cards in those of MDP_1. Actions correspond to picking one of the cards. A nonzero reward is given at the terminal state representing the win rate of the battle agent with the drafted deck. In this case, 70% of battles were won, yielding a reward of 0.4.

matches. In other words, we have no reason to consider cards chosen in the first draft turns as less important than those chosen in the last draft turns.

We define MDP_1 and MDP_2 as the Markov Decision Processes composed of the set of actions, transition function, reward model and discount factor defined above, using, respectively, the $\mathcal{S}_1$ and $\mathcal{S}_2$ state representations. Figure 4.3 depicts a sequence of states, actions and rewards of a sample episode in both MDPs when representing a $(3, 30)$-draft. Finding a solution to either MDP is equivalent to developing a strategy to draft decks in arena mode of a CCG with the specified k and n parameters. In other words, our goal is to find a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that maps every possible state to a probability distribution over actions in a way that maximizes the expected reward (and, consequently, the win rate in the battles). Our resulting draft agent would then use this policy to pick a card in each draft turn.

4.2 Proposed approaches

We tackle the arena drafting MDPs with deep reinforcement learning (DRL). The DRL agent takes the place of a player in the draft, receiving the game state as input on every turn, and producing one of the actions as output. Once the deck is built, a fixed battle agent plays a set of matches using it, and we use the result as feedback on whether the

choices of cards were good. Figure 4.4 depicts the interaction between the DRL agent and the game, as well as its components.

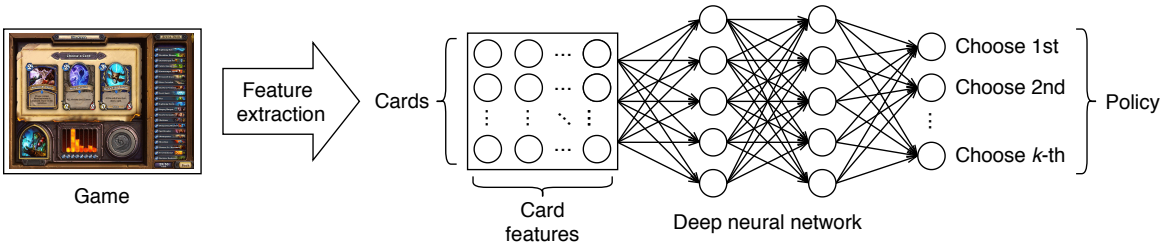


Figure 4.4: The interaction between our agent and the game. From the game, we extract features of the cards in the current draft turn. The features of all cards contained in the state representation are concatenated and fed as input to a deep neural network, which outputs values from which we build a policy. This policy is then used to act in the game (i.e., choose a card).

We propose three approaches that differ in state representation and the type of neural network used by the DRL algorithm. The first variant, *History*, uses a multi-layer perceptron (MLP) network to tackle MDP_1. As discussed in the previous section, MDP_1 encodes all previously picked cards into the state representation, enabling the agent to consider synergies with them when choosing the next card. The second variant, *LSTM*, tackles MDP_2 but relies on a layer of long short-term memory (LSTM) [Hochreiter and Schmidhuber, 1997] units to retain information about past picks without explicitly enumerating them. The last variant, *Immediate*, tackles MDP_2 with a MLP architecture, not considering past picks but reasoning in a much smaller state space. Despite the differences, all approaches share the same training methodology.

In each draft turn, the game state is given as input to the network, which outputs the index of its chosen card. Since neural networks use numeric features only, the game state should be converted to a network-friendly representation, that is, a numeric vector. This representation contains the relevant features (e.g., attack power, mana cost, abilities) of all current card choices and previously picked cards (for MDP_1). The conversion process includes normalizing all numeric or ordinal card features into the $[-1, 1]$ interval and transforming all categorical card features into one-hot encoding. Empty card slots (such as cards not yet picked) are represented by a zero vector.

After the last draft choice in the episode, the battles are conducted by a fixed battle agent using the built deck. Since a large amount of episodes is often needed for DRL algorithms to properly train a network on a complex task, the selection of battle agent should consider a trade-off between performance and computational cost. In other words, the battler should be complex enough to be able to take advantage of

high-performance decks and fast enough so thousands of matches can be played in a timely fashion.

4.3 Game-specific aspects

We have left some game-specific aspects open: the values of k, n and $\mathcal{C}$, the feature extraction, the battle agents used in battles, the opponents, and a game engine to run the draft and battle. Hereafter, we fill these gaps by instantiating the methodology in a collectible card game called *Legends of Code and Magic*.

4.3.1 Testbed

Legends of Code and Magic is a collectible card game created by Kowalski and Miernik in 2018 with the goal of making AI research on CCGs easier. It implements a small yet representative subset of the rules found in established commercial CCGs, enabling much faster testing of algorithms. Differently from most games, integration with AI players is trivial and does not require the use of any middleware library. Moreover, deck building in LOCM is performed exclusively in an arena-like fashion, making it a good testbed for our experiments.

A match of LOCM consists of two phases: draft and battle. In the draft, for $n = 30$ turns, players pick one out of $k = 3$ randomly presented cards to form a deck and is, therefore, equivalent to a $(3, 30)$-draft. Both players are presented the same cards and they cannot observe the choices of each other. In the battle phase, the actual play occurs. The players start with four cards in their hand and draw one more on each turn. They also start with one mana point, which is recharged and incremented by one on each turn. To balance the game, the second player gets an additional card and a non-rechargeable additional mana point at their first turn.

Using mana, the players then take turns placing creature cards in the board, as well as using item cards to make them stronger or weaker in a variety of ways. Creatures can be used to attack the opposing player or their creatures. The game ends when a player has successfully reduced their opponent's health points from the starting amount of 30 to zero.

The game is finite: each player can have a maximum of eight cards in their hand and three creatures on each of the two lanes available. The draft phase has exactly 30 turns, and the battle phase is guaranteed to take at most 55 turns. There are $|\mathcal{C}| = 160$ available cards, and they all have the same set of attributes, as described in

Attribute	Value range	Description
Id	$[1, 160]$	Card identifier.
Name	Text	Card name.
Type	Creature, green item, red item or blue item.	Type of the card. Determines which actions can be done with them.
Cost	$[0, 12]$	How much mana is necessary to summon (if creature) or use (if item) the card.
Attack	$[-2, 12]$	On creatures, determine how much damage is done by the card when attacking. On items, determine how much attack will be given (or taken, if negative) to the target creature when using.
Defense	$[-99, 12]$	On creatures, determines how much damage it can endure. A creature with zero or negative defense is removed from the game. On items, determines how much defense will be given (or taken, if negative) to the target creature when using.
PlayerHP	$[-3, 5]$	How much health points will be given (or taken, if negative) to the current player when the card is played.
EnemyHP	$[-5, 1]$	How much health points will be given (or taken, if negative) to the opposing player when the card is played.
CardDraw	$[0, 2]$	How many additional cards will be drawn by the current player when the card is played.
Breakthrough	True or false	Card abilities. On creatures, determine whether it has each of the abilities. On items, determine what abilities will be given (if green item) or removed (if red or blue item) to target creature when using.
Charge		
Drain		
Guard		
Lethal		
Ward		

Table 4.1: Attributes of cards in *Legends of Code and Magic*.

Table 4.1. Also, the only nondeterminism in the game is the shuffling of the decks in the beginning of the battle.

These characteristics make playing LOCM proficiently a far more attainable feat for AI agents in comparison to other CCGs and, therefore, make LOCM a good first step in the pursuit of superhuman CCG-playing and deck building agents. The use of LOCM in AI research is encouraged by the Strategy Card Game AI (SCGAI) competition, held yearly at the IEEE CEC and IEEE CoG conferences.

4.3.2 Game engine and OpenAI Gym environments

To speed-up our experiments and facilitate further research on LOCM, we developed an open-source re-implementation of the game engine. It is written in Python 3.7 and

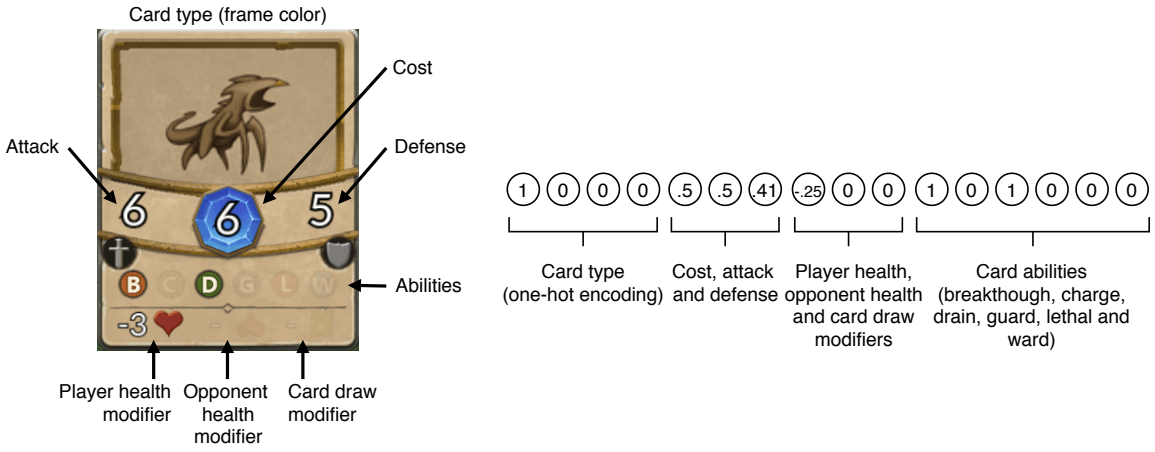
follows the OpenAI Gym [Brockman et al., 2016] interface, to increase compatibility with reinforcement learning algorithms. Gym environments for the draft phase, the battle phase and the full game are present, with one-player and two-player variations.

By not relying on inter-process communication to interact with agents, it can run up to 770 times faster than the original engine,[1] which is essential for DRL experiments that require simulation of many thousands of matches. We implemented some of the literature agents in our game engine, leaving out the ones with trickier implementation. However, our engine can also run agents developed for the original engine by using an adapter class, as well as any combination of their draft and battling strategies.

The engine, Gym environments, the implementation of our approaches, as well as the script used to train the models described in this chapter and the models themselves are available at GitHub.[2] We explain the engine and environments in more depth in Appendix A.

4.3.3 Feature extraction

Before feeding the game state to the networks, we convert it to a numeric vector. The game state in arena drafting is solely composed of cards – the ones already chosen (if using MDP_1) and the current alternatives (in both MDPs). We select the cards features that are relevant to determining their quality, and find a numeric representation for each of them. Figure 4.5 shows an example of a card and its respective numeric form.



(a) The card in the game. (b) The card as input to the neural networks.

Figure 4.5: Example of the conversion of a card in the game state to a numeric vector.

[1] Our engine took 0.62 second to execute 1,000 matches with a specific set of draft and battle agents, while the original engine took 480 seconds with the same set of agents.

[2] Available at `https://github.com/ronaldosvieira/gym-locm`

LOCM cards have fifteen attributes (as described in Table 4.1). Among these, we select the card type, cost, attack, defense, abilities and health and card draw modifiers. With the exception of the card type and abilities, all features are already numeric. We just divide these features by their maximum absolute value, to normalize them to the $[-1, 1]$ range. To convert the card type, a categorical feature, we apply one-hot encoding. In other words, the type feature is transformed into four mutually-exclusive binary features that determine whether the card is of one of the four types available in LOCM (creature or green, red or blue item). We then convert the resulting binary card type features as well as the six binary ability features (breakthrough, charge, drain, guard, lethal and ward) by considering true and false values as 1 and 0, respectively.

