# Improving Hearthstone AI by Learning High-Level Rollout Policies and Bucketing Chance Node Events

Shuyi Zhang and Michael Buro
Department of Computing Science
University of Alberta, Canada
{shuyi3|mburo}@ualberta.ca

***Abstract*—Modern board, card, and video games are challenging domains for AI research due to their complex game mechanics and large state and action spaces. For instance, in Hearthstone — a popular collectible card (CC) (video) game developed by Blizzard Entertainment — two players first construct their own card decks from over 1,000 different cards and then draw and play cards to cast spells, select weapons, and combat minions and the opponent's hero. Players' turns are often comprised of multiple actions, including drawing new cards, which leads to enormous branching factors that pose a problem for state-of-the-art heuristic search methods.**

**In this paper we first present two ideas to tackle this problem, namely by reducing chance node branching factors by bucketing events with similar outcomes, and using high-level policy networks for guiding Monte Carlo Tree Search rollouts. We then apply these ideas to the game of Hearthstone and show significant improvements over a state-of-the-art AI system for this game.**

## I. Introduction

Modern computer games such as collectible card (CC) and real-time strategy (RTS) games, such as Hearthstone and StarCraft by Blizzard Entertainment, are challenging domains for AI research due to the fact that state and action spaces in such games can be quite large, game states are often only partially observable, and there is not much time available for computing good moves. Methods for reducing search complexity include hierarchical state and action abstractions ([1], [2]) and move grouping for Monte Carlo Tree Search (MCTS) [3].

In this paper we concentrate on improving AI systems for Hearthstone, a popular turn-based two-player CC game that features large action and state spaces. Turns usually consist of a series of actions which can lead to over $10,000$ different game scenarios when eventually the opponent gets his turn. Expert-level human players can regularly prune most non-optimal actions very quickly, and thus focus on only a few good action candidates, allowing them to plan ahead effectively. Inspired by this methodology we set out to study how Monte Carlo Tree Search (MCTS) can benefit from reducing branching factors, especially in chance nodes.

The recent successes of using deep neural networks (DNNs) to tackle complex decision problems such as Go and Atari 2600 video games ([4], [5], [6]) also inspired us to study how such networks can be trained to improve rollout policies in CC games.

In the remainder of the paper, we first discuss related work, motivate our main ideas, and describe them in detail. We then describe our application domain Hearthstone and state-of-the-art AI systems, which is followed by a detailed description of how we improved the performance of the Hearthstone AI system Silverfish [7] by using MCTS with chance move bucketing and pre-sampling, and learned high-level rollout policies. We conclude the paper by discussing future research.

## II. Background

In recent years there have been remarkable AI research achievements in challenging decision domains like Go, Poker, and classic video games. AlphaGo, for instance, won against one of the strongest human professionals with the help of deep networks, reinforcement learning, and parallel MCTS [4], and recently an AI system based on deep network learning and shallow counter factual regret computation running on a laptop computer won against professional no-limit Texas Hold'em players [8]. Also, deep Q-learning based programs have started outperforming human players in classic Atari 2600 video games [6]. However, modern computer strategy games, like CC or RTS games, not only have larger state and action spaces, but their complex rules and frequent chance events make the games harder to model than traditional games. Thus, it is challenging to build strong AI systems in this domain, and progress has been slow.

In modern computer games, especially strategy games, players often have to consider multiple objectives during gameplay. RTS game players, for instance, need to manage resources, technological advancement, armies, or even individual combat units in real-time. CC games feature similar challenges, albeit at a much slower pace. As solving each sub-problem alone can be computationally hard already, having to deal with multiple objectives in strategic computer games compounds the complexity. It is therefore infeasible to apply heuristic search algorithms to the original search spaces, and abstractions have to be found to cope with the enormous decision complexities. In the past few years, several ways of reducing search complexity have been studied. For instance, *Hierarchical Portfolio Search* [1] considers a set of scripted solutions for each sub-problem to generate promising low-level actions for high-level search algorithms. Likewise, *Puppet Search* [2], instead of searching in the original game's state space, traverses an abstract game tree defined by choice points given by non-deterministic scripts. Lastly, simple scripts for generating low-level moves for MCTS are used for reducing the branching factor in the CC game "Magic: The Gathering [9]."

