Reinforcement Learning in Collectible Card Games

Student Name: A. Goodall

Supervisor Name: Prof. M. Bordewich

Submitted as part of the degree of [BSc Computer Science] to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract —

Context/Background - Reinforcement learning (RL) has become an increasingly popular area of research in recent years. RL provides us with a very generalised and flexible framework and so it can be applied to many goal-driven problems. In this paper we propose a RL approach to a popular class of problems that exhibit dynamic and challenging properties - Collectible Card Games (CCGs).

Aims - The aim of this project is to explore current state of the art approaches for dealing with CCGs, and to implement and evaluate extensions that have been proposed in the modern literature surrounding this class of problems.

Method - We will be looking at Legends of Code and Magic (LOCM) a CCG specifically designed for AI research. We will implement a varienty of agents based on different approaches and algorithms. Throughout the project we will use an OpenAI Gym reimplementation of LOCM for evaluation, validation and training.

Results - Model-based approaches that work by constructing and evaluating a game state tree have been shown to perform best when following the Monte-carlo Tree-Search (MCTS) algorithm. We implemented and evaluated some exensions to this algorithm which increased its win rate by up to 8.5% when playing against itself. By closely following a different more contemporary approach - the Deep Q-Network (DQN) training method; we constructed an agent that outperformed the MCTS agent and achieved comparable strength to some of the best submissions in the recent LOCM competitions.

Conclusions - In this paper we show the power of both model-based and model-free methods for CCGs. Our approaches are very generalised solutions to Markov decision processes (MDPs), which is a much wider class of problems than CCGs, and as such provide us with useful starting points for similar real world problems that can be abstracted to MDPs.

Keywords — Collectible Card Games, Artificial Intelligence, Reinforcement Learning, Monte-Carlo Tree-Search, Neural Networks, Machine Learning

I INTRODUCTION

Reinforcement learning (RL) can be described simply as learning from experience what to do in a given scenario. More formally, our aim is to capture important characteristics of real problems by modelling them as Markov decision processes (MDPs) and learn the relationships between states and actions to maximise some scalar reward signal in a semi-supervised fashion. This gives us a generalised framework to work with and as a result RL forms the basis for a variety of modern artificial intelligence (AI) applications. Perhaps the most signifigant breakthrough in recent years was when RL and deep learning (DL) were both used to acheive super-human level control for some of the original Atari 2600 games (Mnih et al. 2013). This breakthrough lead to the signifigant rise in popularity surrounding RL and really showed how powerful this framework can be.

Games themselves provide us with useful environments for AI research. Games typically come with quick and easy to interface implementations, so gathering statistics and knowledge about a game's dynamics is often not a problem. Many parallels can be drawn between games and real world problems, so RL algorithms that work for these environments can be good starting points.

Recent improvements on Go (Silver et al. 2017) and Poker (Brown & Sandholm 2019) demonstrated the ability for computers to deal with large state spaces and imperfect information. However, not only do classic board games and card games provide challenges for AI, but so to do modern computer games, which have become an increasingly popular area of research in recent years. Collectible Card Games (CCGs) are an interesting class of modern computer games that provide additional challenges for AI agents (Hoover et al. 2020), along side massive state and action spaces, they have dynamic rules determined by cards in play, added non-determinism of certain actions, and potentially more drastic consequences for bad actions. Naturally there is interest in exploring these games for which designing AI is more difficult.

Traditionally it was thought that developing AI for game playing required highly specific domain knowledge and good heuristic evaluation of game states. The aim of this paper is to develop AI agents for CCGs and in doing so show that we can deal with particularly challenging MDPs using a variety of generalised methods.

A Background

Popular CCGs include Hearthstone (Entertainment & Entertainment 2014), Magic: The Gathering (Garfield 1993), and the more recent Gwent (Red 2017). Hearthstone is a free-to-play online game developed and published by Blizzard Entertainment. It is probably the most popular CCG at the moment with over 2500 collectible cards. It is a turn-based card game played between two opponents. The main goal is to reduce your opponents health from 30 to zero at which point you win. Hearthstone has vast amount of rules and different styles of play that lend themselves to different types of decks; this makes it a notoriously difficult game to develop AI agents for.

Just a few years ago Hearthstone became a popular area for AI research, and was explored at the AAIA17 Data Mining Challenge (Janusz et al. 2017) in 2017. Unfortunately there is no recent open source implementation of the game, so extending this research is difficult. As a result we propose studying a different CCG - Legends of Code and Magic (LOCM) (Jakub Kowalski 2018).

LOCM is a two-player turn-based card game that mimics the rules of Hearthstone and Magic: The Gathering in a more AI friendly way. It was specifically designed for AI research, providing a simpler but still representative environment. It lends itself nicely to AI development because all actions are deterministic, so the only non-determinism in the environment is the cards in your opponent's hand and the cards that are drawn.

A LOCM game consists of two phases: the draft phase and the battle phase. The draft phase consists of building a deck of 30 cards; each turn both players are shown the three cards and they each choose one of them to put in their deck to be used in the battle phase, this happens 30 times until both players decks are full. Note that the selection each player makes is hidden from the other player. In the battle phase, each player takes turns playing cards until one of the players has their health reduced from 30 to zero. Playing cards costs mana which is gained over the course of the game, so typically only weaker cards can be played at the start of the game. Note that in this paper we will look purely at the battle phase because it has a much more signifigant impact on the outcome of the game.

LOCM is the perfect environment to explore the use of RL for CCG environments, the readily available source code can help set up a local environment for visualising LOCM play and the OpenAI Gym reimplementation of the game (Vieira et al. 2019*a*) can be used to provide a more controlled environment for training and evaluation. These tools aleviate the current issue with AI development for Hearthstone and other CCGs that aren't necessarily open source.

B Objectives

Our overall goal is to explore some of the state of the art techniques for CCGs and LOCM, with the idea of improving on them by using extensions or similar techniques from the modern literature surrounding AI for game playing and RL in general. Our main objectives are as follows:

- Start by developing some naive agents based on game state evaluation, to act as more sophisiticated adversaries than the trivial random agent.
- Develop an agent based on the popular minimax tree search method.
- Extend this tree search framework and implement the Monte Carlo tree search algorithm (MCTS) for playing LOCM.
- Find extensions to the MCTS algorithm to be implemented and evaluated against the core algorithm.
- Construct an agent that uses neural networks (NNs) to make decisions while playing LOCM. Follow closely the Deep Q-Network (DQN) training method (Mnih et al. 2013), a model-free value-based method that learns via stochastic gradient descent (SGD).
- Evaluate the model's performance by comparing it to the minimax tree search and MCTS agents.

As advanced objectives we have:

- Look at a different family of deep RL algorithms - policy-gradient methods. Evaluate the final performance of this model and the training dynamics observed against the value-based DQN method.
- Modify our best performing agents so that they can be uploaded to codingame for play againt other people's agents. Ensure that our agents have response times of < 200ms per turn, and compare their performance to baseline and state of the art agents used in some of the recent LOCM competitions.