In summary, from the fifteen card attributes present in LOCM, we select thirteen (all except the card id and name) and transform into a numeric vector of sixteen elements. This is done on all cards in the game state, and the resulting vectors are concatenated to form the network's input.

4.4 Extending to other collectible card games

While we instantiate our methodology in LOCM, we try and anticipate the necessary steps to use it on other CCGs. Table 4.2 lists a selection of the currently most played CCGs that feature an arena-like mode, along with the mode name, the equivalent (k, n)-draft problem, and any extra challenges they require compared to LOCM. We address these extra challenges next:

1. **Card descriptions in natural language**. Unlike LOCM, many collectible card games present their card abilities in natural language rather than using predefined keywords. Therefore, a more complex feature extraction process than the one we use must be employed in this case. We suggest an approach such as the one proposed by Zuin and Veloso [2019], which uses LSTM layers in the network to learn a compact representation for *Magic: the Gathering* cards from their raw text description.

2. **Non-card choices**. Some collectible card games require additional choices during the draft that do not involve cards, such as choosing a deck class. These specific choices can be treated as a separate multi-armed bandit problem [Katehakis and Veinott Jr, 1987] and solved with an appropriate algorithm such as the upper confidence bound (UCB) [Lai and Robbins, 1985] using the win rate as the reward.

Game	Mode	Problem	Extra challenges
The Elder Scrolls: Legends	Arena	$(3, 30)$-draft	(1), (2)
Hearthstone	Arena	$(3, 30)$-draft	(1), (2)
Gwent	Arena[3]	$(4, 26)$-draft	(2)
Legends of Runeterra	Expedition	$(3, 14)$-draft	(1), (2), (3)
Magic: the Gathering	Booster draft	$(15, 45)$-draft	(1), (4), (5), (6)
Pokémon TCG	Booster draft	$(10, 60)$-draft	(1), (4), (5), (6)

Table 4.2: Arena-like game modes on the currently most played CCGs and their characteristics, ordered by similarity with LOCM.

3. **Choosing between sets of cards**. The problem formulation and the feature extraction process can be trivially extended to sets of cards instead of single cards.

4. **Varying** k. The booster draft mode decreases its k value every turn down to 1 and then resets it. A solution may be to assume the maximum k value and fill the missing card choices up to k with a null card as needed.

5. **Non-picked cards are reused**. In the booster draft mode, the $k-1$ non-picked cards in a draft turn are presented to another player in the next turn, which is a relevant factor when choosing. Our LSTM approach already leverages all past card alternatives in its reasoning, while the others can do so indirectly via the win rate.

6. **Drafted cards are not the final deck**. In the booster draft mode, the drafted cards do not represent the final deck, but rather a card pool from which the final deck will be built. A solution may be to pair with an appropriate deck-building agent to handle this additional step.

Thus, we believe our approach can be extended to other CCGs considering that the respective extra challenges are addressed. The solutions we provided for those challenges, however, are not exhaustive.

4.5 Summary

In this section, we proposed a methodology that uses deep reinforcement learning to tackle the problem of deck-building in the arena mode of collectible card games,

[3] *Gwent*'s arena mode was discontinued in October 2020, to be replaced with a new similar mode not yet released as of time of writing.

including its formulation as a Markov decision process and different ways of considering the cards chosen in the previous turns. We instantiated the methodology in *Legends of Code and Magic*, specifying its game-specific aspects, and discussed its use on other collectible card games.

Chapter 5

Experiments

In this chapter, we present the training setup (Section 5.1), the tuning of hyperparameters and network architecture (Section 5.2) and its results. Then, we conduct experiments that compare our approaches among themselves (Section 5.3), with other draft strategies in the literature (Section 5.4), and to other complete AI players (Section 5.5). Lastly, we discuss their results and implications (Section 5.6).

5.1 Training setup

We chose the Proximal Policy Optimization (PPO) algorithm with clipping (see Section 2.4) to train our drafters, after preliminary experiments with other DRL algorithms. We used the implementation available in the *stable-baselines* library[1] under the name of *PPO2*. PPO uses a neural network to parameterize the policy and an estimate of the value function, which gives the expected future rewards.

In LOCM, the drafted decks are used in only a single battle. There are significant differences between playing first and playing second in a battle (see Section 2.1), and current LOCM-playing approaches exploit this asymmetry by using a different drafting policy according to whether they are playing first or second.[2] Following this reasoning, we trained two separate neural networks specialized at, respectively, drafting for first and second players. Each network plays against an earlier version of the other for a determined number of episodes, from which we update both earlier versions with the respective newest version and repeat until the total amount of training episodes is

[1] `https://github.com/hill-a/stable-baselines`

[2] `https://www.codingame.com/forum/t/legends-of-code-magic-cc05-feedback-strategies/50996/63`

reached. This self-play variant is frequently used when the agents being trained are in asymmetric roles [Baker et al., 2020].

We trained our drafters partnered with two different battle agents. The first is *max-attack*, one of the baselines in previous LOCM-based AI competitions. It does all actions it can using the cards with the greatest attack power first. The second is a *greedy* agent [Kowalski and Miernik, 2020], which picks the best action according to a heuristic state evaluation over resulting states of a one-step lookahead.

Each training session consists of 30,000 episodes of the respective approach, namely Immediate, History or LSTM (see Section 4.2). Following the guidelines on reinforcement learning experiments from Colas et al. [2019], we evaluate the performance of our draft strategies in an offline manner twelve times during training. Each evaluation consists in playing 1,000 episodes and recording the battle agent's win rate. To set a common ground between all evaluations, the opponent always use a fixed draft strategy, called *max-attack*, which chooses the card with greatest attack power. The amount of episodes of training and evaluation were empirically determined, minding a balance of the training time and the quality of the resulting drafters.

All training sessions were conducted on machines with Intel Core i7-8700 3.2GHz processors, 16GB of RAM and NVIDIA GeForce GTX 1050 graphic cards with 4GB of VRAM. We used the version 3.7 of Python, alongside TensorFlow 1.14.0, CUDA 10.1 and the NVIDIA driver version 418.56, in the Ubuntu 18.04 operating system. We used 4-core CPU parallelism for draft and battle simulations and the GPU for network-related operations. On average, a training session lasted 30 minutes with the *max-attack* battler, and 3 hours with the *greedy* battler. Yet, the use of a trained drafter in a draft turn takes no more than one millisecond.

5.2 Hyperparameter tuning

To find the ideal network architecture and hyperparameters for PPO, we used the Tree of Parzen Estimators (TPE) optimization algorithm [Bergstra et al., 2011], through the *hyperopt* library [Bergstra et al., 2013]. We ran 50 iterations of the TPE algorithm for each of the six combinations between draft approaches (Immediate, History and LSTM) and battle agents (*max-attack* and *greedy*). In each iteration, a training session is conducted using a specific set of hyperparameters. The highest win rate obtained across the twelve checkpoints is returned to TPE, which selects the set of hyperparameters to be tried next, so as to reduce the uncertainty over promising regions of the hyperparameter space.

The hyperparameters tuned for the PPO algorithm were the learning rate (from 10^{-2} to 5×10^{-5}), the batch size (from 30 to 300 steps), how many mini-batches are formed from the batch (from 1 to 300) and for how many epochs these mini-batches are used to train (from 3 to 20 epochs), the clip range of the loss function (0.1, 0.2 or 0.3) as well as the coefficients of its value function (0.5 or 1) and entropy (from 0 to 10^{-2}) terms. Moreover, we also optimized the number of hidden layers in the networks (from one to three layers; in the LSTM approach, the first layer always use LSTM units), the number of neurons in these layers (from 24 to 256 neurons), the activation function (TanH, ReLU or ELU) and the frequency to update the opposing network's parameters in self-play (every 10, 100 or 1000 episodes).

We detected some patterns in the best set of hyperparameters returned by TPE. As a general trend, larger batch sizes (close to 300) and smaller learning rates (around 10^{-4}) were preferred. Most configurations ended up with shallow 1-layer networks, with the rest settling with 3 layers but less neurons on each layer.

A clearer pattern was that configurations using the *max-attack* battler had a total of 87 neurons in the network, in average, while the ones using *greedy* had a total of around 171 neurons. This may be due to the greater complexity and power of the latter battle agent requiring more elaborate decisions by the networks. In addition, the updates of the opponent network were more frequent in *max-attack* configurations (every 336 episodes, in average) than to *greedy* configurations (every 850 episodes, in average). This possibly means that more training episodes are required for the drafter to learn to defeat their opponent during self-play in the latter scenario. The full sets of hyperparameters found on each hyperparameter tuning we conducted are available at Attachment A.

5.3 Comparison between approaches

In our first experiment, we compared the performance of our three approaches: Immediate, History and LSTM (see Section 4.2). Ten training sessions with different random seeds were conducted for each combination of draft approach and battler, using their best network architecture and set of hyperparameters found.

The win rates obtained in each checkpoint were compiled into learning curves, where the drafter's performance and speed of convergence can be observed. Figure 5.1 shows the mean learning curves of each approach paired with each battle agent. The first and second player networks are compared separately and averaged at the right-hand side.

