Exploring Deep Reinforcement Learning for Battling in Collectible Card Games

Ronaldo e Silva Vieira
Dep. de Ciência da Computação
Univ. Federal de Minas Gerais
Belo Horizonte, Brazil
ronaldo.vieira@dcc.ufmg.br

Anderson Rocha Tavares
Instituto de Informática
Univ. Federal do Rio Grande do Sul
Porto Alegre, Brazil
artavares@inf.ufrgs.br

Luiz Chaimowicz
Dep. de Ciência da Computação
Univ. Federal de Minas Gerais
Belo Horizonte, Brazil
chaimo@dcc.ufmg.br

***Abstract*—Collectible card games (CCGs), such as *Magic: the Gathering* and *Hearthstone*, are a challenging domain where game-playing AI arguably has not yet reached human-level performance. We propose a deep reinforcement learning approach to battling in CCGs, using *Legends of Code and Magic*, a CCG designed for AI research, as a testbed. To do so, we formulate the battles as a Markov decision process, train agents to solve it, and evaluate them against two existing agents of different skill levels. Contrasting with the current state-of-the-art, our resulting agents act fast and can play many battles per second, despite their limited performance. We identify limitations and discuss several promising directions for improvement.**

***Index Terms*—collectible card games, reinforcement learning, artificial intelligence**

I. Introduction

In collectible card games (CCGs), such as *Magic: the Gathering* or *Hearthstone*, players build a deck from a broad set of cards representing creatures and spells and use it to battle other players. From an AI standpoint, CCG battles are turn-based, two-player games containing hidden information, non-determinism, large combinatorial state and action spaces, and rules that may change throughout the game. These factors make them a more challenging domain than games like Go and Texas Hold'em Poker, the protagonists of recent breakthroughs in game-playing AI [1], [2].

Fast human-level AI battlers for CCGs would enable better playtesting tools and help CCG designers in the difficult task of game balancing. They would also provide challenging opponents for human players. Current state-of-the-art primarily relies on tree-search algorithms [3]–[8], which require seconds to play a single battle. Reinforcement learning [9] and ontology-based [10] approaches were also proposed. However, to our knowledge, human-level performance has not been achieved yet [6], [10].

We propose a deep reinforcement learning approach for battles in collectible card games. To do so, we formulate battling as a Markov decision process and train deep neural networks with a variant of the Proximal Policy Optimization algorithm (PPO) [11] to solve it. The resulting agents receive a representation of the game state as input, process it on a multilayer perceptron, and output a single action to be performed in-game. Without using search, the agents can play many battles per second.

We use *Legends of Code and Magic* 1.2 (LOCM) as a testbed. LOCM is a simple, finite, and deterministic CCG designed especially for AI research. We evaluate our resulting agents in battles with random decks against two agents of different skill levels. We conduct a hyperparameter tuning and verify that our agents are faster than the state-of-the-art yet that they achieve a limited win rate.

Our main contribution is a deep reinforcement learning approach to the battling problem in CCGs that yields fast agents and is trainable on a single desktop computer with modest configuration. Secondary contributions are: (i) an analysis and discussion of factors that may have restrained the agents' performance, pointing promising research directions; and (ii) reproducible experiments using exclusively open-source libraries.

II. Related Work

There are many commercial CCGs available. However, most of the current literature on game-playing AI for CCGs concentrates on the two most popular: *Magic: the Gathering* and *Hearthstone*. The two games share many of their core rules, thus, we will address their literature indistinguishably. In 2018, *Legends of Code and Magic* was proposed [12], aiming to foster research on CCGs with a reduced yet representative subset of the rules found in commercial CCGs.

By far, the current most successful approaches to battling use tree-search algorithms, which examine portions of the game-tree to reach a decision. In the game tree, nodes represent states and edges represent actions, connecting a state to its successor given that action. The root node is the current game state, where the player must make a move. Leaf nodes are terminal states, where the game finishes.