In this paper we detail the ideas behind each of our agents, their relative performance, and the limitations and considerations for their underlying algorithms. We improved on the core MCTS algorithm using some of the extensions detailed in *IIIC*, and successfully applied Q-learning to train a Q-network agent that emulates good LOCM play. As a result we achieved a valid comparison between popular techniques for RL applications and AI for game playing. And furthermore, we present methods for constructing LOCM agents that achieve comparable performance to some state of the art approaches.

II RELATED WORK

AI for game playing has been widely explored for a variety of different games, from solving Tic-Tac-Toe (Yannakakis & Togelius 2018) and Checkers (Schaeffer et al. 2007) to state of the art results for Go (Silver et al. 2017) and Atari 2600 games (Mnih et al. 2013). As a result we have a lot to think about when we want to start tackling CCGs. Most work on Magic: The Gathering, the most widely known CCG, has been done on designing (Summerville & Mateas 2016) and analysing cards (Zilio et al. 2018). With Hearthstone, work has been done on deck-building, game playing and win rate prediction strategies (Janusz et al. 2017). Tree search approaches and heuristic state evaluation using game knowledge or NNs encompass most of the ideas than have been proposed for Hearthstone. And indeed some of the original approaches to LOCM followed similar ideas (Garnier 2018). Approaches to the draft phase mostly consist of statistically determining card rankings. Minimax tree search and variants of the MCTS algorithm were widely used for the battle phase. More sophisticed agents submitted for LOCM competitions use hand crafted heuristics and tree pruning strategies to get the most out of these tree search approaches. In section *IIA* we will outline and review pior work on MCTS for game playing.

More recently a pure reinforcement learning approach to LOCM was proposed (Vieira et al. 2019*b*). This approach proposed the use of policy networks for both the draft and battle phase; it learnt directly from the game state representation (as opposed to pixel data) through self-play. In the battle phase the MDP model considers an episode as all battle actions until the end of the game, the game state representation is all information that the current player can see, and the NN that was proposed takes in as input all the game state information and outputs values for each possible actions. The trained draft model was used for an agent called *ReinforcedGreediness* in the most recent LOCM competition (Jakub Kowalski 2019). In section *IIB* we will outline Q-learning - an important concept for RL, and in section *IIC* we will review prior work on deep learning and how it has been used for RL problems.

A Monte Carlo Tree Search

Monte Carlo tree search (MCTS), is a best-first search technique that has been widely used for the classic adversarial board game Go (Chaslot et al. 2006), a game that was thought to be intractable (Cai & Wunsch 2007). It improves on the Monte Carlo sampling technique (Chaslot et al. 2008) by iteratively simulating games from likely game states to generate information about the likelihood of outcomes. It is an algorithm that lends itself nicely to adversarial turn-based games where both players don't necessarily know what the other player is going to do. From few random simulations little knowledge can be gained, but from lots of simulations good game playing strategies can be derived. During the game the algorithm builds a tree of possible game states from the current game state, and determines the best move from the current state. Figure 1 illustrates the main ideas.

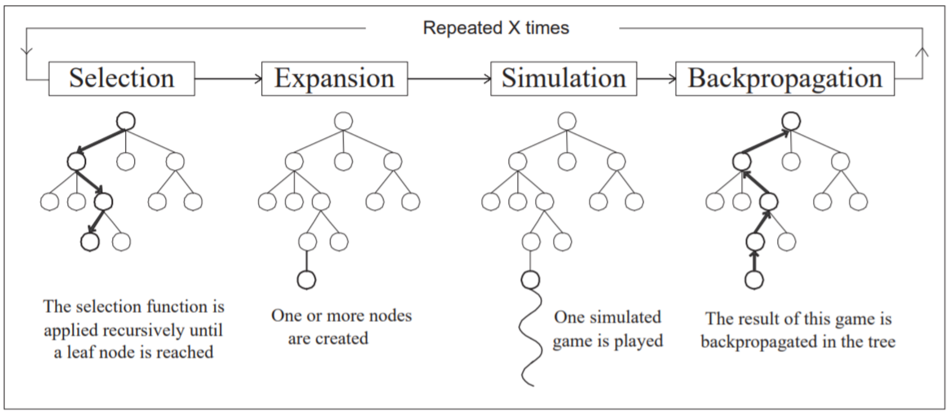
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1: Monte Carlo tree search

Selection - From the current node or game state the algorithm looks at all its possible actions and chooses an action path to start searching down. Conceptually each node in the tree represents a game state and each arc represents exactly one game action. The algorithm selects this action based on some policy, typically we use the Upper Confidence Bounds for Trees (UCT) (Kocsis & Szepesvári 2006), so that the most promising actions are chosen after enough information about them has been gathered. However, there is some chance that we choose a less promising action, and this is because we want to encourage some random exploration (Chaslot et al. 2006). After the action is selected we then continue to select nodes from this new node.

Expansion - An action is chosen for the current node and if we reach an univisted node (a game state we have not explored further), we expand it, adding all its child nodes to the tree.

Simulation - From this new expanded node we simulate a playout of the game according to some playout policy. Typically our playout policy is random and we just pick random actions for the two players until the game is concluded or a terminal state is reached. We return +1 or -1 depending on which player ended up winning the game.

Backpropogation - After the simulated game is finished we propogate the reward back up tree - we update the visit counts and win/loss counts for all the nodes along the path that we traversed. Any game states that we saw during the playout of the game aren't yet part of the tree so no statistics about them are kept.

The win/loss statistics and visit counts are used to inform the agent playing the actual game which move from the current game state is most promising. Typically, we pick the node or action with the highest visit count or highest average wins.

The performance of this algorithm depends on our termination conditions. It is normal to terminate the MCTS roll-out after a certain number of iterations or within a certain time frame. Of course, the more iterations and game simulations we perform the more informed our agent is and the more likely it is to choose the optimal move. Some termination strategies are actually based on this idea of likelihood and only terminate the MCTS roll-out after enough statistical confidence about the best move is gathered (Chaslot et al. 2007).

B Q-Learning

Q-learning forms the basis for a variety of RL techinques such as Q-networks (Riedmiller 2005) and deep q networks (DQNs) (Mnih et al. 2013). Q-learning is a model-free reinforcement learning algorithm that learns the best action to take for some given state. The reason Q-learning underpins a lot of RL ideas is that it is optimal for any finite Markov decision process (FMDP), given infinite exploration time (Watkins & Dayan 1992). In short Q-learning calculates the maximum expected reward for an action taken in any given state, and it can readily be applied to tabular learning methods to solve simple games like Tic-Tac-Toe in a matter of seconds.

Tabular Q Learning - The algorithm constructs a table of state-action value pairs, this effectively gives us a function that determines the quality of an action for any given state,

$$Q : S \times A \to \mathbb{R}$$

where, S is the state space and A is the action space.

The Q table is filled with arbitrary values at the start and we use value iteration, iterating over all state-action pairs to update the values in our table according to the bellman optimality equation described in Figure 2. After enough iterations we will converge to some optimal behaviour. For

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

Figure 2: Bellman Equation

more complex games with larger state and action spaces the size of this table explodes; we can't simply iterate over all actions and all states and this is where deep Q-learning comes into play.