In addition to large branching factors in decision nodes, many modern games feature chance events such as drawing cards, receiving random rewards for defeating a boss, or randomizing weapon effects. If the number of chance outcomes is high, the presence of such nodes can pose problems to heuristic search algorithms such as *ExpectiMax* search or the in-tree phase of MCTS (see below), even for methods that group nodes and aggregate successor statistics [3] or integrating sparse sampling into MCTS [10]. In the work presented here, we concentrate on improving the effectiveness of MCTS applied to games with large chance node branching factors and hierarchical actions by first reducing search complexity in the **in-tree phase** of MCTS, in which repeatedly the best child to explore will be selected until a leaf node is reached, and then improving move selection in the **rollout phase**, in which MCTS will sample action sequences according to a rollout policy until a terminal node is reached or a depth threshold is exceeded.

## III. Chance Event Bucketing and Learning High-Level Rollout Policies

In this section, we present the general problems of applying MCTS into a 2-player strategy computer games, in which the active player can execute multiple actions in a given frame of time, and approaches to solve them.

### *A. 2-Player Strategy Computer Games*

In a strategy computer games, the player can execute multiple actions of different kinds in a given frame of time. For instance, Heroes of Might and Magic is one of the most famous strategy games. CC games are sub-genre of strategy computer games where one can play cards, control minions, in a turn that has a time restriction. RTS is a special case of this kind of game since the frame of time is not discrete. Here we mostly discuss the 2-player (1 on 1) case.

In this kind of games, a turn is defined as a given frame of time, during which the active player can execute a various number of actions of different types consecutively. After the turn ends, the opponent player will similarly do a sequence of actions in his turn. Fig. 1 shows a move tree of a CC game. Player $P_1$'s turn starts after drawing a card from his deck. $P_1$ can then play multiple actions until running out of actions or choosing to end the turn. For instance, $[a_1, a_2, \text{end turn}]$ is one possible move sequence $P_1$ may choose. Chance events might also happen during turns (e.g., modeling dice rolls or drawing more cards).

### *B. Chance Event Bucketing and Pre-Sampling*

To mitigate the problem of high branching factors in chance nodes we propose to group similar chance events into buckets and reduce the number of chance events by pre-sampling subsets in each bucket when constructing search trees. Fig. 2 describes the process by applying above steps to a chance node $C$ with $S = 12$ successors. To reduce the size of the search tree we form $M = 3$ buckets containing $S/M = 4$ original chance events each. We then pre-sample $N = 2$ events from

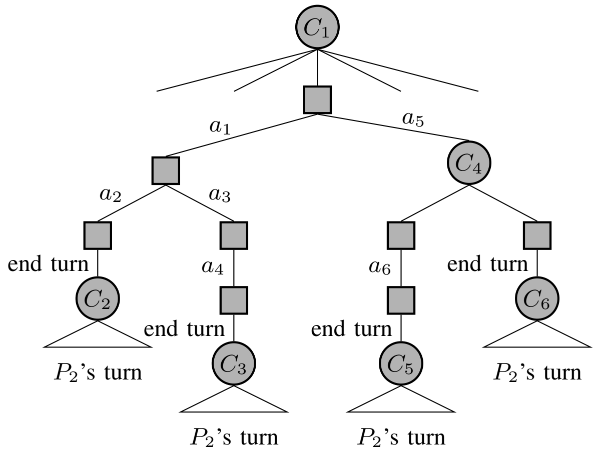

Fig. 1. A sub-tree representing a typical turn in a CC game. Player $P_1$ is to move after a chance event (e.g., drawing a card). **Squares** represent $P_1$'s decision nodes, **circles** represent chance nodes, and edges represent player moves or chance events. After $P_1$ ends the turn, $P_2$'s turn is initiated by a chance node $(C_2, C_3, C_5, C_6)$.

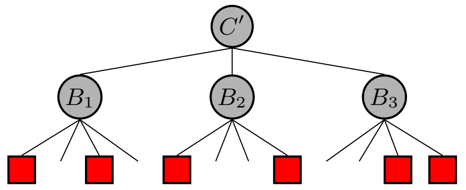

Fig. 2. Bucketing and pre-sampling applied to a chance node $C$ with 12 successors. There are $M = 3$ buckets abstracting $12/M = 4$ original chance events each. Among those $N = 2$ samples are chosen for constructing the actual search tree (red nodes).

each bucket, creating $(S/M) \cdot N = 6$ successors in total which represents a 50% node reduction.