Since the usual amount of nodes in a CCG game tree makes it unfeasible to explore the entire tree, the maximum tree height is usually limited. Upon reaching the maximum height, the deepest nodes are scored with an heuristic, which is often machine learned [6], [9], [13], evolved with evolutionary algorithms [7], [14], or handcrafted [5]. Algorithms such as *Monte Carlo Tree Search* [15] may evaluate a node by estimating its win rate via simulated battles between simple

This work was supported by CAPES, CNPq, and Fapemig.

agents, beginning at that node. This agent may be random [3], heuristic [3] or machine learned [8]. The hidden information present in CCGs is not handled by tree-search methods by default. Most approaches ignore it, but some efforts were made to determinize it [4], i.e., sample possible values to all unknown data, or predict [16] it.

In all battling literature, the feature extraction process is often a collection of all numeric variables in the game state and may include some hand-engineered features. *Magic: the Gathering* cards and *Hearthstone* cards may have its in-game effects described in natural language, what makes thoroughly extracting card features an AI challenge on its own. While most work on battling so far simply ignores card text, some efforts have been made using *word2vec* models [6], [17] and *long short-term memory* layers [18].

Either for calculating next states in tree searches, generating datasets of game states or learning by interaction with reinforcement learning, a *forward model* is needed. Most of the literature use open-source implementations of the game's rules, such as Magarena,[1] Sabberstone,[2] or Metastone,[3] which often come with limitations (such as not having all cards or all rules) but are considered close enough to the original game.

A single work uses reinforcement learning to tackle the battling task [9] on *Hearthstone*. They define a basic set of features (both players' health points, attributes of the cards present in the board and some domain-knowledge features) and apply the Q-learning algorithm with separate multilayer perceptron neural networks for playing cards and for attacking with creatures. The output indicates the quality values (Q-values) of every possible action on that specific game state.

We believe that deeper neural networks than used in [9] trained with deep reinforcement learning algorithms can thoroughly encode raw CCG game states (with no feature engineering), capture the intricate relationship between cards and rules, and select actions that efficiently lead a player to victory. Since there is no tree search involved and due to GPU-based parallelism, we argue that the resulting battle agents would act faster than the current state-of-the-art.

III. Legends of Code and Magic

We instantiate our approach on *Legends of Code and Magic* (LOCM) version 1.2. In LOCM, there are two main types of cards: creatures and items. Creature cards are used to attack the opponent creatures or the opponent, while item cards are used to apply varied effects such as increasing a creature's attack attribute or removing all of its abilities. Figure 1 shows a creature card in LOCM 1.2.

In a battle, each player starts with 30 health points and one mana point, which is recharged and increased by one each turn (up to a maximum of 12). They also start with four cards in their hand (drawn from their shuffled deck) and draw one more each turn. The battle ends when a player reaches zero

[1]https://magarena.github.io

[2]https://github.com/HearthSim/SabberStone

[3]https://github.com/demilich1/metastone

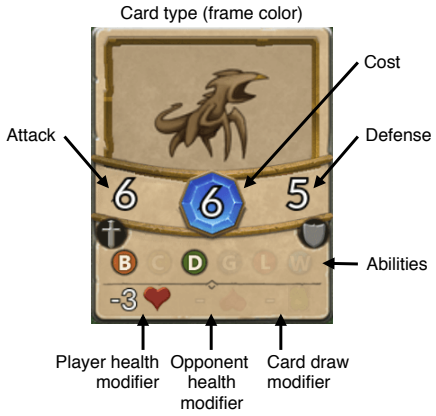

Fig. 1. An example of a LOCM 1.2 creature card and its features.

or fewer health points, and their opponent wins. The players take turns in which they can:

- **Summon a creature**: spend mana equivalent to a creature card's cost to place it on the board. The player must choose which of the two lanes to place the creature.
- **Use an item**: spend mana equivalent to an item card's cost to apply the item's effects to a target. The target may be any creature on the board or the opponent.
- **Attack with a creature**: select one of their creatures on the board to deal damage equivalent to that creature's attack attribute to a target. The target may be the opponent or any creature on the same lane as the attacking creature. Damage on creatures reduces their defense attribute (any creature with 0 or fewer defense points is removed from the board), while damage on players reduces their health points. Creatures can attack once per turn and cannot attack the turn they were summoned.
- **Pass the turn**.