C Deep Q-Learning

Deep Q-learning methods use NNs, typically fitted with additonal convolutional layers, to approximate our Q function from before. This idea works by speeding up the learning of finite problems. It does this because NNs can generalise unseen states to previously seen similar states. Typically, these NNs take as input an obseravtion of the environment or some game state representation and output Q values for each action.

Deep learning itself has been shown to be effective for tasks such as image classification (Krizhevsky et al. 2012) and speech recognition (Dahl et al. 2011). Deep learning techniques have been successful in part due of the large amounts of labelled training data that is available, and the clever sampling technqiues that are used to ensure that datasets are as unbiased as possible. Deep learning techniques expect some fixed underlying distribution and assume the samples from the data are i.i.d (independently and identically distributed). With RL we learn from scalar rewards, which are often sparse and delayed, so extra care is needed to ensure we meet these assumptions.

When deep learning was used to successfully master Atari 2600 games for the first time (Mnih et al. 2013), it presented a new approach to AI for game playing and RL. Experience replay (Lin 1993) was used to overcome the issue of non i.i.d data collected from playing Atari 2600 games. Further improvements soon followed, such as Double-DQN (Van Hasselt et al. 2016), Prioritised Replay (Schaul et al. 2015), and Dueling Networks (Wang et al. 2016), all of which helped improve training dynamics and stability. Policy-gradient methods have also been introduced in recent years, the actor-critic method is an example of such a method that has been shown to be very powerful when distributed over multiple processing cores.

III SOLUTION

Various approaches to constructing AI for LOCM are proposed in the following section. We first explore model-based and tree search methods to determine the best approach of that form. We then look at some model-free methods that use NNs to make decisions about actions while playing the game. In section *IV* and *V* we will present in more detail the performance and a general discussion about each method.

A Greedy agent

The first non trivial method we propose is a greedy strategy. This is a model-based method which means we require a model of the environment that we can query, we also require that it emulates dynamics and rules of LOCM accurately. We will use the OpenAI Gym reimplementation of LOCM (Vieira et al. 2019*a*) throughout this section for playing games, evaluating strategies, observing outcomes of actions and training networks.

Algorithm 1 Greedy Strategy

```
    Input: current game state $s$, heuristic $H$
    Output: action to play $a$
  $best \leftarrow None$
  $max \leftarrow -\infty$
  for all actions do
      $s' \leftarrow$ play $action$ from $s$
      $v \leftarrow H(s')$
      if $v > max$ then
          $best \leftarrow action$
          $max \leftarrow v$
      end if
  end for
  return $best$
```

The idea behind this strategy is simple, we loop through all our available actions in the current game state; using our model we play each action from the current game state s and we observe a new game state s' for each action, we then evaluate the new game state using a heuristic state evaluation function $H \rightarrow \mathbb{R}$. We then pick the action a that maximises $H(s')$ and play it in the real game. Algorithm 1 outlines this approach.

Of course this method relies heavily on the quality of our heuristic H. The exact details of this function H are omitted but the basic idea is as follows: start from zero, look at each of your cards in play, assign it a value (based on how strong it is) and add each value to the running total, look at each of your opponents cards in play, assign it a value (in the same way) and subtract it from the running total, return the running total. There are some additional properties of the game state that our heuristic takes into account, such as player health points and additional card draws, but this is the general idea. More specifically, our heuristic H is based on the playout policy in (Kowalski & Miernik 2020). We also optimised the weights in the heuristic H with a basic genetic parameter search algorithm, which improved its win rate by 10.25% when playing against itself. This strategy gives us a nice baseline to compare against and our heuristic H will be useful for some of the policies in the more sophisticated strategies presented in this section.

B Minimax tree search agent

This method follows some of the original solutions for LOCM proposed during the first competition (Garnier 2018). The method is model-based so again we require a working model of the environment which we can query before making decisions. This was a popular approach in the first LOCM competition because constructing a model environment that was representative was a challenge in itself.

The general idea is to construct a search tree from the current game state using our working model. We iterate over all our immediate available actions and for each action we simulate 10 2-turn deep playouts. When simulating turns we follow a given policy π, that plays actions from the current player's point of view. In our implementation we choose $\pi(s)$ to be $\text{argmax}_a\,(H(s|a))$, which says play the action which maximises the heuristic value of the subsequent state, effectively π is our greedy strategy from *IIIA*. During the playouts we simulate random card draws and fill the opponent's deck and hand with random cards from the draft phase. Figure 3 illustrates the algorithm. After simulating an action a_i from the current game state our turn may not necessarily be over, so we need to complete our turn according to π - our greedy strategy. We then simulate

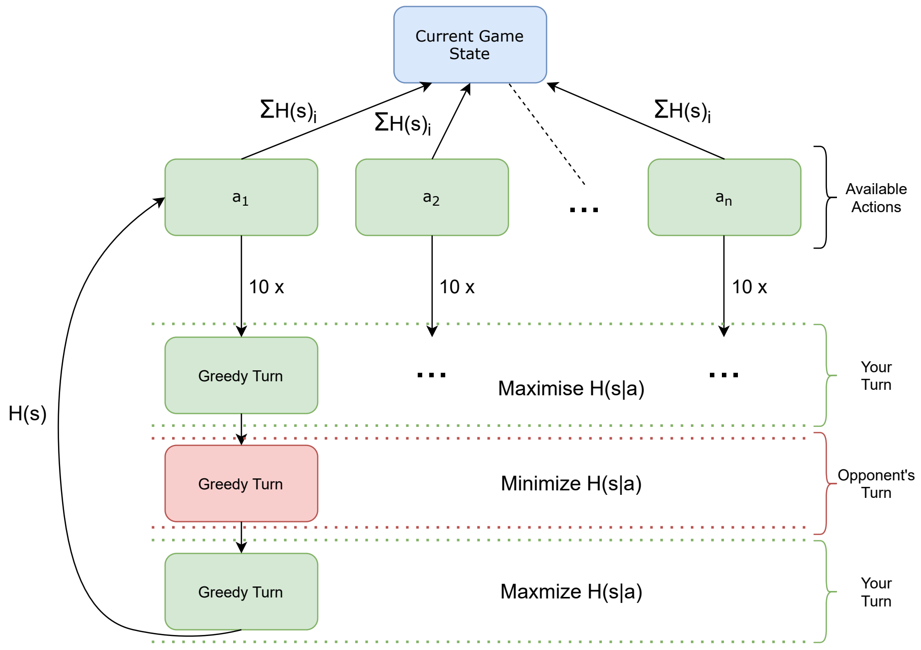

Figure 3: 2-depth minimax tree search

our opponents turn greedily, or in other words we pick actions that minimise the value returned by our heuristic H. After the playout we return the value $H(s)$ of the game state we reached, which says how good our current position is in two turns time. We sum these H values over the 10 playouts, giving us an expectation for all our actions, we then choose the action with the highest sum and play it.