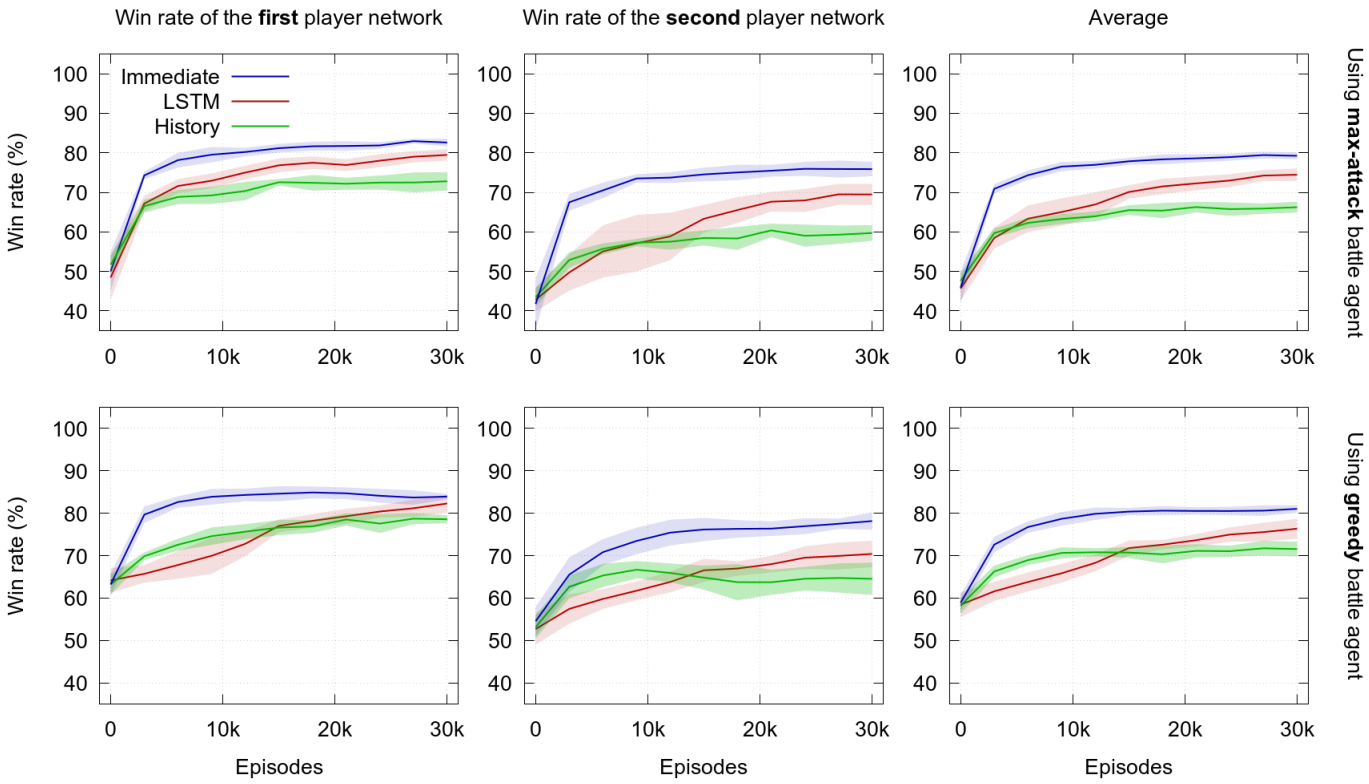


Figure 5.1: Learning curves of Immediate, History and LSTM drafters. Left and middle plots show the performance of the networks specialized at drafting for first and second players, respectively, whereas the right plot shows their average. Plots in the top row are trained and evaluated using the *max-attack* battle agent, while those in the bottom row use the *greedy* battle agent. Solid lines and shaded areas are the mean and standard deviation of the win rate, respectively.

The results show a better performance by the Immediate approach in all scenarios, followed by LSTM. Although History achieves better performance than LSTM early in the training, it also seem to settle earlier, while the latter continues to improve and eventually reaches better win rates. This suggests that, for a training session of 30,000 episodes, the simplicity of a much smaller state space outperforms the ability of History and LSTM to make decisions considering previously chosen cards. Another hypotheses is that the companion battle agents might not be able to leverage card synergies or that LOCM's cards and rules may be too simple to enable the emergence of relevant synergies or deck archetypes.

Figure 5.1 also shows the known advantage of playing first in LOCM, as first players invariably obtained higher win rates than second players. Furthermore, at the end of the training, most of the learning curves (especially those of LSTM) still display an increasing trend, meaning that better draft strategies could probably be achieved

by longer training sessions.

5.4 Comparison with other draft strategies

In our second experiment, we evaluated our resulting drafter against baseline and state-of-the-art draft strategies. The first and simplest one is a *random* draft. The second strategy is *max-attack*, which chooses the creature card with greatest attack power. The last three, named *coac*, *closet-ai* and *icebox*, are draft strategies from the best submissions of past LOCM-based competitions, which draft according to previously calculated card rankings. Since our goal is to find the best performing draft strategies, our agents were represented by the best policy obtained in each of their ten training sessions.

To compare our drafters with the selected ones, we observed several metrics in two separate round-robin tournaments containing all of them. On each of the two tournaments, every pair of drafters faced each other in twelve sets of 1,000 matches, where one of the drafters played as first player in half of the sets and then switched roles with the other drafter for the remaining half. To reduce variance, we used the same random seeds between pairwise match-ups, that is, in every i-th match between any pair of agents, the same set of choices were presented to both drafters and the same shuffling was applied to their finished deck. In the following subsections, we compare the drafters by win rate (Subsection 5.4.1), by mana curve (Subsection 5.4.2), and by similarity of choices (Subsection 5.4.3).

5.4.1 By win rate

We extracted the pairwise win rate (agent vs. agent) and aggregate win rate (agent vs. all other agents) from the round-robin tournaments, as depicted by Figure 5.2. The data show that our approaches outperform all tested draft agents using either battler. As in the previous experiment, the Immediate approach achieved the best win rates, followed by LSTM and History.

We performed paired t-tests to verify the significance of the difference in performance among our three approaches and the best performing competing drafter (*icebox* and *coac*, when using the *max-attack* and *greedy* battlers, respectively). We found all differences to be significant with $p < 0.001$, except the one between the performance of History and *coac* when partnered with the *greedy* battle agent, with a p-value of 0.253. In that case, we attested a significant difference between the performance of History and the second best performing competing drafter, *icebox*.

	random	*max-attack*	*coac*	*closet-ai*	*icebox*	History	LSTM	Immediate	**Average**
random	50	45.83	45.31	40.18	34	30.11	24.31	19.51	34.18
max-attack	54.17	50	51.24	43.54	37.7	33.3	27.08	20.28	38.19
coac	54.7	48.76	50	43.12	37.96	34.28	27.56	21.54	38.27
closet-ai	59.82	56.46	56.88	50	44.06	39.5	31.94	23.36	44.57
icebox	66	62.3	62.04	55.94	50	44.86	38.82	30.04	51.43
History	69.9	66.7	65.72	60.5	55.14	50	42.88	37.98	56.97
LSTM	75.7	72.92	72.44	68.06	61.18	57.12	50	43.82	64.46
Immediate	80.49	79.72	78.46	74.14	67.24	62.02	56.18	50	71.18

(a) Round-robin tournament using *max-attack* battler.

	max-attack	*random*	*closet-ai*	*icebox*	*coac*	History	LSTM	Immediate	**Average**
max-attack	50	41.67	38.08	35.86	28.84	27.78	25.76	18.84	30.98
random	58.34	50	45.87	45.39	36.73	36.51	34.5	28.06	40.77
closet-ai	61.92	54.14	50	49.48	41.02	39.82	37.64	28.66	44.67
icebox	64.14	54.61	50.52	50	43.34	40	39.04	30.16	45.97
coac	71.16	63.27	58.98	56.66	50	49.1	47.9	43.04	55.73
History	72.22	63.49	60.18	60	50.9	50	48.1	39.2	56.3
LSTM	74.24	65.51	62.36	60.96	52.1	51.9	50	42.62	58.53
Immediate	81.16	71.94	70.32	68.86	56.96	60.8	57.38	50	66.77

(b) Round-robin tournament using *greedy* battler.

Figure 5.2: Performance of our best drafters plus the current draft strategies in literature in two round-robin tournaments, each using a different battler. Cells represent the win rate (in %) of the row's drafter on 20 sets of a thousand matches against the column's drafter. The average column shows the average win rate of the draft agents against all opponents in the tournament.

Since most draft agents in the literature were created to work with a specific battle agent, an ideal experiment setting would involve training our approaches with each of these battlers and comparing individually. However, the computational cost of such task is prohibitive. It is important to mention that current deck-building approaches on the literature also have this issue, and resorted either on experiment settings similar to ours [Kowalski and Miernik, 2020] or very limited/no comparative experiments [García-Sánchez et al., 2016; Bhatt et al., 2018; Chen et al., 2018].

Nevertheless, when using the *max-attack* battler, our average trained networks can win up to 79.72% of the matches on average against the original *max-attack* drafter

and win more than 50% of the matches against all other drafters – evidence that our approaches can find better draft strategies.

5.4.2 By mana curve

Besides the win rate, other aspects show differences between our trained draft agents and the drafters in the literature. For each agent, we measured the histogram of mana costs of the cards in the average deck built by that agent in the tournament, that is, the average mana curve (see Section 2.1) of those decks. Figure 5.3 shows the average mana curves of each drafter, comparing with that of a random deck (which follows the distribution of mana costs in the set of all cards in the game), as a reference.

All of our trained agents displayed a tendency to pick more cards with mana costs of three or less, and, consequently, to pick less cards with mana costs of four or more, as shown by their curves being above and below the reference curve on the respective values of mana cost. This tendency was slightly stronger in the mana curves of those of our drafters which trained with the *greedy* battler than those which trained with the *max-attack* battler, as well as in the mana curves of Immediate drafters more than in those of LSTM and History drafters.

On the other hand, the other selected drafters showed less similarities among themselves: the *coac* and *closet-ai* drafters built decks with average mana curves close to the reference, while *max-attack* and *icebox* achieved very distinct distributions, tending towards choosing higher-cost cards. *Max-attack*'s mana curve can be explained by its draft strategy that explicitly favors cards with higher attack attributes, since this characteristic has a high correlation with having a high mana cost. Despite *coac*, *closet-ai* and *icebox*'s strategies being based on card rankings derived by analyzing large batches of match replays, their methodologies and the battle agents they observed in those matches differ, and, thus, they yielded different mana curves.

Some other points worth mentioning are: (i) our agents trained with the *greedy* battler were the only ones to show preference for zero-cost cards;[3] (ii) following the trend of favoring lower-cost cards, our agents drafted decks almost entirely composed of cards with six or less mana cost; and (iii) there were no significant differences between mana curves of first and second player agents of our approaches, going against a hypothesis created by human players which states that second players are favored by mana curves slightly biased into picking more low-cost cards, in comparison to first players.[4]

[3] Our agents trained with the *max-attack* battler had good reasons not to favor zero-cost cards: all of them are either items, which are entirely disregarded by the battler, or low-attack creatures, which are heavily disfavored by the battler.

[4] `https://www.codingame.com/forum/t/legends-of-code-magic-cc05-feedback-strategies/`