While LOCM's rules are simpler than those of commercial CCGs, they still represent the essential characteristics of the genre: drawing cards from a custom deck, using mana to play cards, having creature and spell/item cards with different abilities, and combat with creatures. LOCM is also finite: each player can have up to eight cards in their hand and six creatures on the board (three on each lane). A creature may have any combination of the six abilities present in the game, which affect the combat rules regarding that creature.[4]

IV. Methodology

To tackle LOCM's battle with deep reinforcement learning, we formulate it as a Markov decision process (MDP). The resulting episodic MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$ whose elements are defined next:

- The **set of states** $\mathcal{S}$ contains all possible game states of a battle turn in LOCM. A game state in a battle consists of all the information the active player can observe at any given moment:

[4]For a comprehensive list of LOCM 1.2's rules, see https://github.com/acatai/Strategy-Card-Game-AI-Competition/blob/master/GAME-RULES.md

– The players' statistics (4 features, 2 players);
– Which cards are in the active player's hand (8 card slots, 160 possible cards, 1 player);
– Which creatures are on the players' board (6 creature slots, 9,216 possible creatures, 2 players).

Hence, the size of $\mathcal{S}$ is:

$$|\mathcal{S}| = (2 \times 4) \times (160^8) \times (2 \times 9{,}216^6) \approx 4.21 \times 10^{42}.$$

- The **set of actions** $\mathcal{A}$ at any turn consists of the actions available to the active player. A player can always `PASS` their turn, `SUMMON` creature cards from their hand to one of the lanes, `USE` item cards from their hand targeting a creature on the board or their opponent, or make their creatures on the board `ATTACK` another opponent creature on the same lane or the opponent. The possible combinations of card indexes on hand, lanes, item targets, attacking creatures, and attacking targets yield a total of 1 `PASS` action, 16 `SUMMON` actions, 104 `USE` actions, and 24 `ATTACK` actions, thus:

$$|\mathcal{A}| = 1 + 16 + 104 + 24 = 145.$$

- The **transition function** $T(s_{t+1}|s_t, a_t)$ follows the rules of the battle.
- The **reward function** $\mathcal{R}(s)$ rewards the agent at the end of the battle with 1 if they won and -1 if they lost. Formally,

$$\mathcal{R}(s) = \begin{cases} 0, & \text{if } s \text{ is non-terminal,} \\ 1, & \text{if } s \text{ is terminal and the battle was won,} \\ -1, & \text{if } s \text{ is terminal and the battle was lost.} \end{cases}$$

- The **discount factor** γ equals 0.99 to slightly encourage shorter episodes yet still reward all choices in a battle since all may contribute equally to whichever reward the agent obtains at the end.

Finding a solution to this MDP is equivalent to developing a strategy to battle in LOCM. In other words, an agent can play a battle following any policy $\pi(a|s)$ that maps every state s in $\mathcal{S}$ to a probability distribution over every action a in $\mathcal{A}$.

We tackle the battle MDP with deep reinforcement learning. In other words, we train a deep neural network to act as a player in the game. In each battle turn, we convert the game state to a numeric vector containing the relevant features normalized to the range of $[-1, 1]$. A zero vector represents empty card slots on the player's hand and board. We then give this numeric vector as input to the network, which outputs its policy (a probability distribution over all 145 actions). At last, we sample an action from the policy and perform it in the game. Figure 2 illustrates this interaction loop.