This approach is very effective if our playout policy π is exactly our opponents strategy, and as such this approach is very effective against the greedy strategy presented in *IIIA*. By simulating 10 playouts, we hope overcome the imperfect information and randomness of the environment. This strategy allows us to observe the effect of performing more simultaions and looking deeper in to the tree, the details of these experiments are given in section *IV*.

C MCTS agent

This method follows the MCTS algorithm outlined in section *II* - an algorithm that has been successful in planning and strategy based games. Again, this method is a model-based approach, because we require a model to perfom playouts. We model each game state as a tree node, and its direct children correspond to the observed game state after taking exactly one action.

Initialization - We pass the current game state to the algorithm and parse it as a tree node. The node class comes with a hash function so we can hash game states and look them up in dictionaries quickly, this is key to the performance of the algorithm. We initialise 3 dictionaries, *children* which stores the direct children of each tree node, N which stores the visit counts for each tree node, and Q which stores the accumulated reward or total wins for each tree node.

Iteration - From the root node (the current game state) we construct a path and recursively select nodes according to the UCT (Upper Confidence Bound applied to Trees) policy until we reach an unexplored node. UCT is important as it facilitates the directed search of our game state tree, if we followed a random selection policy we would be looking down paths that are unlikely or not useful for gathering information about the current actions available to us. UCT works as follows, let I be the set of nodes reachable from the current node p, and select the child k such that,

$$k \in \text{argmax}_{i \in I} \left(v_i + \text{C} \times \sqrt{\frac{\ln(N_p)}{N_i}} \right)$$

where v_i is the value of a node which is defined as $\frac{Q_i}{N_i}$, or in other words the average observed reward (accumulated from playouts), and C is a constant hyperparameter (we chose C to be 1.41 for all our experiments). Note that N_i and Q_i are the visit counts and accumulated reward for i respectively. The C $\times \sqrt{\frac{\ln(N_p)}{N_i}}$ term can be interpreted as the exploration bonus, for small parent visit counts N_p we want to visit each child at some point, but for large parent counts N_p the v_i term dominates and we only want to select children we know have high reward, because these are likely actions to take in the future.

After reaching an unexplored node x, that is $x \notin$ *children*, we expand it; adding it to the tree. We then simulate a playout of the game from this node according to some simulation policy π, in the core algorithm our simulation policy π is just the random strategy. After the playout is completed we backpropogate the result of the game up the path that we traversed. We increment the visit counts of each of the nodes in our path, and update the accumulated rewards or Q values for each of the nodes in our path appropriately. In our implementation we perform as many iterations as possible within 1000ms, so the performance of our implementation can vary accross hardware. In our experiments we achieved around 750 iterations /s (hardware details are given in section *IV*). GPU speed up was explored but no signifigant increase in performance was observed, because the main bottleneck is the LOCM environment model.

Action Selection - After enough simulations we return the immediate action a with the highest average reward, or most wins on average,

$$a \in \text{argmax} \left(\frac{Q_a}{N_a} \right)$$

We then play this action in the real game.

We present the following extensions to the core MCTS algorithm, and we evaluate their impact in section *IV*.

Move Ordering - This modification specifically relates to the expansion strategy. In the core MCTS algorithm when we expand a node we add it to the *children* dictionary, its key is the hash value of the node and its value is all its direct children. As a result we can often get a high branching factor for nodes in the tree. With move ordering we consider a smarter expansion strategy based purely on game knowledge. In LOCM there are three types of actions we can take during the battle phase, ATTACK, SUMMON, and USE. Based on understanding of the game we know that there is no advantage in playing ATTACK actions before SUMMON or USE actions. So with move ordering when we expand a node we let its children be the resulting game states after all available SUMMON and USE actions. If there are no available SUMMON or USE actions then we let its children be the resulting game states after all available ATTACK actions. This signifigantly reduces the branching factor of nodes on average and removes some repeated information in the tree, as a result we can hope to perform simulations deeper into the tree.

Epsilon Greedy Simulation - This modification aims to tackle the unrepresentative simulation policy that we follow. In the core MCTS algorithm we follow a random strategy when we simulate game playouts, that is both players pick random actions until the game is complete. If we followed a more sophisticated simulation policy π our simulations will be more representative of a real playout as opposed to something random (Świechowski & Mańdziuk 2013). This modification performs greedy actions with some probability ϵ during the playout (we used $\epsilon = 0.4$ in our experiments). Again, we will use our heuristic H for this greedy play.

Urgency Simulation - This modification also deals with the simulation policy of the core MCTS algorithm. Instead of picking actions randomly during simulation or with some epsilon greedy strategy we draw them from a probability distribution. This probability distribution is constructed by softmaxing the outputs from our heuristic H for each available action. This gives us the probability of taking some action $a_i \in$ *available actions* as,

$$P(a_i) = \frac{\text{exp}(H_{a_i})}{\sum_j \text{exp}(H_{a_j})}$$

where H_{a_i} is the heuristic value of the resulting game state after taking action a_i.

Fixed Depth Early Cutoff - Early cutoff is a strategy whereby instead of simulating the game to completion, we stop early and return a heuristic value which says which player is more likely to win from this position. We will explore the effect of early cutoff because it has been shown to be effective for other CCGs (Janusz et al. 2017). The idea behind this is simulating games is costly, and if we can afford to stop early while maintaining high accuracy of the outcome of the game why shouldn't we? Fixed depth early cutoff is an early cutoff strategy that simulates a fixed number of turns before backpropogating some heuristic reward back up the tree. In our implementation we cut off the simulation after 4 turns and returned a reward of 1 if $H(s) > H(s_{\text{opponent}})$ and -1 otherwise, where s is the game state reached after the simulation and s_{opponent} is the same game state from our opponents point of view.

Probabalistic Early Cutoff - This early cutoff strategy works differently to the previous one but the idea behind it is the same. Instead of cutting off simulation after a given number of turns we cutoff simulation with some small probability p after each turn. In our implementation we set p to 0.1 and returned a reward of 1 if $H(s) > H(s_{\text{opponent}})$ and -1 otherwise, in a similar fashion to before.

Progressive UCT - In the base MCTS algorithm we follow the UCT selection policy when selecting nodes to add to our current search path. This is an effective and widely used strategy, but during the first few iterations our selection policy is seemingly random because the nodes in our search tree have small visit counts. Progressive UCT is one of the progressive strategies for MCTS algorithms for Go proposed by (Chaslot et al. 2007). This modification uses our heuristic H to direct the algorithm down promising and likely paths when the visit counts at the current node in the tree are small. We modify the UCT formula from before with a heuristic bonus,

$$k \in \text{argmax}_{i \in I} \left(v_i + \text{C} \times \sqrt{\frac{\ln(N_p)}{N_i}} + \lambda \frac{H_i}{N_i} \right)$$

where H_i is the heuristic value of the resulting game state after taking action i, and λ is a scaling parameter (we set λ to 0.01 in our experiments). The implementation for this is not necessarily so straightforward; computing H_i for each child node every time we select a new node to add to our path is very expensive, instead we initialise a new dictionary *H* which stores the heuristic value of each node in the tree. We compute and store these values when we need them and we use hashing for fast look-up.