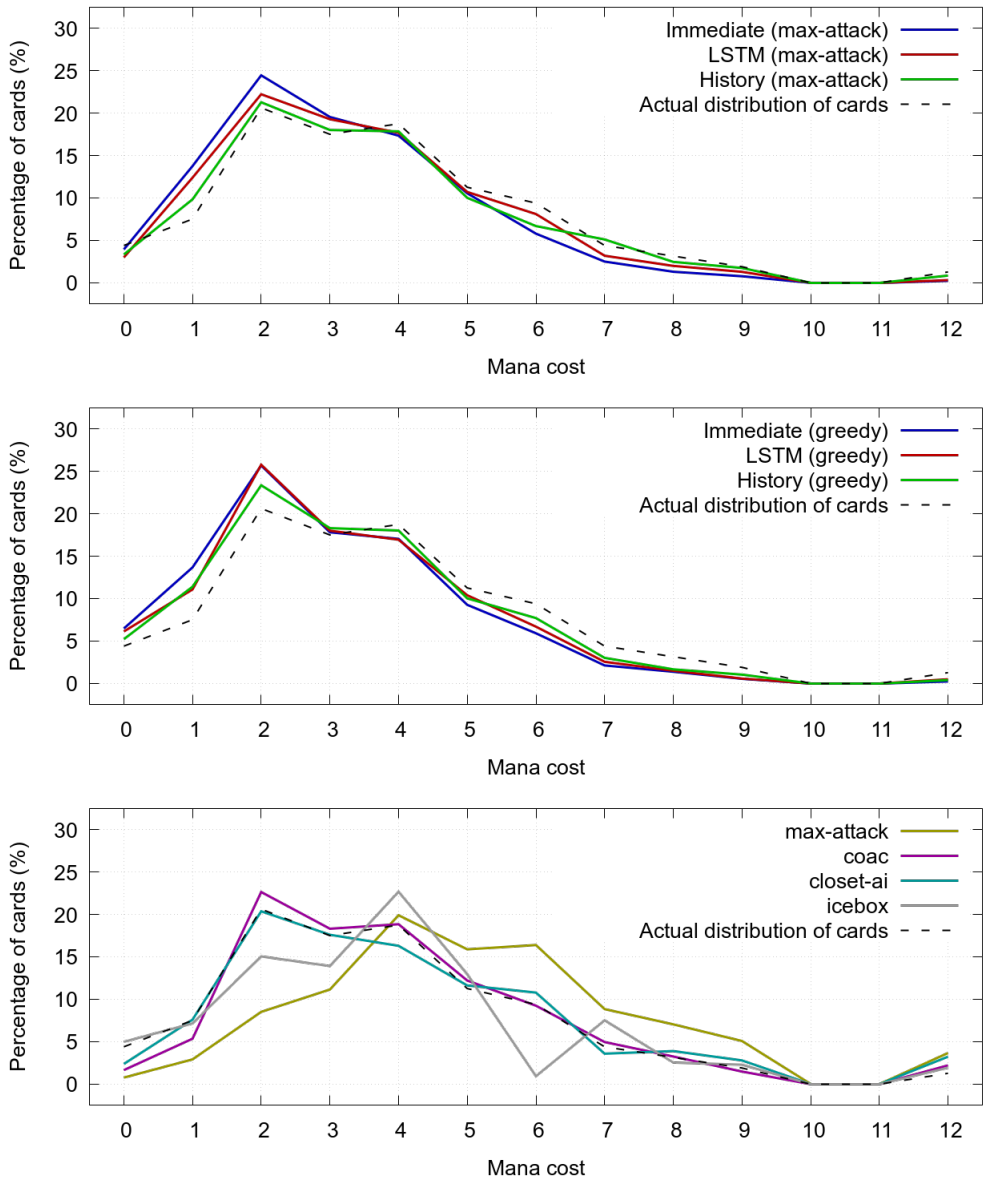


Figure 5.3: The average mana curves of each drafter in the tournaments. In the top, middle and bottom plots are, respectively, the average mana curves of our approaches trained with the *max-attack* battler, our approaches trained with the *greedy* battler, and the drafters in the literature. The actual distribution of mana costs of all cards the in the game, and, therefore, the average mana curve of a random deck, is represented by a dashed line in all plots. There are no cards with a mana cost of ten or eleven.

50996/63

	random	*max-attack*	*coac*	*closet-ai*	*icebox*	History/MA	LSTM/MA	Immediate/MA	History/GR	LSTM/GR	Immediate/GR
random											
max-attack	33.16										
coac	33.35	38.01									
closet-ai	33.38	44.57	41.17								
icebox	33.42	55.76	32.71	39.16							
History/MA	33.24	38.63	33.48	39.35	40.61						
LSTM/MA	33.21	39.6	31.74	40.36	43.08	47.65					
Immediate/MA	33.28	35.62	30.38	40.03	41.77	49.35	55.39				
History/GR	33.33	30.38	34.8	34.94	33.38	39.74	39.31	42			
LSTM/GR	33.33	28.99	35.48	36.2	33.15	39.22	44.24	45.93	41.25		
Immediate/GR	33.44	27.89	33.59	35.71	32.94	40.05	42.71	47.73	43.39	44.08	

Figure 5.4: Similarity of draft choices between agents. Cells indicate, in percentages, how frequently the row and column drafters chose the same card (out of three) in the 600,000 draft turns they played against each other in the tournament. Agents with the /MA and /GR suffixes were trained with the *max-attack* and *greedy* battlers, respectively. Symmetric values are omitted. The similarity of an agent with itself is of 100%.

5.4.3 By similarity of choices

We also analyzed the similarity between the 600,000 individual choices performed by each agent in the tournament.[5] For every pair of agents, we measured the percentage of choices in which the two of them agreed, that is, the ones in which they chose the same card when presented to the same card alternatives, as shown by Figure 5.4.

As expected, the choice similarity between the *random* drafter with every other drafter was around 33.34%, since it chooses randomly one of the three available cards. In general, the greatest percentages of similarity were found between our draft agents, especially with those trained with the same battle agent. In fact, the second greatest similarity value found, 55.39%, belong to the Immediate and LSTM drafters trained with the *max-attack* battler.

An unforeseen result was the similarity of 55.76% between the *icebox* and *max-attack* drafters – despite their dramatic difference in performance in both tournaments, they agreed in more than half of the choices. However, this resonates with their average mana curves, which were both shifted right (favoring higher-cost cards) compared to

[5]Each draft agent participated in a total of 20,000 unique matches, each of them containing 30 draft turns. This results in the same $20{,}000 \times 30 = 600{,}000$ unique draft turns presented to each draft agent, which required 600,000 individual choices.

those of the other drafters. Our drafters trained with *max-attack* were slightly similar to them while those trained with *greedy* were not. Also, *closet-ai* maintained some level of similarity to all drafters.

As a last effort to explore the tournament data, we applied Principal Component Analysis (PCA) [Jolliffe and Cadima, 2016] to reduce the choices vectors with 600,000 dimensions to three dimensions, to build a visual representation of how the drafters compare to each other. As all dimensions are categorical and not numeric as PCA expects, we converted the three possible values (first, second and third card) to angles that describe equidistant points in a circumference (30, 120 and 210 degrees). By using the sine and cosine of the respective angles, we ended up with 1,200,000 dimensions.

We also applied the K-means algorithm [Lloyd, 1982] to separate our resulting 3D points into clusters. Using silhouette analysis [Rousseeuw, 1987], we determined that the optimal value of k, that is, the optimal number of clusters, is equal to 5. Figure 5.5 shows the resulting 3D representation and clustering.

The three-dimensional representation of the drafters' choices during the tournament reiterated some of the observations made from the similarity table: the *random* is not similar to any agent and, thus, was placed far away from them. The *max-attack* drafter was closest to *icebox* both in choice similarity and in the representation. The high similarity between all of our drafters, which was even higher between those trained with the same battler, was also reflected in the plot. In fact, the clustering algorithm assigned two separate clusters to our drafters that trained with the same battler, while *icebox*, *max-attack* shared a third cluster, and *coac* and *closet-ai* shared a fourth one. Some runs of the clustering algorithm with different random seeds moved *closet-ai* to *icebox*'s cluster. Lastly, the similar positioning of our approaches relative to each other in the red and blue clusters is also worth noting.

Considering all results so far, there are many evidences that our trained draft agents build very different decks than the agents in the literature, and, in turn, obtain significantly better performance.

5.5 Agent improvement in the CoG 2019 LOCM tournament

As a comparative way to measure the performance of our best drafter, we re-executed the matches of the Strategy Card Game AI (SCGAI) competition held at IEEE Conference on Games (CoG) 2019, modifying the full *max-attack* agent, which was part of the tournament, to use our best draft strategy, namely, Immediate (see Section 4.2).

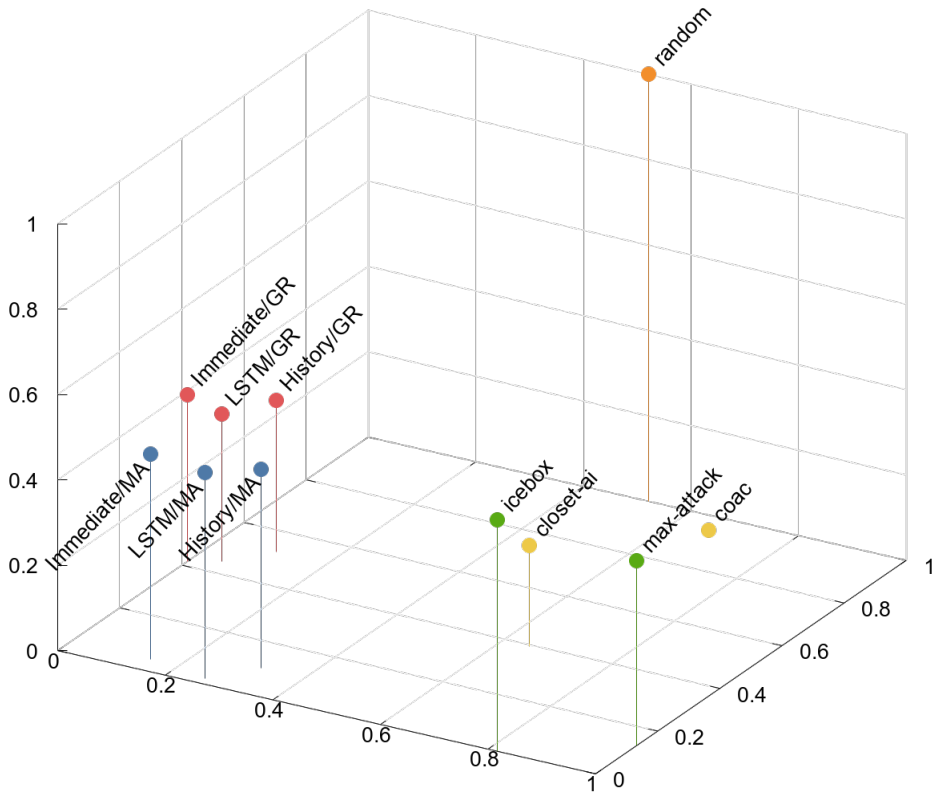


Figure 5.5: Three-dimensional representation of the choices of each drafter in the tournaments. The axes have no explicit meaning, only the distances between agents are informative. Drafters with same color belong to the same cluster. Agents with the /MA and /GR suffixes were trained with the *max-attack* and *greedy* battlers, respectively.

The competition gathers the state-of-the-art of LOCM-playing agents in a round-robin tournament, and is, therefore, an ideal scenario to evaluate our draft strategies.

The source-code of the agents and of the tournament itself are available on GitHub.[6] We used them to run the tournament twice (with the original and enhanced agent) with approximately 3000 matches between every combination of agents, yielding a total amount of approximately 350,000 matches per tournament. Figure 5.6 shows the agents' ranking and win rates in the original and modified competitions.[7]

With the aid of our best drafter (Immediate), the *max-attack* agent improved its

[6]See `https://legendsofcodeandmagic.com/COG19`

[7]The results from our reenacted competition are slightly different from the original ones from SCGAI 2019, probably due to hardware characteristics. The most notable difference is the drop in performance of the *Marasbot* agent, that achieved third place in the original competition.

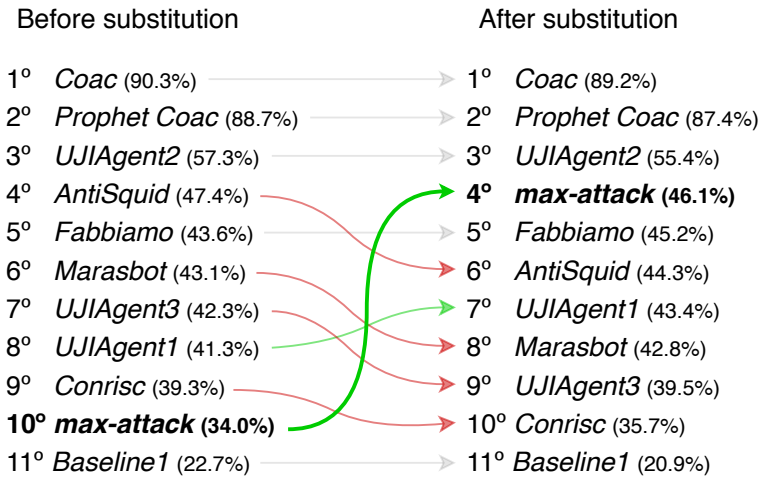


Figure 5.6: Tournament with agents from the IEEE CoG 2019 Strategy Card Game AI Competition before and after the substitution of the *max-attack* agent.

position from the tenth to the fourth place, winning 46.1% of the matches instead of the previous 34%. Although only trained with matches against a single opponent, our draft strategy was able to increase *max-attack*'s individual win rate against the majority of the opponents in the tournament. In some cases, however, the opponent agents performed better against the modified *max-attack* than against its original version, showing some degree of non-transitivity in the game – what works well against some opponents may be ineffective against others.