We use a variant of the Proximal Policy Optimization (PPO) algorithm [11], a standard go-to algorithm in deep reinforcement learning. Our network architecture comprises a standard multi-layer perceptron with two output heads. The first is a linear layer with 145 values, followed by a softmax, representing the policy $\pi(a|s)$ for the game state s received as input. The second is a linear layer with a single value,

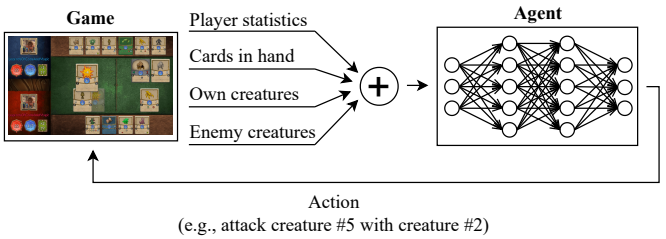

Fig. 2. Interaction loop between our agent and the game during a battle in LOCM. The agent receives a representation of the game's current state and decides which action to perform.

representing the value function $V_\pi(s)$ for the game state s received as input (PPO uses this estimated value function to calculate its loss).

Despite having 145 different actions, not all are always valid actions. For instance, if a player has no cards in their hand, then all `SUMMON` and `USE` actions are invalid. Most states in $\mathcal{S}$ have only a few valid actions, and preliminary experiments revealed that this sparseness critically impacts the convergence of reinforcement learning algorithms. Thus, we use a variant of the PPO algorithm with *invalid action masking* [19]: before the softmax activation, all logits that refer to invalid actions are set to $-\infty$. As a result, they have zero probability in the resulting softmax distribution.

Our feature extraction process works as follows: we select four relevant features from the statistics of each player, namely, their health points, mana points, next rune, and amount of cards to be drawn next turn. From the cards in the active player's hand, we select the card type, cost, attack, defense, abilities, health modifiers, and card draw modifiers. From the creatures in each player's board, we select attack, defense, abilities, and whether the creature can attack this turn (if it belongs to the active player).

Except for the card type and abilities, all features are already numeric. We normalize them by dividing each feature by its maximum absolute value. We apply one-hot encoding to convert the card type, a categorical feature. We then convert the remaining binary features by considering true and false values as 1 and 0. Table I shows the total number of features in the game state and each of its parts.

TABLE I
NUMBER OF FEATURES IN A GAME STATE AND IN EACH OF ITS PARTS.

Feature group	Features per unit	Amount	Total features
Player statistics	4	2	8
Card in hand	16	8	128
Own creature on board	9	6	54
Enemy creature on board	8	6	48
Total			**238**

V. Experiments

Next, we present the setup for our experiments (Section V-A), the tuning of the hyperparameters of our approach (Section V-B), and the resulting agents and their performance (Section V-C).

A. Setup

We used the *stable-baselines3* library (version 1.4.0) [20] to train our agents and the MaskablePPO implementation of the PPO algorithm present in the auxiliary *sb3-contrib* library (version 1.4.0). As a forward model, we used the *gym-locm* library (version 1.3.0) [21], which contains an open-source implementation of LOCM 1.2's rules exposed as OpenAI Gym [22] environments, to facilitate the use of reinforcement learning algorithms. Using the Gym paradigm, we minimize the agent-game communication overhead present in the original engine, which is critical for approaches that require simulation of large amounts of matches (most so far).

We trained the agents for 100,000 episodes in self-play, i.e., the opponent agent faced during training is an older version of the agent itself. We updated the opponent's network parameters to a copy of the agent's network parameters from time to time. Following best practices in reinforcement learning experimenting [23], we stopped training every 2,000 episodes to save the network's parameters and evaluate the agent in an offline manner: the agent faced a fixed set of opponents (so all network evaluations are comparable) during 500 episodes each, and we extracted its win rate and other statistics. During both training and evaluation, both battle agents played with random decks, and the training agent and their opponent switched roles (who plays first and second) every episode.

To evaluate, we used two different battle agents. The first, called *max-attack* (MA), is one of the baselines in the *Strategy Card Game AI competition*. The second, called *one-step look-ahead* (OSL), is one of the agents used in [24]. MA is a simple rule-based agent that can play thousands of battles per second with limited performance, and OSL has greater performance at the cost of runtime speed. While a state-of-the-art agent would be ideal, their runtime speed is prohibitive.