D Q-network agent

In the subsection we propose a model-free based approach to LOCM, that learns purely off reward signals. We will use Q-networks to approximate Q values of observed game states and the training methodology we use follows closely DQN method (Mnih et al. 2013) that has been successful for a variety of games. The agent takes in as input an array of 238 floating point numbers which are scaled down by a factor of 256 before being passed through the Q-network. This input is a numerical representation of the current game state which includes information such as both player's health points, cards in play, cards in hand, and opponent's actions played last turn. Note that this representation doesn't provide a perfect MDP model of LOCM as we don't capture the total history of moves, but we can say it's good enough. The network outputs values for all 145 distinct actions that can be taken in the battle phase, this includes every combination of targets and origins for ATTACK, SUMMON and USE actions. Not all actions are available from any given state, and as such we mask the output values of illegal actions by setting them to negative infinity. We then play the action a that corresponds to the highest masked Q value output by our network,

$$a \in \text{argmax}_{a \in A} \left(m\&Q(s,a) \right)$$

where A is the set of all possible actions and the output of our Q-network $Q(s,a)$ is masked by m.

$$m\&Q(s,a) = \begin{cases} Q(s,a) & \text{if } a \in m \\ -\infty & \text{otherwise} \end{cases}$$

Training - during training we use tricks that help improve convergence, such as epsilon-greedy exploration, replay buffer, and target network which were all used in the revised version of the original Atari 2600 paper (Mnih et al. 2015):

1. Initialize the parameters θ for the Q-network $Q(s,a)$ and target network $\hat{Q}(s,a)$ with random weights, and empty the replay buffer.
2. With probability ϵ select a random available action a, otherwise select $a = \text{argmax}_a \left(m\&Q(s,a) \right)$ using our Q-network and action mask m.
3. Play the chosen action a and observe reward r and the next state s' and new action mask m'.
4. Store the tuple (s, a, m, r, s', m') in the replay buffer.
5. Sample a random mini-batch of tranisitons from the replay buffer.
6. For each transition in our mini-batch calculate the target $y = r$ if the epsiode is done, or $y = r + \gamma \max_{a' \in A} \left(m'\&\hat{Q}(s', a') \right)$
7. Update the parameters of our Q-network θ in the negative direction of the gradient with respect to the mini-batch MSE loss, defined as, $\mathcal{L} = (m\&Q(s,a) - y)^2$.
8. Every N steps copy the parameters θ from our Q-network $Q(s,a)$ to our target network $\hat{Q}(s,a)$

The exact architecture of our Q-network is similar to the plain *just_ram* architecture from (Sygnowski & Michalewski 2016). We have 2 hidden layers between the 238 inputs and 145 outputs, both with rectified linear units (ReLU). The number of neurons for each layer is an arbitrary choice; the first layer has 512 neurons and the second layer has 256 neurons. The overall training strategy follows the general idea of training the network against agents of increasing difficulty before training through self-play. The training dynamics for each session are given in section *IV*.

Note that during training we also linearly decay ϵ from ϵ_{start} to ϵ_{final} over a given number of steps so that our agent slowly starts exploiting what it has learnt. We used the following typical parameters for each training session: [$\gamma = 0.99$, learning rate = 10^{-5}, replay buffer size $= 100000$, replay warm up $= 20000$, sync N steps $= 1000$, $\epsilon_{start} = 1.0$ or 0.2, $\epsilon_{final} = 0.01$, ϵ decay steps $= 1000000$]

E Policy-gradient method

Policy-gradient methods are a different family of model-free methods that learn a probability distribution over actions for a given state, also called the policy $\pi(a|s)$, which dictates how the agent selects actions to play. These methods work for partially observable MDPs which should be a more fitting abstraction, as it captures the imperfect information in our environment. In the Q-network approach we approximated $Q(s, a)$ which informed our policy given formally by $\pi(s) = \text{argmax}_a\,(m\&Q(s, a))$. With policy-gradient methods we want to learn this policy $\pi(a|s)$ directly by calculating the policy-gradient and updating the model parameters of our network in the direction of the policy-gradient at each SGD step.

The policy-gradient method we will use is the Proximal Policy Opimization (PPO) Algorithm (Schulman et al. 2017). This method is similar to the widely used Advantage Actor-Critic (A2C) method in that we use a dueling net architecure, and compute state values $V(s)$ and advantages $A(a, s)$.

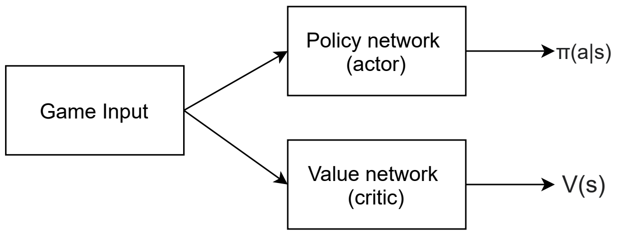

Figure 4: Dueling net architecture, with stochastic actor

PPO introduces a different objective function J_θ for approximating the policy gradient, instead of using the gradient of the logarithm of the probability of the action taken, we use the ratio between the new and old policy scaled by the advantages,

$$J_\theta = E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}A_t\right]$$

Note that during training we clip this objective function J_θ to prevent us from updating the policy too far. We calculate advantages in a slightly different way aswell,

$$A_t = \sigma_t + (\gamma\lambda)\sigma_{t+1} + (\gamma\lambda)^2\sigma_{t+2} + ... + +(\gamma\lambda)^{(T-t+1)}\sigma_{T-1}$$

where $\sigma_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ and λ is a constant hyperparameter.

Training - we follow the training procedure presented below:

1. Initialize the network parameters θ with random values.
2. Play an T-step trajectory in the environment by sampling actions from the current policy π_θ, and at each time step save the state s_t, action a_t, action mask m_t, and reward r_t.
3. For each time step t in our trajectory:

- Get the policy $\pi_\theta(s_t)$ and value $V(s_t)$ from our network.
- Calculate the advantages A_t and $R_t = A_t + V(s_t)$.

4. For N epochs sample a random mini-batch of transitions from the trajectory.
5. For each transition in our mini-batch:
   - Accumulate the value losses $\mathcal{L}_{\text{value}}$ as the MSE loss between $V(s_t)$ and R_t, defined as $\mathcal{L}_{\text{value}} = (V(s_t) - R_t)^2$.
   - Calculate the ratios between the new and old policies $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$.
   - Accumulate the policy losses $\mathcal{L}_{\text{policy}}$, given by the clipped objective function,

$$\mathcal{L}_{\text{policy}} = J_\theta^{\text{clip}} = E_t\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right].$$

6. Update the parameters of our network θ in the direction of the policy gradients (policy loss) and in the negative direction of the value gradients.
7. With SGD we minimise the loss so our loss function can be expressed as $\mathcal{L} = \mathcal{L}_{\text{value}} - \mathcal{L}_{\text{policy}}$.
8. We also add an entropy regularization loss term to encourage some additional exploration, $\mathcal{L}_H = \sum_i \pi_\theta(s_i) \log \pi_\theta(s_i)$, this gives us the loss function $\mathcal{L} = \beta\mathcal{L}_H + \mathcal{L}_{\text{value}} - \mathcal{L}_{\text{policy}}$, where β is a constant hyperparameter.