5.6 Summary

According to our experiments, our reinforcement learning drafting agents, when partnered with the same battle agents they were trained with, outperform all the other selected drafters (Figure 5.2) by building very different decks (Figures 5.3, 5.4 and 5.5). They also improved the ranking of the *max-attack* agent in the reenacted IEEE CoG 2019 LOCM tournament (Figure 5.6). Our drafting agents learn from their own choices, without domain knowledge or labeled data, and their decision process is fast, as it does not involve lookahead searches.

In our results, the performance of literature drafters varied significantly when paired to different battlers. Also, training with different battlers made our approaches yield drafters that chose significantly different (Figures 5.4 and 5.5). These points confirm that the tasks of deck-building and battling are influenced by each other. Moreover, our results also advocate for a positive response to our research question, that deep reinforcement learning methods can achieve competitive performance when

compared to current state-of-the-art drafting approaches.

A direct improvement on our methodology is to use a reward model that reflects the average win rate of the deck against a pool of diverse opponents, instead of just one. This might result in more robust drafters, since they will train in a more accurate representation of the scenario they will be used. On the other hand, the computational cost of each training episode would increase.

Our goal in this thesis is to generate draft strategies that are optimized for a single given playing strategy. However, if a more general drafter is intended, one could also use a reward model that accounts the wins of the deck in matches played by different battlers. This way, our methodology might generate more versatile drafters that would achieve good performance (yet not the best they could achieve) with many battlers, in exchange of an increased computational cost in training.

Another hypothesis on generating more flexible drafters is to explicitly account for different playing styles, using ad-hoc teamworking methods [Stone et al., 2010], where an agent must adapt to previously unseen partners. For example, the drafter could be coupled with different battlers during training, but in addition to the battle result, the agent would observe the execution of the matches. This would allow the drafter to infer the playing style of its partner using, for example, player modeling approaches [Davidson et al., 2000; Ganzfried and Sandholm, 2011; Machado et al., 2011; He et al., 2016]. The different playing styles of battlers could then be mapped to specific deck-building policies.

The surprising result of our experiments is that the Immediate agent, which disregards previously picked cards, achieved the best overall performance. As previously discussed, this suggests that a greater amount of training episodes may be needed for History and LSTM to exploit their ability of considering card synergies. It can also be explained by the fact that our drafters trained with simple battle partners, that may not be advanced enough to exploit combined card effects.

The more sophisticated battlers in the literature, that could leverage cards combinations, are based in tree-search techniques, which would slow training considerably. In a brief comparison, the *max-attack* battler can finish more than 5 whole matches by the time *coac*, the champion bot of past LOCM-based tournaments, finishes its reasoning for one typical turn in a match.

A possible solution would be to train a reinforcement learning battler as well, that acts based on a representation of the current state of the battle. With no searches involved, it would produce quick responses and enable fast training of draft strategies. However, training two interfering policies (draft and battle) simultaneously is not trivial: the drafter must learn to generate decks according to an ever-changing playing

style, while the battler must learn to play with decks coming from an unstable distribution of decks. This would probably worsen the known stability issues of current deep reinforcement learning algorithms. Fixing one while training the other, and then swapping them from time to time could be a suitable solution.

To reduce the amount of episodes required to train our approaches, it may be useful to conduct some form of dimensionality reduction technique on the card feature vectors such as principal component analysis [Jolliffe and Cadima, 2016], auto-encoder networks [Kramer, 1991] or node embedding (such as the one used by Zuin and Veloso [2019] for *Magic: the Gathering* cards). An effort on input simplification was to sort the card choices and previously picked cards in a state by an arbitrary criterion, to virtually reduce the space state, as all states containing permutations of the same cards would result in the same input passed to the network. However, this has not yielded better draft strategies as per our preliminary experiments, and was discarded in favor of treating all different permutations of the same cards as unique inputs, requiring the networks to be more robust.

Chapter 6

Conclusion

This chapter presents an overview of the work presented in this thesis (Section 6.1), along with a discussion on the contributions (Section 6.2), limitations (Section 6.3) and directions for future research (Section 6.4).

6.1 Overview

Despite the growth in the amount of work on games and the advances in game artificial intelligence (AI) in the last decade, collectible card games (CCGs) are not yet played in superhuman level. They have a vast set of intricate, dynamic rules and their mastery requires solving two interdependent problems: deck building and battling. This thesis tackles the problem of deck building in the arena mode of collectible card games, where players draft their deck one card of a time from a set of randomly presented candidates. We modeled the problem as a Markov Decision Process and presented three deep reinforcement learning (DRL) approaches that vary on how to deal with the information of previous card choices when choosing a new card. We measured the performance of our drafters by observing the win rate of the decks they built in matches played by simple battle agents.

We use *Legends of Code and Magic*, a CCG designed for AI research, as a testbed to evaluate our methodology. Our draft strategies, trained in self-play, outperformed all competing drafters, including the ones used by top bots in LOCM-based AI competitions, when coupled with two different battle agents. Moreover, we showed that a participant of one of these competitions would have achieved the fourth instead of the tenth place if using our best drafter. These are evidences in favor of a positive response to our research question, that deep reinforcement learning methods can achieve competitive performance in comparison to the current state-of-the-art drafting approaches.

The discovery of strong draft policies is a step towards competent AI for CCGs. Once trained, approaches based on reinforcement learning are fast, as they do not need to perform computationally expensive operations such as lookahead searches. The benefits of fast and strong CCG-playing AIs result not only in more challenging opponents for humans, but also contribute to game balancing by allowing efficient playtest of new rules or cards.

6.2 Contributions

The main contributions of our work are:

- **A game-agnostic methodology for developing fast, competent draft strategies**. Our methodology reduces the problem of drafting in arena mode of CCGs to a Markov Decision Process, and tackle it with a deep reinforcement learning algorithm. It does not require powerful hardware or labeled data to train, and takes no more than one millisecond to run. Besides, unlike previous drafting approaches, our drafters observe card features, thus not being tightly tied to the specific card set seen on training.

- **A collection of competitive draft strategies for *Legends of Code and Magic*.** We provide the final network parameters of our best draft strategies from the Immediate, History and LSTM approaches, as ZIP files (importable by the *stable-baselines* package) and JSON files.[1]

- **Software contributions**. We implemented the rules of *Legends of Code and Magic* and created OpenAI Gym environments for reinforcement learning algorithms to play the game's draft and battle phases, following our Markov Decision Process formulations and some additional variations. We also developed a Python script to use any of our trained draft strategies in existing LOCM bots, which overrides their original draft strategies with ours. The aforementioned implementations are detailed in Appendix A and are available at GitHub.[2] Moreover, we submitted a bot to the Strategy Card Game AI competition, which consisted of our best drafter and a best-first search battler.[3]

[1] `https://github.com/ronaldosvieira/gym-locm/tree/1.0.0/gym_locm/trained_models`
[2] `https://github.com/ronaldosvieira/gym-locm`
[3] `https://github.com/ronaldosvieira/reinforced-greediness`

6.3 Limitations

The main limitations of our work are:

- **A large amount of matches is required to train**. Each of our draft strategies for LOCM required the execution of 42,000 episodes, divided between training and evaluation. Combining all training sessions performed for the experiments in this thesis, a total of 5,880,000 episodes were run. Although the current deck building literature also require a large amount of matches to develop competitive solutions as performance metrics for decks are of statistical nature, this number reflects the low sample-efficiency of the Proximal Policy Optimization algorithm and of deep reinforcement learning methods in general.
- **It considers a single opponent during training**. As discussed in Section 5.6, our drafters train by playing matches against a single opponent (drafter and battler). While it was enough to produce draft agents which performed better than the ones in literature, considering a diverse set of opponents instead may result in more robust drafters.

6.4 Directions for future research

This thesis is not a definitive work in deck building in CCGs with reinforcement learning. Many extensions of our approaches as well as different ones are possible, and worth investigating. Some directions for future research are:

- **Investigate the better performance of the history-oblivious approach**. Despite our initial expectations, considering the history of choices so far when choosing a new card has not yielded better draft agents as per our experiments. Our leading hypothesis is that more training episodes may be needed for such approaches to be able to perform well enough in the much more difficult problem they tackle. An alternative hypothesis is that LOCM's cards and rules may be too simple for any relevant synergies between cards to exist. This could be tested by altering the game to force at least one strong synergy among cards, and observing whether the history-aware approaches finally achieve better performance than Immediate.
- **Extract insights from the trained networks**. Our trained draft strategies consist of neural networks that learned how to choose cards to maximize the

win rate of a battler, given the features of the available cards. By inspecting the networks' parameters, one could obtain insights about the decision process, including which card features are important and affect positively or negatively the choice.

- **Consider diverse opponents during training**. Our resulting drafters could benefit of being trained facing more diverse opponents. This would avoid results such as those of our third experiment (Section 5.5), where the use of our draft strategy, which trained facing a single opponent, yielded a drop in performance against two of the ten adversaries, although yielding a significant performance gain against the others. As proposed in Section 5.6, the reward model could be modified to consider a set of different opponents.

- **Improve sample-efficiency**. Our reward model is sparse and noisy – we give a single reward signal for all the choices in an episode, which is based on the performance of the resulting deck as a whole in a few amount of battles. Therefore, a large amount of episodes is needed to obtain a reliable feedback on a specific card choice.

  To reduce noise, chosen cards that were not drawn during any of the evaluation battles could have their respective card choices removed from the training batch, as the resulting reward was not influenced by them. To reduce sparseness, the incomplete deck obtained after each card choice, composed of the cards drafted so far, could be used in battles against other incomplete decks of same size, enabling non-null rewards across the episode. In this case, a sensible minimum deck size should be observed, since battles with very small decks may not yield representative rewards.

- **Use the methodology on commercial CCGs**. With its simpler rules, *Legends of Code and Magic* served as a good testbed and a first step towards competent drafting approaches for CCGs. A second step would be to use and validate our methodology on more complex CCGs. *Hearthstone* would be a good choice due to its popularity among players and game AI researchers, as well as the abundance of implementations of its rules.

- **Try the Curve approach.** Part of the unpublished drafting approaches for LOCM found it useful to draft cards as to fit a pre-calculated ideal mana curve, that is, they aim to pick a specific amount of cards with each mana cost so that

the use of mana in each battle turn is optimized.[4] This idea is central to deck building in other CCGs [Karsten, 2014]. The Curve approach would augment Immediate (see Section 4.2) by adding the amount of cards with each mana cost that were already picked to the state representation. LSTM and History, approaches that keep track of the history of picks, are able to learn to consider the mana curve, since the mana cost of every card drafted so far is available. Instead of letting that knowledge be learned by the network on its own, Curve would add a shortcut by explicitly giving the current mana curve in the input.