We conducted all training sessions on a machine with an Intel Core i7-8700 3.2GHz processor, 16GB of RAM, and an NVIDIA GeForce GTX 1050 graphic card with 4GB of VRAM. We used Python 3.8.10, PyTorch 1.11.0, CUDA 11.4, and the NVIDIA driver version 470.129.064 in Ubuntu 20.04. We used 4-core CPU parallelism for battle simulations and the GPU for neural network operations. The experiments used a small fraction of the machine's memory and computing power. The experiment code and instructions to reproduce them are available on GitHub.[5]

B. Hyperparameter Tuning

We used the Bayesian tuning method of *Weights & Biases* [25], which uses a Gaussian Process for hyperparameter optimization. Our objective function is the win rate versus OSL. We executed 35 different training sessions, each with a different set of hyperparameters chosen by the Bayesian method. Table II lists the hyperparameters we optimized and their search ranges. The remaining hyperparameters were set to a reasonable value.

TABLE II
Hyperparameters optimized, their value ranges and where they originate.

Hyperparameter	Value range	Origin
Opponent update frequency	Every 10, 100, or 1,000 episodes	Self-play
Depth of the network	3 to 12 layers	Neural net.
Width of the hidden layers	32 to 512 neurons	
Batch size	64, 128, 256, 512, 1024 or 2048 steps	PPO algorithm
Amount of mini-batches	Batch size divided by 1, 2, 4, 8, or by itself	
Amount of epochs	1 to 24 epochs	
Learning rate	1×10^{-2} to 1×10^{-6}	

We chose the most promising set of hyperparameters after evaluating the metrics from the best five and used the following values to train our final agents: Batch size: 512, minibatch size: 1, epochs: 1, clip range: 0.2, entropy coefficient: 0.005, value function coefficient: 1, Num. of hidden layers: 7, Num. of neurons in hidden layers: 455, activation function of hidden layers: ReLU, learning rate: approximately 4.114×10^{-3}, opponent network update frequency: every 10 episodes.

C. Results

With the chosen set of hyperparameters, we repeated the training session five times with different random seeds to increase the statistical significance of the results. Figures 3 and 4, respectively, show the win rates of the agents during training in self-play and during evaluation against MA and OSL.

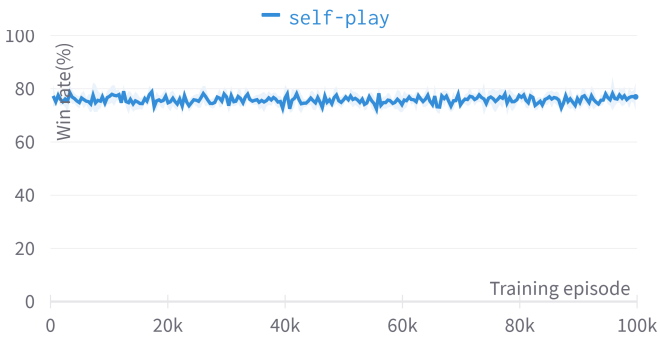

Fig. 3. Win rate of our self-play agents during the 100,000 training episodes, playing against an earlier version of themselves (self-play). The line represents the mean, while the (barely visible) shaded area represents the standard deviation. Every point in the line corresponds to the win rate of all training episodes since the last update of the opponent's network parameters.

[5]https://github.com/ronaldosvieira/gym-locm/tree/1.3.0/gym_locm/experiments/papers/sbgames-2022

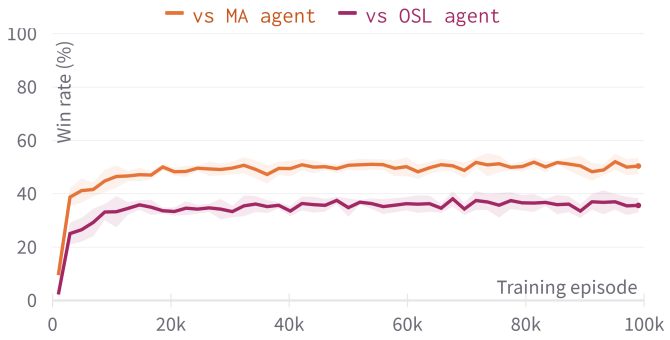

Fig. 4. Win rate of our self-play agents during evaluation against MA and OSL across the 100,000 training episodes. The lines represent the mean, while the shaded areas represent the standard deviation.