The policy network and value network have the similar network architectures. Both take in 238 inputs and have 2 hidden layers with ReLU, the first layer has 512 neurons and the second has 256 neurons. The policy network has 145 outputs which correspond to Q values for each action, we mask these outputs with an action mask m so there is no chance of picking an illegal move in the same way we did for the Q-network. We then softmax the masked Q values to get a stochastic policy $\pi_\theta(s)$ for some state s. The value network has a single scalar output which corresponds to the state value $V_\theta(s)$.

The results of the overall training strategy are detailed in section *IV*, but we will follow the same idea we did for the Q-network approach; train against opponents of increasing difficulty before training through self-play.

We used the following typical parameters during training: [$\gamma = 0.99$, learning rate $= 10^{-5}$, entropy regularization $\beta = 0.01$, trajectory length $T = 1025$, $\epsilon_{clip} = 0.2$, advantage $\lambda = 0.95$, batch size $= 64$, N epochs $= 2$]

IV RESULTS

In the following section, we present the results of various experiments, network training dynamics and a final analysis of the agents detailed in section *III*. Table 1 outlines the hardware that we used for the experiments and the final analysis. Note that Google colab was used for training, and as such no specifications for the training are given because they are not consistent.

A Minimax tree search experiments

For our minimax tree search agent we looked at the effect of increasing the number of simulations, the results are given by Table 2. We also looked at the effect of simulating 4 greedy turns as opposed to 2 greedy turns. In our experiments we played 100 games between the different minimax tree search agents and the base agent that simulates 10 2-turn deep greedy playouts. Our results conclude that searching deeper into the game state tree is important for gathering

Component	Spec.
Processor	Intel Core i7-10750H @ 2.60GHz
GPU	NVIDIA GeForce GTZ 1650 Ti
RAM	16GB
Operating System	Windows 10 Home (64-bit)

Table 1: Hardware Specifications

Win Rate v.s Minimax (2-depth) (10 sim) tree search agent		
Agent	**Win Rate (%)**	**Average Decision Time (s)**
(control) Minimax (2-depth) (10 sim)	51.0	2.455
Minimax (2-depth) (25 sim)	52.0	5.987
Minimax (4-depth) (10 sim)	66.0	4.432
Minimax (4-depth) (25 sim)	64.0	12.17

Table 2: Minimax tree search with varying depth and simulations

information and making well informed actions. We can also say that the amount of simulations we perform using the same policy is less important, and that we capture the randomness of card draws and our opponents hands with just 10 simulations.

B MCTS experiments

Table 3 presents the results from our experiments with extensions to the base MCTS algorithm. For each of the extensions described in section *III* we played 200 games between the base MCTS agent and a version of the base MCTS agent modified with the extension.

Win Rate v.s MCTS agent		
Agent	**Win Rate (%)**	**Average Simulations (/s)**
(control) MCTS	51.5	725.8
MCTS + Move Ordering	58.5	739.3
MCTS + Epsilon Greedy	17.5	61.88
MCTS + Urgency	7.5	36.61
MCTS + Fixed Depth	43.0	852.8
MCTS + P Early Cutoff	53.5	803.8
MCTS + Prog. UCT	47.0	627.1

Table 3: Evaluation of MCTS extensions

The control experiment is what we expect - when we play games between agents of equal strength we would expect to get around a 50.0% win rate. So, from our control experiment we will say that any extension that acheives above 51.5% win rate against the base MCTS agent is indeed an improvement.

Move ordering was clearly an effective extension to the base MCTS algorithm, acheiving a 58.5% win rate against the base MCTS agent.

The epsilon greedy and urgency simulation strategies were poor and suffered from the same problem; simulating games with anything other than the trivial random strategy is costly in terms of time. For both these extensions we are making calls to our heruistic H frequently, and as a result we simulated far fewer playouts per second which gives us this poor performance.

Probabalistic (P) early cutoff was our best early cutoff strategy and improved on the base MCTS agent slightly. Our early cutoff strategies helped improve our simulations per second and as a result we could gather more information. For fixed depth early cutoff perhaps we were not simulating deep enough, so our simulations were much less informative on average and this could be why we got worse perfromance.

Progressive UCT performed slightly worse, even with the dictionary speed up we simulated fewer playouts than we would normally expect and this could be the reason for the worse win rate. However, progressive UCT is incredibly hard to tune - our choice of $\lambda = 0.01$ and our heuristic bonus $\frac{H_i}{N_i}$ can have drastic effects, so these results aren't necessarily conclusive.

C Network training

Our overall training strategy for both the Q-network agent and policy-gradient method were to train against the random agent until a sufficient win rate was acheived, train against the greedy agent until a sufficient win rate was acheived, and then train through self-play. Figure 5 illustrates the full training dynamics for both methods. Unfortunately we could not get the policy-gradient

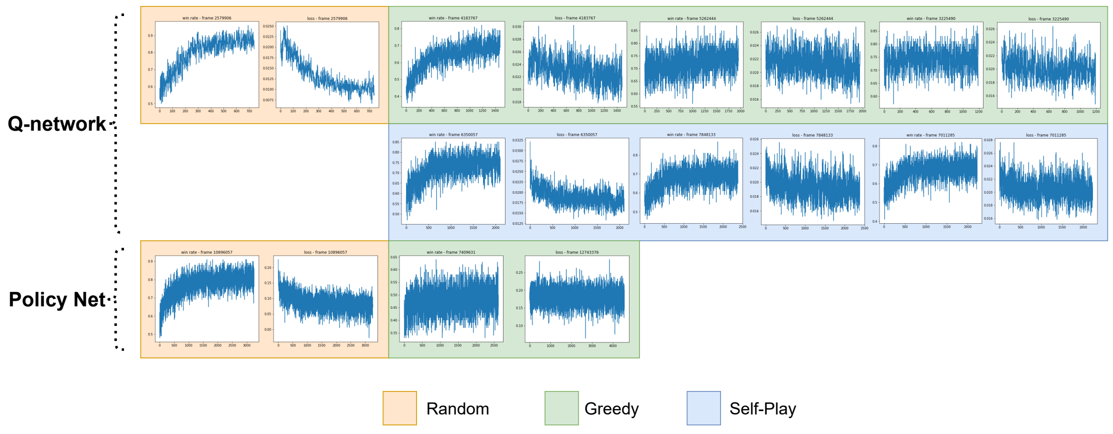

Figure 5: Training dynamics (win rate followed by loss)

method to learn to beat the greedy agent. Policy-gradient methods are on-policy which means they can only train on data gathered by playing with the current policy, as a result gathering batches to train on is particularly slow unless we have some sort of multi-agent distributed setup. Conversely, our Q-network approach is off-policy meaning we can learn off old samples, so getting batches to train on is easy (we just sample our replay buffer). As such we were able to train quicker and beat the greedy agent quite signifigantly after 3 training sessions.