In practical cases, the probability of each bucket is different and search agent should go to each bucket according to its probability. For the extreme case of a very skewed distribution, we can allocate more of sample budget in the larger bucket and less budget for the smaller ones. Also, the choice of $M$ and $N$ should be chosen with respect to the search space and bucket abstraction. For simple state abstraction, $M$ can be small. If the nodes in the buckets are very different, $N$ can be large. Also, there is a trade-off of more accurate sample and less search efficiency when adjusting the value of $M$ and $N$.

### *C. Learning High-Level Rollout Policies*

In many games, actions can be categorized by levels of dependencies. For example, choosing a card to play in CC games can be considered a high-level action, while selecting a target for that card can be regarded a dependent low-level action. Fig. 3 shows a typical move sequence in which high-level "play card" actions are followed by low-level "choose target" actions.

In a turn that can consist of multiple actions, the most important part is choosing high-level actions because they reflect the high-level strategy. For instance, if a player decides to attack, he will play more attacking high-level actions, and once the high-level actions are fixed, we only need to search the low-level actions that follow the high-level decisions. Fast

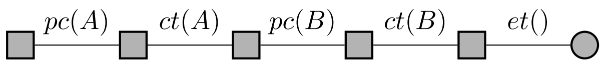

Fig. 3. A typical CC game move sequence: $pc(X)$: play card $X$, $ct(X)$: choose target for card $X$, et(): end turn.

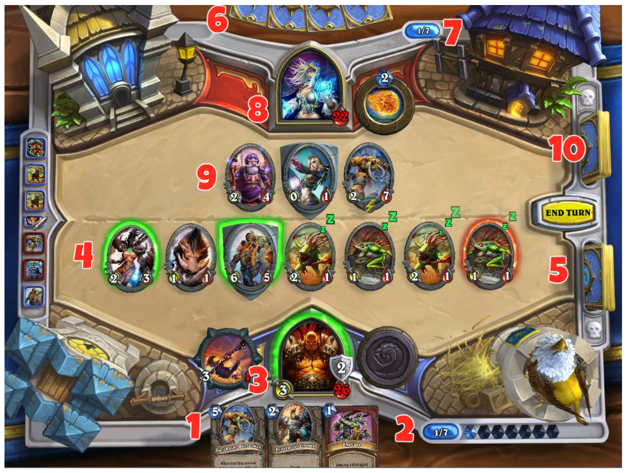

Fig. 4. Hearthstone GUI
Player 1: (1 hand) (2 mana) (3 hero) (4 minions) (5 deck)
Player 2: (6 hand) (7 mana) (8 hero) (9 minions) (10 deck)

heuristics or action scripts may be able to effectively handle this part. If this is indeed the case, we can construct fast and informed stochastic MCTS rollout policies by training a high-level policy $\pi(a, s)$ that assigns probabilities to high-level actions $a$ in states $s$, and — during the rollout phase — sample from $\pi$ and invoke low-level action scripts to generate dependent actions. This idea is exciting, because the quality of rollout policies is crucial to the performance of MCTS, but up until now, only simple policies have been trained due to speed reasons. In games with complex action sets hierarchical turn decompositions allows us to explore speed vs. quality tradeoffs when constructing rollout policies, as we will see later in Section 6.

## IV. Hearthstone

In this section, we first describe the game of Hearthstone, which is one of the most popular CC video games, to make the reader familiar with the game for which we will later present experimental results. In the second part we introduce previous work on Simulators and AI Systems for Hearthstone.

### A. Game Description

Hearthstone is a 2-player turn-based zero-sum strategy game with imperfect information. It starts with a coin flip to determine which player will go first. Players then draw their starting cards from their constructed 30 card decks. The player who goes first draws three cards and the player who goes second draws four cards and gains a special card called The Coin. Before the game starts, both players can swap out any of their starting cards for other cards from the top of their deck. The cards they swap out are then shuffled back into the deck. The game GUI is shown in Fig. 4. The key concepts in Hearthstone are:

- **Mana crystals** are needed to play cards from the hand. On the first turn, each player has one mana crystal. At the beginning of each turn, the limit of each player's mana crystals is increased by 1, and all the mana crystals are replenished.
- **Game state.** The game state has seven components: 2 heroes, the board, 2 hands, and 2 decks. The **hero** is a special type of minion that has 30 health points. A hero can only attack when equipped with a weapon and the number of attacks depends on the weapon. The game ends if and only if one hero's health value is $\leq 0$. The **board** is the battlefield where minions can attack each other. It is important to evaluate who is leading on the board because, in most games, the winning strategy is to take control of the board by trading minions and then use the minions on the board to defeat the opponent's hero. In their **hands** players hold cards that are hidden from the opponent. A player can use minion cards to capture the board or use spells to remove your opponent's minions and deal damage to the opponent's hero. Usually, having more cards in their hand allows players to handle more complex board configurations. However, just holding cards without playing them may lead to losing the control of the board. The **deck** is a collection of cards that have not been drawn yet. If a player plays all cards without ending a game, he will take fatigue damage every time he needs to draw a card from the deck. In tournament play, players have no knowledge about the opponent's deck. However, in the experiments reported later, we assume to know.
- **Cards** represent actions that a player can take by playing that card and consuming mana crystals. There are three main types: minion, spell, and weapon cards. Minion cards are placed in the board area. **Minions** have health points and attack values and can attack Heroes and other minions. Most minions have unique abilities (e.g. Minions with "Taunt" ability can protect their allies by forcing the enemy to deal with them first). **Spells** are played directly from a player's hand and have an immediate special effect. **Weapons**, like spells, are also played straight from a player's hand. However, they add a weapon to a player's arsenal allowing him to attack directly with his hero.
- **Gameplay**. Before a turn starts, the system will draw a card for the player to move. The active player can then choose which cards to play subject to available mana crystals. Some card actions will be followed by selecting a target. The player can also select a minion to attack an opponent's minion. Players usually end turns when their objective has been accomplished or there is no more action available.

### B. Hearthstone Simulators and AI Systems

The subsection describes Hearthstone simulator and AI Systems including the state-of-the-art AI player.

- **Nora** is a Hearthstone AI player that learns from random replays using a random forest classifier to choose the action one-shot [11]. It is able to defeat the random player in 90% of the games but it still loses against simple scripted players. Nora's game simulator models an early version of Hearthstone.
- **Metastone** is a feature-rich and well maintained Hearthstone simulator [12], that features a GUI and simple AI

systems, like greedy heuristic players, within the simulator, but the strength is not very high.

• **Silverfish** is a strong search-based Hearthstone AI system. It features a powerful end-of-turn state evaluation that has been tuned by human expert players, a move pruning system, an opponent modeling module that can generate commonly played actions, and a 3-turn look-head *ExpectiMax* search module that utilizes opponent modeling. Silverfish can beat rank-10 players in Hearthstone: BRM (http://hearthstone.gamepedia.com/Blackrock_Mountain) version, which is considered above the average human player strength.

For the work reported in this paper, we use Silverfish as the base-line to compare with. Silverfish has a simulator that limits the AI to 3-ply searches. To compare with Silverfish, we added features enabling Silverfish to play full games. There are some difficulties in implementing Hearthstone AI: First, there are over 1000 cards with different effects. For each card, we need to write specific scripts. Second, the game rules and mechanisms are complicated and all the cards have special effects, so the simulator needs to have multiple checkers to handle all the complex situations caused by action interactions. Even the real game itself is not bug-free. We spent considerable time on adding functions to the simulator to make it work in our experiments.

## V. Improving Silverfish by Using MCTS with Chance Event Bucketing

In this section we describe how we improved Silverfish by using MCTS and bucketing chance events as described in Section III. We start by describing our algorithm which is a variant of determinized MCTS [13], then discuss the bucketing scheme we use to reduce the large chance node branching factor in Hearthstone, and lastly present experimental results that indicate a significant performance gain.

### A. Determinized MCTS

Since Hearthstone is an imperfect information game, to improve Silverfish using search, we chose to use determinized search algorithms that yield good results in Contract Bridge and Skat [14] and "Magic: The Gathering" [15]. Specifically, we use a variant of determinized UCT (DUCT) [13], which is the UCT variant of Algorithm 1. This algorithm samples some worlds from the current information set in advance, and then in every iteration picks one and traverses down the sub-trees that fit the context of the world. If multiple worlds share one action, the statistics of that action are aggregated and used for selecting actions. When done the algorithm returns the most frequently visited move.

### B. Search Time Budget

In Hearthstone a turn consists of a sequence of actions. The best move sequence is constructed by recursively selecting the most visited child in the turn. However, if we return such a move sequence the last actions in this sequence may have low visit counts. In this case, we need to do an extra search starting from the node preceding the first rare move. In our implementation we allocate a fraction $T \cdot \beta$ of the original search time $T$ for the initial search. If there is a move in the returned move sequence with visit count $< \psi$ (a constant), then we allocate $(1-\beta) \cdot T$ and start a new search from the preceding node. Otherwise, the remaining time will be used to complete the original search.