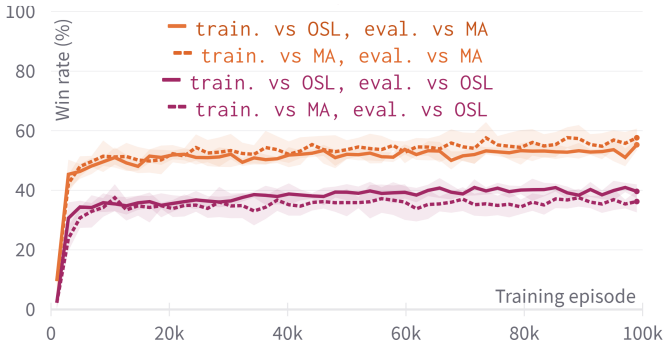

Fig. 5. Win rate of our non-self-play agents during evaluation against MA and OSL across the 100,000 training episodes. The lines represent the mean, while the shaded areas represent the standard deviation. Dashed lines represent agents trained vs. MA, and solid lines represent agents trained vs. OSL.

The results show that our best agents won approximately 51% and 36% of the battles against the MA and OSL agents. The best scores in all training sessions against them were 55.8% and 42.6%, respectively. This difference is expected since OSL is a more powerful agent than MA. Most of the increase in win rate happens in the first 10,000 episodes, after which the agents kept increasing but at a lower rate. Results also show that our agents maintain a win rate of 76% with little deviation when in self-play. Since we expect the win rate against a freshly updated opponent to be close to 50%, our agents seem to learn to outperform their old version quickly.

We also analyzed metrics other than win rate. Taking less than 2,000 training episodes, our agents converged to make about 3.6 actions per turn: 44% attack actions, 22% summon actions, 10% use actions, and 24% pass actions. These percentages are congruent with CCG battles in general, which demand more combat than board-management actions. A battle lasted 7.25 turns on average against MA and 7.5 turns against OSL.

On average, a training session lasted 1 hour and 27 minutes, spending approximately half that time evaluating against the OSL battle agent. A forward pass in the neural network of a trained agent returns a single action and takes around two milliseconds, including the time required to calculate the valid actions and to advance the forward model's state. Considering 3.6 actions per turn on average, playing an entire turn takes around 7.2 milliseconds. This result indicates that our approach is faster than current state-of-the-art LOCM agents, which take 40 to 200 milliseconds to act in a battle turn, depending on the density of the state.

As an additional comparison, we repeated our experiment twice, training directly against the MA and OSL agents. The training setup and random seeds were identical, except for the opponent faced in the training episodes. Figure 5 shows the win rates of the agents throughout the training against MA and OSL when evaluated against MA and OSL. The results showed that the statistics were similar to the self-play experiment, except that the agents could attain slightly better win rates especially when evaluating against the same agent they faced during trained (usually 4-5 percentage points higher).

VI. Discussion

According to our experiments, our deep reinforcement learning approach yields fast agents that do not require powerful hardware or many days or weeks of computation to train. Their performance, however, lies considerably below the state-of-the-art, which would win almost 100% of the battles against MA (as seen in the *Strategy Card Game AI competition*). Below we discuss limitations in our current approach and ways to tackle them and improve the agents' performance.