- **Try different values of k and n.** Although $(3, 30)$-drafts seem to be the most common configuration in the arena modes of commercial collectible card games, other values of k and n do exist (as discussed in Section 4.4) and some others may be proposed in the future. An analysis of the difficulty, the entropy and optimality of draft strategies and AI techniques as k and n change (e.g., a draft with $k = |\mathcal{C}|$ approximates deck-building in a constructed mode) could produce useful insights about CCGs and about deck-building in arena mode.

- **Develop a deep reinforcement learning battler**. As discussed in Section 5.6, deep reinforcement learning battlers could be fast enough to be used on our methodology. A strong DRL battle agent could improve the quality of learned draft strategies by leveraging more complex combinations of card effects and, therefore, reflecting those on the resulting rewards. It would also be a promising product on its own, especially for CCG AI competitions.

- **Find low-dimensional card representations**. In LOCM, our numeric card representation contained sixteen dimensions (see Section 4.3.3). The six binary ability dimensions, for instance, are sparse – the vast majority of the cards contain up to one ability. In commercial CCGs, which feature tens of different abilities, this would be especially true. Dimensionality reduction techniques such as principal component analysis [Jolliffe and Cadima, 2016], auto-encoder networks [Kramer, 1991] or node embeddings [Zuin and Veloso, 2019] may find robust low-dimensional representation for cards, which could make training easier. Our initial efforts found good four- and six-dimensional card representations for LOCM.

- **Explore the hidden information**. Usually, in CCGs, the player does not know which cards are in their opponent's deck and hand or in which ordering

[4] https://www.codingame.com/forum/t/legends-of-code-magic-cc05-feedback-strategies/50996/63

their own deck is. Although this hidden information takes place in the battles, the available solutions to tackle it are dependent on the deck building format. In constructed game modes, prediction of the opponent's deck is usually done by identifying most probable co-occurrence of cards considering the cards already played by the opponent [Bursztein, 2016], as the metagame is usually formed by dominant deck archetypes that share the same cards.

In the arena-like deck building format present in LOCM, since the two players in battles have drafted their decks from the same sets of card choices, predicting which cards are in the opponent's deck is easier. This could be tackled by keeping track of all cards presented in the draft phase and building a probabilistic model of the opponent's hand and deck which is updated as they play and draw cards. This model would enable, for instance, more accurate simulation of future opponent turns in battlers based on tree search methods, as well as a richer state representation in deep reinforcement learning battlers.

Our initial efforts in predicting the opponent's hand for a tree-search LOCM battler[5] have yielded a decrease in performance in comparison to not trying to predict. According to our experiments, the additional variance introduced by the many new possibilities of future opponent moves weighted more than the benefit of estimating them.

[5] `https://github.com/ronaldosvieira/prophet-coac`

Bibliography

Adam, S., Busoniu, L., and Babuska, R. (2012). Experience replay for real-time reinforcement learning control. *IEEE Trans. Syst. Man Cybern. Part C*, 42(2):201--212.

Baker, B., Kanitscheider, I., Markov, T. M., Wu, Y., Powell, G., McGrew, B., and Mordatch, I. (2020). Emergent tool use from multi-agent autocurricula. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546--2554.

Bergstra, J., Yamins, D., and Cox, D. D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML 2013*. JMLR.org.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680.

Bhatt, A., Lee, S., de Mesentier Silva, F., Watson, C. W., Togelius, J., and Hoover, A. K. (2018). Exploring the hearthstone deck space. In Dahlskog, S., Deterding, S., Font, J. M., Khandaker, M., Olsson, C. M., Risi, S., and Salge, C., editors, *Proceedings of the 13th International Conference on the Foundations of Digital Games, FDG 2018, Malmö, Sweden, August 07-10, 2018*, pages 18:1--18:10. ACM.

Bjørke, S. J. and Fludal, K. A. (2017). Deckbuilding in Magic: the Gathering using a genetic algorithm. Master's thesis, NTNU.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540.

Brown, N. and Sandholm, T. (2019). Superhuman AI for multiplayer Poker. *Science*. ISSN 0036-8075.

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Liebana, D. P., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1--43.

Bursztein, E. (2016). I am a legend: Hacking hearthstone using statistical learning methods. In *IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016*, pages 1--8. IEEE.

Chen, Z., Amato, C., Nguyen, T. D., Cooper, S., Sun, Y., and El-Nasr, M. S. (2018). Q-deckrec: A fast deck recommendation system for collectible card games. In *2018 IEEE Conference on Computational Intelligence and Games, CIG 2018, Maastricht, The Netherlands, August 14-17, 2018*, pages 1--8. IEEE.

Colas, C., Sigaud, O., and Oudeyer, P. (2019). A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms. In *Reproducibility in Machine Learning, ICLR 2019 Workshop, New Orleans, Louisiana, United States, May 6, 2019*. OpenReview.net.

Cowling, P. I., Ward, C. D., and Powley, E. J. (2012). Ensemble determinization in Monte Carlo tree search for the imperfect information card game Magic: The Gathering. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(4):241--257.

Cunningham, J. (2007a). Introduction to efficiency. `https://magic.wizards.com/en/articles/archive/magic-academy/introduction-efficiency-2007-07-07`. Accessed: 2020-06-29.

Cunningham, J. (2007b). What is the metagame? `https://magic.wizards.com/en/articles/archive/magic-academy/what-metagame-2007-01-06`. Accessed: 2020-06-29.

Davidson, A., Billings, D., Schaeffer, J., and Szafron, D. (2000). Improved opponent modeling in poker. In *International Conference on Artificial Intelligence, ICAI'00*, pages 1467--1473.

De Mesentier Silva, F., Canaan, R., Lee, S., Fontaine, M. C., Togelius, J., and Hoover, A. K. (2019). Evolving the hearthstone meta. In *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*, pages 1--8. IEEE.

Dockhorn, A., Frick, M., Akkaya, Ü., and Kruse, R. (2018). Predicting opponent moves for improving hearthstone AI. In Medina, J., Ojeda-Aciego, M., Galdeano, J. L. V., Pelta, D. A., Cabrera, I. P., Bouchon-Meunier, B., and Yager, R. R., editors, *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations - 17th International Conference, IPMU 2018, Cádiz, Spain, June 11-15, 2018, Proceedings, Part II*, volume 854 of *Communications in Computer and Information Science*, pages 621--632. Springer.

DoubleXP (2014). Consistency in Hearthstone: The why and how | DoubleXP. `https://doublexp.com/courses/training/lesson/60`. Accessed: 2020-06-29.

Fontaine, M. C., Lee, S., Soros, L. B., de Mesentier Silva, F., Togelius, J., and Hoover, A. K. (2019). Mapping hearthstone deck spaces through map-elites with sliding boundaries. In Auger, A. and Stützle, T., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 161--169. ACM.

Fukushima, K. and Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267--285. Springer.

Ganzfried, S. and Sandholm, T. (2011). Game theory-based opponent modeling in large imperfect-information games. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 533--540. International Foundation for Autonomous Agents and Multiagent Systems.

García-Sánchez, P., Tonda, A. P., García, A. M., Squillero, G., and Merelo Guervós, J. J. (2018). Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone. *Knowl. Based Syst.*, 153:133--146.

García-Sánchez, P., Tonda, A. P., Squillero, G., García, A. M., and Merelo Guerv'os, J. J. (2016). Evolutionary deckbuilding in hearthstone. In *IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016*, pages 1--8. IEEE.

Góes, L. F. W., Da Silva, A. R., Saffran, J., Amorim, A., França, C., Zaidan, T., Olímpio, B. M., Alves, L. R., Morais, H., Luana, S., et al. (2016). Honingstone: Building

creative combos with honing theory for a digital card game. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(2):204--209.

He, H., Boyd-Graber, J., Kwok, K., and Daumé III, H. (2016). Opponent modeling in deep reinforcement learning. In *International Conference on Machine Learning*, pages 1804--1813.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In McIlraith, S. A. and Weinberger, K. Q., editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3215--3222. AAAI Press.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735--1780.

Hoover, A. K., Togelius, J., Lee, S., and de Mesentier Silva, F. (2020). The many AI challenges of hearthstone. *KI*, 34(1):33--43.

Jolliffe, I. T. and Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202.

Karsten, F. (2014). Frank analysis - finding the optimal mana curve via computer simulation. `https://www.channelfireball.com/all-strategy/articles/frank-analysis-finding-the-optimal-mana-curve-via-computer-simulation`. Accessed: 2020-06-24.

Katehakis, M. N. and Veinott Jr, A. F. (1987). The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262--268.

Khalifa, A., Bontrager, P., Earle, S., and Togelius, J. (2020). PCGRL: procedural content generation via reinforcement learning. *CoRR*, abs/2001.09212.

Koska, A. (2010). The consistency principles. `https://magic.tcgplayer.com/db/article.asp?ID=8905`. Accessed: 2020-06-29.

Kowalski, J. and Miernik, R. (2018). Legends of Code and Magic. `https://legendsofcodeandmagic.com`. Accessed: 2020-03-27.

Kowalski, J. and Miernik, R. (2020). Evolutionary Approach to Collectible Card Game Arena Deckbuilding using Active Genes. *arXiv e-prints*, page arXiv:2001.01326.

Kramer, M. A. (1991). Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233--243.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 22(1):79--86.

Lai, T. L. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4--22.

Le, V. (2019). Coac/locm. `https://github.com/coac/locm`. Accessed: 2019-06-14.

Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 28(2):129--136.

López, C. E., Cunningham, J., Ashour, O., and Tucker, C. S. (2020). Deep reinforcement learning for procedural content generation of 3d virtual environments. *Journal of Computing and Information Science in Engineering*, 20(5).

Machado, M. C., Fantini, E. P. C., and Chaimowicz, L. (2011). Player modeling: Towards a common taxonomy. In Mehdi, Q. H., Elmaghraby, A. S., Moreton, R., Yampolskiy, R. V., and Chariker, J., editors, *16th International Conference on Computer Games, CGAMES 2011, Louisville, KY, USA, 27-30 July, 2011*, pages 50--57. IEEE Computer Society.

Metz, C. (2016). The Sadness and Beauty of Watching Google's AI Play Go. `https://www.wired.com/2016/03/sadness-beauty-watching-googles-ai-play-go`. Accessed: 2020-04-03.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529--533.

Mouret, J. and Clune, J. (2015). Illuminating search spaces by mapping elites. *CoRR*, abs/1504.04909.

Pal, S. and Mitra, S. (1992). Multilayer perceptron, fuzzy sets, and classification. *IEEE transactions on neural networks*, 3(5):683.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53--65.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Sammut, C. and Webb, G. I., editors (2010). *Mean Squared Error*, pages 653--653. Springer US, Boston, MA.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In Bengio, Y. and LeCun, Y., editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. (2015). Trust region policy optimization. In Bach, F. R. and Blei, D. M., editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889--1897. JMLR.org.

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

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354--359.

Stone, P., Kaminka, G. A., Kraus, S., and Rosenschein, J. S. (2010). Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Swiechowski, M., Tajmajer, T., and Janusz, A. (2018). Improving Hearthstone AI by combining MCTS and supervised learning algorithms. In *CIG*, pages 1--8.