The self-play training strategy we used for the Q-network was straightforward, we played 2 agents against eachother with the same Q-network policy $\pi(s) = \text{argmax}_a (m\&Q_\theta(s,a))$, except we froze the weights of one of these networks and trained the other. Put simply, we trained the Q-network to beat itself until we acheived a sufficient win rate. After we acheived this sufficient win rate we started again but this time with both agents following this new Q-network policy $\pi(s) = \text{argmax}_a (m\&Q_{\theta_{\text{new}}}(s,a))$. We did this 3 times to get our fully trained Q-network, and we observed better performance against the other agents after each of the 3 training sessions.

Note that during self-play training we interleaved experience from playing the greedy agent, with the idea of maintaining good behaviours that had already been learnt (Parisi et al. 2019). This strategy helped direct the learning process and prevent some sort of mode collapse that was observed initially.

D Final analysis

For the final analysis of the agents constructed in this paper we played a tournament where each agent plays eachother over 200 games and we record the win rates. The results of the tournament and final leaderboard are given in table 4.

Tournament					
	Random	Greedy	Minimax	Best MCTS	Q-Network
Greedy	86.5%	-	-	-	-
Minimax	95.5%	77.5%	-	-	-
Best MCTS	97.5%	88.5%	64.0%	-	-
Q-Network	93.5%	69.5%	38.0%	21.0%	-
Policy Net	76.5%	41.0%	21.5%	11.0%	19.5%

Leaderboard	
#	Agent
1	Best MCTS
2	Minimax
3	Q-Network
4	Greedy
5	Policy Net

Table 4: Tournament & final leaderboard

Clearly our best MCTS agent is very powerful - it beats all the other agents fairly signifigantly, but what is most interesting is how it performs against the minimax tree search agent. With the minimax tree search agent we don't constrain the amount of time we give it to think, and on average it takes roughly 4.5 seconds per action. With the best MCTS agent we only give it 1 second per action, so the fact that it still beats the minimax tree search agent under this constraint really shows us how powerful the algorithm is.

Our Q-network is our best model-free method which is what we expect because we trained it for longer than the policy network. It is encouraging to see that it acheives a good performance against the greedy agent, and it is clear that it has learnt to extract meaningful information from the game state represetation to make good decisions. Our policy network has clearly learnt something as it performs well against the random agent, however it is also clear that it is signifigantly weaker than the other approaches.

We will take forward the best MCTS agent and the Q-network agent. We will adapt them to the constraints of the actual LOCM implementation for evaluation against baselines and state of the art submissions used in some of the recent LOCM competitions.

Table 5 presents the performance of our agents against two baselines (Rule Based and Max Attack), and two state of the art submissions (Reinforced Greediness and Coac) that were used in the most recent LOCM compeition (Jakub Kowalski 2019). We played 1000 games between our agents and each LOCM agent to generate these results. Our agents also followed the same strategy as the coac agent for the draft phase, because now we are playing full games and not just the battle phase.

Clearly the best MCTS agent suffers from the time constraints imposed by the actual LOCM implementation, but we still get reasonable results against some of the state of the art agents. We were however able to get better results with our Q-network agent which is encouraging for future work.

Win Rate v.s. LOCM agents				
	Rule Based	Max Attack	Reinforced Greediness	Coac Agent
Best MCTS (175ms)	79.9%	80.1%	37.9%*	32.7%*
Q-Network	83.5%	85.4%	52.0%*	43.1%*

* games played in the official LOCM implementation

Table 5: Evaluation against LOCM agents

V EVALUATION

In the following section we will give a general discussion about the strengths and limitations of the different methods used in the paper, not just for CCGs but in a broader context too. We will also discuss the suitability of the overall approach used in this paper.

A Strengths

In this paper MCTS has been shown to be a powerful algorithm for dealing with MDPs, in section 4 we show that it beats all other approaches when we give it 1000ms to ”think”. We expect this because we have this strong assumption that MCTS is allowed to effectively query the future (using our environment model), which purely model-free methods cannot do. Monte Carlo methods are incredibly useful because they do not require full understanding of an environment’s dynamics, or more formally they do not require a complete explicit probability distribution of all possible state transitions. We learn optimal behaviour directly from sampled episodes, which in many cases is easier to generate than fully explicit transition probabilities. And as such our implemenetation of the core MCTS algorithm for LOCM can be readily applied to other CCGs and even wildly different problems that can be abstracted to MDPs. Through smart episodic sampling we overcome the non-determinism of card draws and the imperfect information that is our opponents hand. Monte Carlo methods have also been shown to be robust to the violation of the Markov property (Sutton & Barto 2018), so it doesn’t matter that our game state representation of LOCM doesn’t incroporate the entire history of moves.

We have also shown in this paper how useful linear approximations via NNs can be for dealing with MDPs. Figure 5 illustrates the potential for networks to learn good behaviours by extracting relevant properties from our game state representation of LOCM. Once again we do not require any prior knowledge about transition probabilities, and so we can readily apply these methods to a much wider class of problems. CCGs also exhibit challenging properties for AI agents and so it is encouraging to see that we can use Q learning to create agents that perform well against strong opponents. Our model-free methods have an inherent advantage in that they can make decisions very quickly, which is important particularly for LOCM. This also means we can think about building more powerful domain specific solutions using our fully trained Q-network as a backbone for certain policies instead of our heuristic H.

B Limitations

The main limitation of our model-based approaches is of course the strong requirement of an accurate environment model. For LOCM we are lucky to have such a model, but as we mentioned earlier this is not a given for all CCGs and other real world environments. In some cases where we have this environment model, it may not necessarily emulate the problem we are trying to solve accurately enough, or it may be particularly slow to query. One way of dealing with this issue is to learn an environment model purely from experience as is done with imagination augmented agents (Racanière et al. 2017), and sample trajectories into the future to inform actions.

Another issue with the specific implementations of our tree search agents is that we learn everything online. This means that the performance of our solution is sensitive to hardware specifications and time constraints. Table 5 illustrates this effect - our Q-network agent outperforms the best MCTS agent when we limit the time per turn to 200ms. The Alpha Go Zero (Silver et al. 2017) method incorporates the best of both worlds and trains a policy network while performing Monte Carlo simulations, to be used for the playout policy. The idea is that this makes each Monte Carlo simulation more informative, as it mimics expected play. We can say as a result, that fewer simulations are needed to come to the same conlusion about the best move, and so we can still achieve good performance even under strict time constraints.

While our Q-network approach has been shown to be effective for LOCM, this may not be the case for all problems or even all CCGs. Off-policy methods can often be unstable which is why we used tricks such as epsilon greedy exploration, and target network during training. While these tricks helped our network converge for LOCM play against average difficulty opponents, we may need to incorporate additional tricks if we want to learn from play against harder opponents - which could improve our final model.

C Approach

Our goal was to understand, implement and evaluate a variety of methods for constructing AI agents for CCGs. We achieved a valid comparison of a variety of model-based and model-free methods for a class of problems that exhibit challenging properties. We also observed the limitations and challenges associated with each of these methods. And we implemented and evaluated potential domain specific extensions to acheive a good understanding how RL can be applied and optimised for CCGs.