Diverse opponents. Ten episodes seemed enough for our agents to outperform their self-play opponents during training. However, their performance during evaluation against MA and OSL did not increase that fast. Our agents may be in a rock-paper-scissors-like loop where they can easily beat their earlier version with a simple change in their strategy and obtain good rewards without necessarily making progress against other opponents. Moreover, training against a single fixed opponent seemed to encourage our agents to overfit their strategy to that agent. A solution may be to diversify the opponents seen during training: simultaneously train in self-play (maybe against many earlier versions) and against other agents.

Permutation-invariance. Our current state and action representations are prone to a positional bias: a card in the first slot of the player's hand is no more important than a card in the fifth slot, yet, there are far more states where a card in the first slot exists than in the fifth slot. Hence, the agent is more likely to play the first card rather than the fifth regardless of their relative quality because the agent will see much more samples with the first card being played than the fifth. Moreover, our current formulation considers card permutations (e.g., a state with cards A and B in hand and another with B and A) as different states, greatly enlarging the state space. A promising direction may be to incorporate permutation-invariant network architectures, such as in Set Transformers [26].

Reward shaping. Our current reward model is sparse, i.e., most states receive no reward. Thus, the agent is guided only by whether it wins the battle, disregarding good choices made during defeats. In reinforcement learning, reward shaping is

a common technique used to speed-up learning by giving non-zero rewards for performing actions that usually lead to victory [27]. Possible alternatives for CCGs include rewarding whenever the opponent loses health or a card on the board. Adjusting the win-loss rewards by a factor proportional to the length of the battles may also encourage shorter wins and lengthier losses.

Deal with the combinatorial action space. Some approaches in large combinatorial action spaces train multiple policies: for each "actionable unit" in the game (in our case, cards in hand or on the board), there is a policy to select which action to make (including no-op) and their parameters [28]. This modeling makes training more efficient and mitigates positional bias but would require multiple forward passes of a single network to select actions for all cards in hand and on the board. Other strategies to handle large combinatorial action spaces we consider is to learn policies over algorithms [29] or proto-actions [30] instead of actions. Using separate networks for each action type, like in [9], also reduces the action space.

Dedicated card-encoder network. A significant part of the game states in CCGs consists of cards. However, our network architecture currently has no dedicated component for encoding cards. We expect using a separate card-encoder network to encode all cards before inputting them to the primary network to be beneficial, especially when applying our methodology to commercial CCGs, where natural language processing will be required.

Tackle hidden information. Our current MDP formulation disregards the existence of the opponent player, treating his actions and hidden information as part of the stochasticity of the problem. We consider tackling these aspects either explicitly (using a partially observable MDP) or implicitly (e.g., using a probability model of the hidden information as part of the game state).

VII. Conclusion

In this paper, we proposed a deep reinforcement learning approach to the problem of battling in collectible card games. We formulated it as a Markov decision process and used a variant of the Proximal Policy Optimization algorithm to train agents in self-play, with *Legends of Code and Magic* as a testbed. We evaluated the resulting agents against two existing battle agents, considering the win rate and other metrics.

While our agents' performance does not reach the current state-of-the-art, they act considerably faster, being able to play many entire battles per second. We analyzed the results, discussed limitations of the current approach, and pointed out promising directions to solve them and improve performance. We intend to tackle some of these directions as future work.

We consider this work a step toward superhuman game-playing agents for collectible card games, which we understand as one of AI's current milestones. We hope it encourages further research on this challenging and promising topic.

References

[1] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, 2017.

[2] N. Brown and T. Sandholm, "Superhuman AI for multiplayer poker," *Science*, 2019.

[3] C. D. Ward and P. I. Cowling, "Monte Carlo search applied to card selection in Magic: The Gathering," in *CIG*, 2009.

[4] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble determinization in Monte Carlo tree search for the imperfect information card game Magic: The Gathering," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 4, no. 4, 2012.

[5] A. Santos, P. A. Santos, and F. S. Melo, "Monte Carlo tree search experiments in hearthstone," in *CIG*, 2017.

[6] M. Swiechowski, T. Tajmajer, and A. Janusz, "Improving Hearthstone AI by combining MCTS and supervised learning algorithms," in *CIG*, 2018.