Thorndike, E. L. (1911). Animal intelligence. *Psych Revmonog*, 8(2):207--208.

van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In Schuurmans, D. and Wellman, M. P., editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2094--2100. AAAI Press.

Vieira, R., Chaimowicz, L., and Tavares, A. R. (2019). Reinforcement learning in collectible card games: Preliminary results on Legends of Code and Magic. In *18th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2019, Rio de Janeiro, Brazil, October 28-31, 2019*. IEEE.

Vieira, R., Tavares, A. R., and Chaimowicz, L. (2020). Drafting in collectible card games via reinforcement learning. In *19th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2020, Recife, Brazil, November 7-10, 2020*, pages 54--61. IEEE.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350--354.

Wang, D. and Moh, T. (2019). Hearthstone AI: oops to well played. In *ACMSE*, pages 149--154.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In Balcan, M. and Weinberger, K. Q., editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1995--2003. JMLR.org.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD dissertation, University of Cambridge.

Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. Springer. ISBN 978-3-319-63518-7.

Zuin, G. L. and Veloso, A. (2019). Learning a resource scale for collectible card games. In *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*, pages 1--8. IEEE.

Appendix A

The `gym-locm` repository

This appendix describes the contents of the `gym-locm` repository on Github, which contains the source code for (i) our implementation of *Legends of Code and Magic*'s (LOCM) rules, (ii) the OpenAI Gym environments describing the draft, the battle and the full game in one and two-player variants, (iii) the reproduction-ready experiment scripts, and (iv) the final draft agents generated by our methodology. The repository can be accessed at `https://github.com/ronaldosvieira/gym-locm`.

A.1 The game engine

Legends of Code and Magic, or LOCM, was created in 2018 by Kowalski and Miernik, as a simpler alternative to the current commercial collectible card games meant to facilitate AI research on the topic.[1] To do so, LOCM features a small subset of the rules of other CCGs, and provides a very easy way to integrate with AI agents developed in any programming language. However, this easiness and language-agnosticism takes a toll on performance. To instantiate our methodology, which required the execution of millions of matches, we decided that it was worth it to reimplement the game optimizing for performance at the expense of agent compatibility. We chose Python, as it is currently the most used programming language for reinforcement learning and AI in general.

The implementation revolves around the `State` class (`gym_locm.engine.State`), which represents a state in the game. When a new `State` is instantiated, it is automatically configured to represent the first turn of the draft phase of a LOCM match. The state can then be advanced by calling its `act` method, passing an `Action` object (which represents a player's action in the game). Passing a PICK action in the draft phase will

[1] More information on the game as well as the original game engine is available at `https://legendsofcodeandmagic.com`.

pick a card and pass the turn, and the state will then represent the opposing player's turn. When all players pick their last card in the draft, the state automatically is set to represent the first turn of the battle phase. Then, the `act` method can be used to perform SUMMON, USE and ATTACK actions (the `State.available_actions` attribute always holds a list with all actions that are valid in the current state). However, the turn is not passed until a PASS action in fed to `act`.

Moreover, a `State` can be created from a textual game state representation (as used by the original LOCM engine) via the `State.from_native_input` method. This enables agents developed for our engine to be quickly adapted to play on the original engine. Figure A.1 shows the class diagram for the `gym_locm.engine` module, including all aforementioned classes and methods.

A.1.1 The available agents

We provide a collection of draft agents and of battle agents to use along with our LOCM engine. They all follow the `Agent` interface (`gym_locm.agents.Agent`), which requires implementation of three methods: (i) `seed`, which sets a specific random seed for any stochastic processing within the agent; (ii) `reset`, which resets the internal state of the agent; and (iii) `act`, which takes a `State` as argument and returns an `Action` as per the agent's strategy. Tables A.1 and A.2 describe, respectively, the draft and battle agents available, as well as if they are stochastic (i.e., if there is any randomness involved in its reasoning) or stateful (i.e., if they keep any information from previous rounds).

We also provide fully-fledged compatibility with agents developed for the original LOCM engine via the `NativeAgent` class and its two subclasses `NativeDraftAgent` and `NativeBattleAgent`. Their use differs from the other available agents only by the presence of the `cmd` parameter on its constructor, which should receive a string containing the terminal command necessary to execute the native agent (i.e., `python3 agent.py`).

A.2 The OpenAI Gym environments

OpenAI Gym is an open-source interface to reinforcement learning tasks which has become the most popular way to program tasks involving Markov decision processes in the recent years [Brockman et al., 2016]. A reinforcement learning task that is programmed following the Gym interface is called a Gym environment, and have two main methods: `reset`, which resets the problem to a initial state; and `step`, which

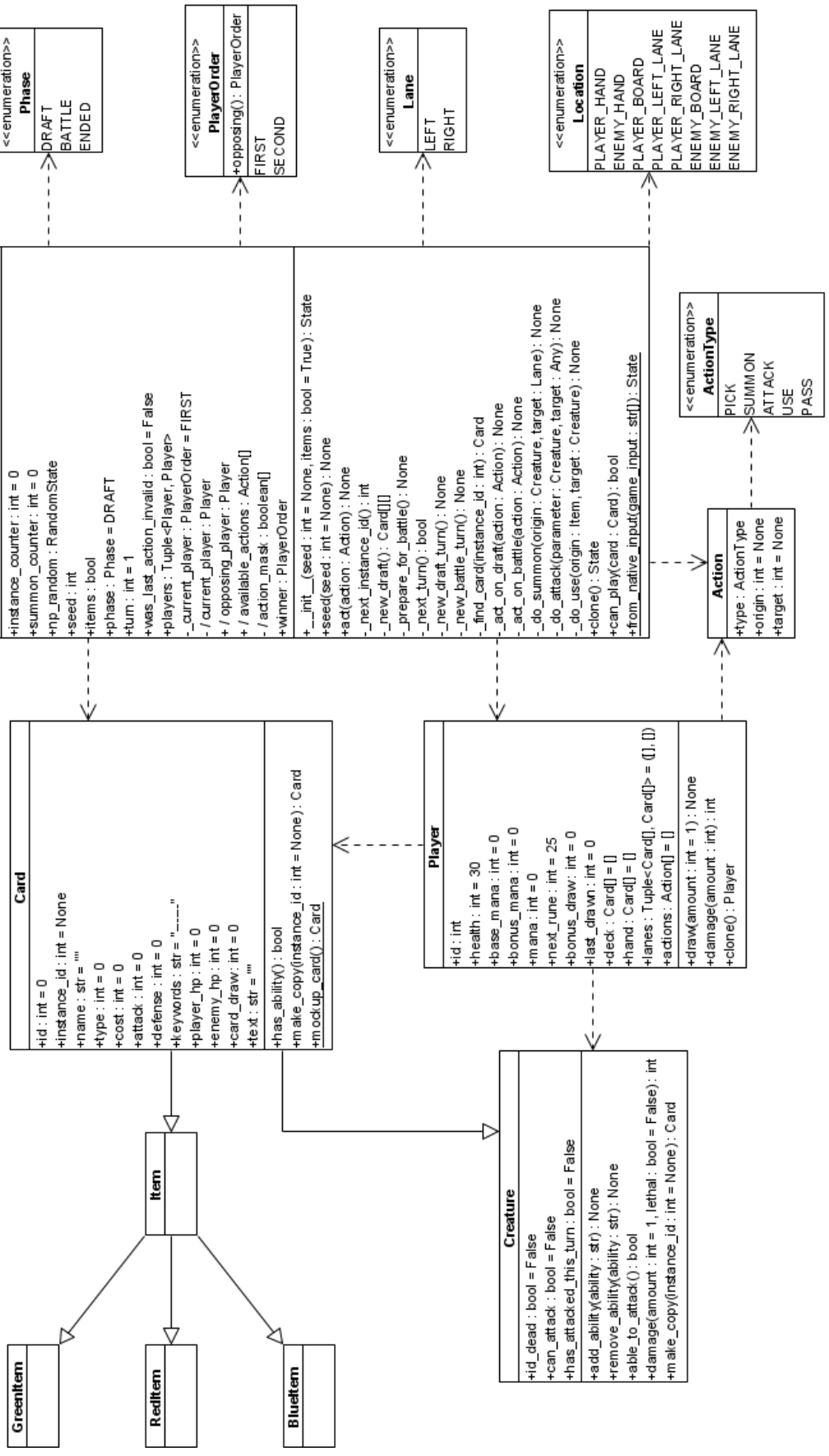


Figure A.1: Class diagram of the `gym_locm.engine` module.

Class name	Stochastic	Stateful	Strategy
`PassDraftAgent`	No	No	Always passes the turn (this is equivalent to always choosing the first card).
`RandomDraftAgent`	Yes	No	Drafts at random.
`RuleBasedDraftAgent`	No	No	Same as *Baseline1* from the SCGAI competition.
`MaxAttackDraftAgent`	No	No	Same as *Baseline2* from the SCGAI competition.
`IceboxDraftAgent`	No	No	Same as used by CodinGame's user *Icebox*.
`ClosetAIDraftAgent`	No	No	Same as used by CodinGame's user *ClosetAI*.
`UJI1DraftAgent`	No	Yes	Same as *UJIAgent1* from the SCGAI competition.
`UJI2DraftAgent`	No	Yes	Same as *UJIAgent2* from the SCGAI competition.
`CoacDraftAgent`	No	Yes	Same as *Coac* from the SCGAI competition.
`NativeDraftAgent`	(1)	Yes	Wrapper for native agents developed for the original LOCM engine.

Table A.1: Draft agents available in our source-code. (1) Depends on the wrapped agent.

Class name	Stochastic	Stateful	Strategy
`PassBattleAgent`	No	No	Always passes the turn.
`RandomBattleAgent`	Yes	No	Chooses a valid action at random (including passing the turn).
`RuleBasedBattleAgent`	No	No	Same as *Baseline1* from the SCGAI competition.
`MaxAttackBattleAgent`	No	No	Same as *Baseline2* from the SCGAI competition.
`GreedyBattleAgent`	No	No	Same as *greedy* from [Kowalski and Miernik, 2020].
`MCTSBattleAgent`	(2)	No	Uses a Monte Carlo tree search algorithm (experimental).
`NativeBattleAgent`	(1)	Yes	Wrapper for native agents developed for the original LOCM engine.

Table A.2: Battle agents available in our source-code. (1) Depends on the wrapped agent. (2) Depends on the agent used as the MCTS's default policy.

takes an action as argument, advances the problem state, and returns the outcome (a reward signal, the new state and some additional information).

To facilitate further research on the game (due to the popularity of the Gym interface and "plug-and-play" algorithms developed for them), we developed Gym environments to represent the reinforcement learning problems of drafting and battling in a LOCM match, with the agent acting for one or both of the players. They are available, respectively, via the `LOCMDraftSingleEnv`, `LOCMDraftEnv`, `LOCMBattleSingleEnv` and `LOCMBattleEnv` classes in the `gym_locm.envs` module, and wrap over a `State` object.