On-policy methods are typically more stable and should perfom better than value-based methods for games with imperfect information, as they abstract the environment in a more appropriate way. The policy-gradient method proposed in this paper gave us poor results, and perhaps if we had focused more on this framework or another specific method that has been shown to be effective for general game playing, we could have attained a more sophisticated agent and a more thorough parameter evaluation. For example the Alpha Go Zero method (Silver et al. 2017) - a framework that acheived super-human performance for Go, could have been explored in more detail and more in depth to acheive possibly excellent performance for LOCM.

VI CONCLUSIONS

With regard to our initial objectives the project was an overall success. We implemented some sophisticated model-based tree search agents and incorporated domain knowledge to improve their overall performance. We looked at two classes of model-free deep RL methods; value-based and policy-gradient methods, and demonstrated their ability to learn good behaviours in complex environments. We further modified our best performing agents so that they met certain time constraints to achieve good performance against state of the art solutions.

We showed that using move ordering can signifigantly improve the MCTS algorithm for LOCM, this result can be valid for other CCGs and potentially similar games or environments. We also constructed a fully trained Q-network that outperforms the hand crafted heuristic state evaluation function H.

More work can certainly be done on policy-gradient methods for this class of problems, different algorithms such as A2C could be used and more thorough hyperparameter tuning could help improve convergence.

References

Brown, N. & Sandholm, T. (2019), 'Superhuman ai for multiplayer poker', *Science* **365**(6456), 885–890.

Cai, X. & Wunsch, D. C. (2007), 'Computer go: A grand challenge to ai', *Challenges for Computational Intelligence* pp. 443–465.

Chaslot, G., Bakkes, S., Szita, I. & Spronck, P. (2008), Monte-carlo tree search: A new framework for game ai., *in* 'AIIDE'.

Chaslot, G., Saito, J.-T., Bouzy, B., Uiterwijk, J. & Van Den Herik, H. J. (2006), Monte-carlo strategies for computer go, *in* 'Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium', pp. 83–91.

Chaslot, G., Winands, M., Uiterwijk, J., Van Den Herik, H., Bouzy, B. & Wang, P. (2007), Progressive strategies for monte-carlo tree search, *in* 'Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)', pp. 655–661.

Dahl, G. E., Yu, D., Deng, L. & Acero, A. (2011), 'Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition', *IEEE Transactions on audio, speech, and language processing* **20**(1), 30–42.

Entertainment, B. & Entertainment, W. B. (2014), 'Hearthstone: Heroes of warcraft', *Blizzard. Ver* **5**.

Garfield, R. (1993), 'Magic: The gathering', *Wizards of the Coast* **27**, 28.

Garnier, P. (2018), 'Legends of code & magic - a codingame contest'.
URL: *https://donsetpg.github.io/blog/2018/09/02/LOCAM/*

Hoover, A. K., Togelius, J., Lee, S. & de Mesentier Silva, F. (2020), 'The many ai challenges of hearthstone', *KI-Künstliche Intelligenz* **34**(1), 33–43.

Jakub Kowalski, R. M. (2018), 'Legends of code and magic', https://github.com/CodinGame/LegendsOfCodeAndMagic.

Jakub Kowalski, R. M. (2019), 'Strategy card game ai competition', https://github.com/acatai/Strategy-Card-Game-AI-Competition.

Janusz, A., Tajmajer, T. & Świechowski, M. (2017), Helping ai to play hearthstone: Aaia'17 data mining challenge, *in* '2017 Federated Conference on Computer Science and Information Systems (FedCSIS)', IEEE, pp. 121–125.

Kocsis, L. & Szepesvári, C. (2006), Bandit based monte-carlo planning, *in* 'European conference on machine learning', Springer, pp. 282–293.

Kowalski, J. & Miernik, R. (2020), 'Evolutionary approach to collectible card game arena deckbuilding using active genes', *arXiv preprint arXiv:2001.01326* .

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012), 'Imagenet classification with deep convolutional neural networks', *Advances in neural information processing systems* **25**, 1097–1105.

Lin, L.-J. (1993), Reinforcement learning for robots using neural networks, Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & 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., Fidjeland, A. K., Ostrovski, G. et al. (2015), ‘Human-level control through deep reinforcement learning’, *nature* **518**(7540), 529–533.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C. & Wermter, S. (2019), ‘Continual lifelong learning with neural networks: A review’, *Neural Networks* **113**, 54–71.

Racanière, S., Weber, T., Reichert, D. P., Buesing, L., Guez, A., Rezende, D., Badia, A. P., Vinyals, O., Heess, N., Li, Y. et al. (2017), Imagination-augmented agents for deep reinforcement learning, *in* ‘Proceedings of the 31st International Conference on Neural Information Processing Systems’, pp. 5694–5705.

Red, C. P. (2017), ‘Gwent: the witcher card game’.

Riedmiller, M. (2005), Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method, *in* ‘European Conference on Machine Learning’, Springer, pp. 317–328.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. & Sutphen, S. (2007), ‘Checkers is solved’, *science* **317**(5844), 1518–1522.

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015), ‘Prioritized experience replay’, *arXiv preprint arXiv:1511.05952* .

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017), ‘Proximal policy optimization algorithms’, *arXiv preprint arXiv: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.

Summerville, A. & Mateas, M. (2016), Mystical tutor: A magic: The gathering design assistant via denoising sequence-to-sequence learning, *in* ‘Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment’, Vol. 12.

Sutton, R. S. & Barto, A. G. (2018), *Reinforcement Learning: An Introduction*, second edn, The MIT Press.
URL: *http://incompleteideas.net/book/the-book-2nd.html*

Świechowski, M. & Mańdziuk, J. (2013), ‘Self-adaptation of playing strategies in general game playing’, *IEEE Transactions on Computational Intelligence and AI in Games* **6**(4), 367–381.

Sygnowski, J. & Michalewski, H. (2016), Learning from the memory of atari 2600, *in* ‘Computer Games’, Springer, pp. 71–85.

Van Hasselt, H., Guez, A. & Silver, D. (2016), Deep reinforcement learning with double q-learning, *in* ‘Proceedings of the AAAI Conference on Artificial Intelligence’, Vol. 30.

Vieira, R., Chaimowicz, L. & Tavares, A. R. (2019*a*), ‘Openai gym environments for legends of code and magic’, https://github.com/ronaldosvieira/gym-locm.

Vieira, R., Chaimowicz, L. & Tavares, A. R. (2019*b*), 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’, pp. 611–614.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. & Freitas, N. (2016), Dueling network architectures for deep reinforcement learning, *in* ‘International conference on machine learning’, PMLR, pp. 1995–2003.

Watkins, C. J. & Dayan, P. (1992), ‘Q-learning’, *Machine learning* **8**(3-4), 279–292.

Yannakakis, G. N. & Togelius, J. (2018), *Artificial intelligence and games*, Vol. 2, Springer.

Zilio, F., Prates, M. & Lamb, L. (2018), Neural networks models for analyzing magic: The gathering cards, *in* ‘International Conference on Neural Information Processing’, Springer, pp. 227–239.