[7] H. Chia, T. Yeh, and T. Chiang, "Designing card game strategies with genetic programming and monte-carlo tree search: A case study of hearthstone," in *SSCI 2020*. IEEE, 2020.

[8] T. Papagiannis, G. Alexandridis, and A. Stafylopatis, "Applying gradient boosting trees and stochastic leaf evaluation to MCTS on hearthstone," in *ICMLA 2020*. IEEE, 2020.

[9] I. Kachalsky, I. Zakirzyanov, and V. Ulyantsev, "Applying reinforcement learning and supervised learning techniques to play hearthstone," in *ICMLA 2017*, 2017.

[10] A. Stiegler, K. P. Dahal, J. Maucher, and D. J. Livingstone, "Symbolic reasoning for hearthstone," *IEEE Trans. Games*, vol. 10, no. 2, 2018.

[11] J. Schulman *et al.*, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[12] J. Kowalski and R. Miernik, "Legends of Code and Magic," https://legendsofcodeandmagic.com, 2018, accessed: 2021-12-14.

[13] D. Wang and T. Moh, "Hearthstone AI: Oops to well played," in *ACMSE*, 2019.

[14] P. García-Sánchez, A. P. Tonda, A. J. F. Leiva, and C. Cotta, "Optimizing hearthstone agents using an evolutionary algorithm," *Knowl. Based Syst.*, vol. 188, 2020.

[15] C. Browne *et al.*, "A survey of monte carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, 2012.

[16] A. Dockhorn, M. Frick, Ü. Akkaya, and R. Kruse, "Predicting opponent moves for improving hearthstone AI," in *IPMU 2018*. Springer, 2018.

[17] A. Janusz, L. Grad, and D. Slezak, "Utilizing hybrid information sources to learn representations of cards in collectible card video games," in *ICDM Workshops*, 2018.

[18] G. L. Zuin, L. Chaimowicz, and A. Veloso, "Deep learning techniques for explainable resource scales in collectible card games," *IEEE Trans. Games*, 2022.

[19] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," *CoRR*, vol. abs/2006.14171, 2020. [Online]. Available: https://arxiv.org/abs/2006.14171

[20] A. Raffin *et al.*, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[21] R. Vieira, A. Rocha Tavares, and L. Chaimowicz, "OpenAI Gym Environments for Legends of Code and Magic," 10 2020. [Online]. Available: https://github.com/ronaldosvieira/gym-locm

[22] G. Brockman *et al.*, "OpenAI Gym," *CoRR*, vol. abs/1606.01540, 2016.

[23] C. Colas, O. Sigaud, and P. Oudeyer, "A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms," in *Reproducibility in Machine Learning, ICLR 2019 Workshop*. OpenReview.net, 2019.

[24] J. Kowalski and R. Miernik, "Evolutionary approach to collectible card game arena deckbuilding using active genes," *arXiv e-prints*, p. arXiv:2001.01326, 2020.

[25] L. Biewald, "Experiment tracking with weights and biases," 2020. [Online]. Available: https://www.wandb.com/

[26] J. Lee *et al.*, "Set transformer: A framework for attention-based permutation-invariant neural networks," in *ICML*. PMLR, 2019.

[27] M. J. Mataric, "Reward functions for accelerated learning," in *Machine Learning, Proceedings of the Eleventh International Conference*, 1994.

[28] S. Huang, S. Ontañón, C. Bamford, and L. Grela, "Gym-μrts: Toward affordable full game real-time strategy games research with deep reinforcement learning," in *CoG 2021*. IEEE, 2021, pp. 1–8.

[29] A. R. Tavares, S. Anbalagan, L. S. Marcolino, and L. Chaimowicz, "Algorithms or actions? A study in large-scale reinforcement learning," in *IJCAI 2018*, J. Lang, Ed. ijcai.org, 2018.

[30] Dulac-Arnold *et al.*, "Deep reinforcement learning in large discrete action spaces," *arXiv preprint arXiv:1512.07679*, 2015.