Using an Gym environment is easy: Listing A.1 shows the necessary source-code to run a LOCM match with the single-player draft environment and an agent that drafts randomly. After importing the necessary libraries, it builds the appropriate environment using `gym.make`,[2] and initializes a random draft agent. Then, it obtains

[2]The equivalent arguments to build the other mentioned environments as well as other possible configurations are available at `https://github.com/ronaldosvieira/gym-locm`.

the initial state of the environment by using `env.reset` and initializes a control variable (`done`) to detect the end of the episode. While the episode has not finished, it queries the agent for an action, given the current state, and uses that action to advance the environment to the next state. After the end of the episode, it prints which player ended victorious in the LOCM match.

```
import gym
import gym_locm

env = gym.make('LOCM-draft-v0')
agent = gym_locm.agents.RandomDraftAgent()

state = env.reset()
done = False

while not done:
    action = agent.act(state)
    state, reward, done, info = env.step(action)

print("Winner:", info['winner'])
```

Listing A.1: Executing a LOCM match with the single-player draft environment.

A.3 Reproducing our experiments

All of the experiments we conducted and described on Chapter 5 can be reproduced with the `gym-locm` repository. For each one of them, we provide scripts with many tunable parameters as well as instructions to run them and the original parameters we used. All experiment scripts are located in the `gym_locm.experiments` module.

A.3.1 Hyperparameter tuning

To perform a hyperparameter tuning, simply execute the `hyp-search.py` script:

```
python3 gym_locm/experiments/hyp-search.py --approach <approach> \
--battle-agent <battle_agent> --path hyp_search_results/ \
--seed 96765 --processes 4
```

The list and range of hyperparameters explored is available in the Attachment A. we performed a separate hyperparameter tuning for every combination of `<approach>`

(immediate, history and lstm) and <battle_agent> (max-attack and greedy). Each execution of the script took around 2 days with the *max-attack* battle agent and more than a week with the *greedy* battle agent. To learn about other script's parameters, execute it with the --help flag.

A.3.2 Comparison between approaches

To train two draft agents (a first player and a second player) with a specific draft approach and battle agent, in asymmetric self-play, simply execute the training.py script:

```
python3 gym_locm/experiments/training.py --approach <approach> \
--battle-agent <battle_agent> --path training_results/ \
--n-switches <n_switches> --layers <layers> --neurons <neurons> \
--act-fun <activation_function> --n-steps <batch_size> \
--nminibatches <n_minibatches> --noptepochs <n_epochs> \
--cliprange <cliprange> --vf-coef <vf_coef> --ent-coef <ent_coef> \
--learning-rate <learning_rate> --seed 32359627 --concurrency 4
```

We trained 20 draft agents (ten first players and second players) of each combination of <approach> and <battle agent>, using the best sets of hyperparameters found for them in the previous experiment. That comprises ten runs of the script above, in which we used the seeds 32359627, 91615349, 88803987, 83140551, 50731732, 19279988, 35717793, 48046766, 86798618 and 62644993. To learn about other script's parameters, execute it with the --help flag. Running the script with all default parameters will train a immediate drafter with the *max-attack* battler, using the best set of hyperparameters we found for that combination. Each run of the script took around 50 minutes with the *max-attack* battle agent and around three hours with the *greedy* battle agent.

A.3.3 Comparison with other draft strategies

To run the tournament, simply execute the tournament.py script:

```
python3 gym_locm/experiments/tournament.py \
--drafters random max-attack coac closet-ai icebox \
    gym_locm/trained_models/<battle_agent>/immediate/ \
    gym_locm/trained_models/<battle_agent>/history/ \
    gym_locm/trained_models/<battle_agent>/lstm/ \
```

```
--battler <battle_agent> --concurrency 4 --games 1000 \
--path tournament_results/ --seeds 32359627 91615349 88803987 \
83140551 50731732 19279988 35717793 48046766 86798618 62644993
```

To run either of the two tournaments we described, the `<battle_agent>` portion should be replaced with either `max-attack` or `greedy`. The script will create files at the `tournament_results/` folder describing the individual win rates of every set of matches, the aggregate win rates, average mana curves and every individual draft choice made by every agent, in CSV format, for human inspection, and as serialized Pandas[3] data frames (PKL format), for easy further data manipulation. To learn about other script's parameters, execute it with the `--help` flag.

To reproduce the plot containing the agent's three-dimensional coordinates found via Principal Component Analysis and grouped via K-Means (Figure 5.5), simply execute the `similarities.py` script:

```
python3 gym_locm/experiments/similarities.py \
--files max_attack_tournament_results/choices.csv \
greedy_tournament_results/choices.csv
```

Its execution takes a few minutes and will result in a PNG image saved to the folder in which the script was executed.

A.3.4 Agent improvement in the CoG 2019 LOCM tournament

We used the source-code of the Strategy Card Game AI (SCGAI) competition[4] to re-execute the matches of its CoG 2019 edition, replacing the *max-attack* player (named *Baseline2*) with a personalized player featuring our best draft agent and the battle portion on the *max-attack* player. This can be reproduced by altering line 11 of the runner script (`run.sh`) from `AGENTS[10]="python3 Baseline2/main.py"` to:

```
AGENTS[10]="python3 gym_locm/toolbox/predictor.py \
--battle \"python3 Baseline2/main.py\" \
--draft-1 gym_locm/trained_models/max-attack/immediate/1st/6.json
--draft-2 gym_locm/trained_models/max-attack/immediate/2nd/8.json"
```

[3] Available at `https://pandas.pydata.org`.
[4] Available at `https://legendsofcodeandmagic.com/COG19`.

Then, to execute the tournament, run the `run.sh` script. Parallelism can be achieved by running the script in multiple processes or machines. Save the output to text files named `out-*.txt` (with a number instead of `*`) in the same folder, then run the `analyze.py` script to extract win rates. The runner script can take up to several days, and the analyze script can take up to some hours.

A.4 Using our trained agents

In the `gym_locm.trained_models` module there are all draft agents we trained and considered in our experiments. They are organized in the following folder structure:

`trained_models/<battle_agent>/<draft_approach>/<player>/<file>.(zip|json)`

Where `<battle_agent>` means which battle agent played the battles while they were being trained (either `max-attack` or `greedy`), `<approach>` means the draft approach that was used (either `immediate`, `history` or `lstm`), `<player>` means which role they were trained for (either `1st` or `2nd`), and `<file>` is a number from 1 to 10 that represents the training session in which the agent was trained.

The network models are represented as ZIP files, which can be loaded by the *stable-baselinse* library, and JSON files, which can be used in our `predictor.py` script in the `gym_locm.toolbox` module. To do the latter, execute it in one of the two scenarios below:

1. To use different draft agents when playing first and second, with a battler called `player.py`:

   ```
   python3 gym_locm/toolbox/predictor.py \
   --draft-1 gym_locm/trained_models/greedy/immediate/1st/4.json \
   --draft-2 gym_locm/trained_models/greedy/immediate/2nd/3.json \
   --battle "python3 /path/to/player.py"
   ```

2. To use the same model when playing first and second, with a battler called `player`:

   ```
   python3 gym_locm/toolbox/predictor.py \
   --draft gym_locm/trained_models/max-attack/history/1st/5.json \
   --battle "./path/to/player"
   ```

Attachment A

Hyperparameters

This attachment provide the collection of hyperparameters used in our experiments, following the hyperparameter optimization methodology described in Section 5.2.

We conducted a separate hyperparameter search for each combination of approach (Immediate, History or LSTM) and playing strategy (*max-attack* or *greedy*). Each hyperparameter search consisted in running 50 training sessions using different sets of hyperparameters, feeding back to the optimization algorithm the greatest win rate achieved in each of them.

At the end of a search, we selected the two sets of hyperparameters among the carried training sessions that maximized the win rate of the first and second player networks. These set of hyperparameters were then used to obtain the results for the first and second player networks of each approach in our experiments.

Table A.1 lists the hyperparameters that we optimized, as well as their value ranges and origin. We choose the value ranges empirically and based on previous knowledge about neural networks and the Proximal Policy Optimization algorithm. Tables A.2 and A.3 shows the best sets of hyperparameters found to train each approach with the *max-attack* and *greedy* battle agents, respectively.

Hyperparameter	Value range	Origin
Update frequency of the opponent network	Every 10, 100 or 1000 episodes	Self-play
Depth of the network	1 to 3 layers	Network architecture
Size of the hidden layers	24 to 256 neurons	
Activation function of neurons in hidden layers	TanH, ReLU or ELU	
Batch size	30 to 300 steps	PPO Algorithm
Amount of mini-batches	1 to 300	
Amount of epochs to train with each batch	3 to 20 epochs	
Clipping range of the loss function	0.1, 0.2 or 0.3	
Weight of the value function in the loss function	0.5 or 1.0	
Weight of the entropy term in the loss function	$[0, 0.01]$	
Learning rate	$[0.01, 0.00005]$	

Table A.1: Hyperparameters optimized in our methodology, their value ranges and origin.

Hyperparameter	Immediate		History	LSTM	
	1st player	2nd player	Both players	1st player	2nd player
Update frequency of the opponent network	Every 10 ep.	Every 10 ep.	Every 100 ep.	Every 100 ep.	Every 100 ep.
Depth of the network	3 layers	1 layer	1 layer	3 layers	1 layer
Size of the hidden layers	52 neurons	81 neurons	93 neurons	25 neurons	24 neurons
Activation function of neurons in hidden layers	ELU	TanH	TanH	TanH	TanH
Batch size	300 steps	300 steps	240 steps	240 steps	150 steps
Amount of mini-batches	100	150	120	1	1
Amount of epochs to train with each batch	3 epochs	8 epochs	3 epochs	17 epochs	10 epochs
Clipping range of the loss function	0.1	0.1	0.3	0.1	0.1
Weight of the value function in the loss function	0.5	0.5	0.5	1.0	0.5
Weight of the entropy term in the loss function	8.48×10^{-3}	6.34×10^{-3}	6.59×10^{-3}	2.06×10^{-3}	5.39×10^{-3}
Learning rate	1.38×10^{-4}	1.62×10^{-4}	3.68×10^{-4}	4.57×10^{-4}	4.01×10^{-4}

Table A.2: Best sets of hyperparameters found for each of our approaches when paired with the *max-attack* playing strategy.

Hyperparameter	Immediate		History		LSTM
	1st player	2nd player	1st player	2nd player	Both players
Update frequency of the opponent network	Every 1000 ep.	Every 1000 ep.	Every 100 ep.	Every 1000 ep.	Every 1000 ep.
Depth of the network	1 layer	1 layer	1 layer	1 layer	3 layers
Size of the hidden layers	169 neurons	154 neurons	199 neurons	199 neurons	51 neurons
Activation function of neurons in hidden layers	ELU	TanH	ELU	ELU	TanH
Batch size	270 steps	270 steps	270 steps	270 steps	210 steps
Amount of mini-batches	135	135	135	135	1
Amount of epochs to train with each batch	20 epochs	5 epochs	4 epochs	18 epochs	16 epochs
Clipping range of the loss function	0.1	0.1	0.3	0.1	0.1
Weight of the value function in the loss function	1.0	0.5	1.0	1.0	1.0
Weight of the entropy term in the loss function	5.95×10^{-3}	7.93×10^{-3}	7.23×10^{-3}	4.36×10^{-3}	8.55×10^{-3}
Learning rate	2.28×10^{-4}	1.74×10^{-4}	6.16×10^{-5}	5.00×10^{-5}	1.11×10^{-4}

Table A.3: Best sets of hyperparameters found for each of our approaches when paired with the *greedy* playing strategy.