**Algorithm 1** Determinized MCTS

```
1:  procedure Determinized MCTS($I, d$)
2:      $worlds \leftarrow$ Sample($I, numWorlds$)
3:      while search budget not exhausted do
4:          for $n$ in $worlds$ do
5:              $e \leftarrow$ Traverse($n$)
6:              $l \leftarrow$ Expand($e$)
7:              $r \leftarrow$ Rollout($l, d$)
8:              PropagateUp($l, r$)
9:          end for
10:     end while
11:     return BestRootMove()
12: end procedure
13:
14: procedure Traverse($n$)
15:     while $n$ is not leaf do
16:         if $n$ is chance node then
17:             $n \leftarrow$ SampleSuccessor($n$)
18:         else
19:             $n \leftarrow$ SelectChildDependingOnCompatibleTrees($n$)
20:         end if
21:     end while
22:     return $n$
23: end procedure
24:
25: procedure Rollout($n, d$)
26:     $s \leftarrow 0$
27:     while $n$ not terminal and $s < d$ do
28:         $s \leftarrow s + 1$
29:         $n \leftarrow$ Apply($n$, RolloutPolicy($n$))
30:     end while
31:     return Eval($n$)
32: end procedure
```

### C. Empirical Chance Event Bucketing

The number of possible turn outcomes in Hearthstone is enormous due to multiple actions played in a row and card-drawing chance events. To mitigate this combinatorial explosion we apply chance event bucketing as follows. In Hearthstone, cards with similar mana cost usually have similar strengths. We can therefore categorize cards by their mana cost to form $M$ buckets. The actual bucket choice depends on the card deck we are using and can be optimized empirically. In the experiments reported later we used buckets shown in Table I. For determining the number of pre-samples $N$ we experimented with various settings depending on the number of cards to be drawn. The most effective choice was $N = 2$ when one card is drawn, and $N = 1$ if more cards are drawn.

### D. Utilizing Silverfish Functionality

Our DUCT search module utilizes Silverfish's complex rule-based evaluation function tuned by expert-level human players. This evaluation function only evaluates the end-of-turn game state by taking the hero, minion, and hand features, the number of cards drawn, and penalty of actions executed during the

TABLE I
CARD BUCKETING BY DECK AND MANA COST IN HEARTHSTONE

| Deck | Buckets |
|---|---|
| Mech Mage | [1] [2] [3] [4,5] [6..10] |
| Hand Warlock | [1,2,3] [4] [5] [6] [7..10] |
| Face Hunter | [1] [2] [3..10] |

TABLE II
WIN % (STDERR) VS. SILVERFISH

| Mirror Match | DUCT-Sf | DUCT-Sf+CNB |
|---|---|---|
| Mech Mage | 66.5 (3.3) | 76.0 (3.0) |
| Hand Warlock | 54.0 (3.5) | 71.5 (3.1) |
| Face Hunter | 60.0 (3.5) | 69.5 (3.2) |
| Combined | 60.1 (2.0) | 72.3 (1.8) |

turn into account. We keep this function in DUCT because it is fast (since it's rule-based) and sufficient to make simple evaluations. We also use parts of the rule-based pruning code in Silverfish's move generator that can prune bad moves, such as dealing damage to our hero.

## E. Experiments

To evaluate the effect of adding DUCT and chance node bucketing (CNB) to Silverfish we ran two experiments on an Intel i7-4710HQ CPU 3.5 GHz Windows 8.1 computer with 16 GB RAM. In the first experiment we let DUCT-Sf without CNB play 3 mirror matches, in which both players use the same deck (either Mech Mage, Handlock, or Face Hunter), against the original Silverfish player, allowing 5 seconds thinking time per move and using DUCT parameters $d = 5$, $numWorlds = 10$, UCT's optimized exploration constant $c = 0.7$, and time management parameters $\beta = 2/3$ and $\psi = 50$. The results shown in Table II indicate that the performance of DUCT-Sf is superior to Silverfish's in all 3 matches. In the second experiment we let DUCT-Sf with CNB play against Silverfish. The results, listed in the last column of Table II show an even greater playing strength gain.

## VI. LEARNING HIGH-LEVEL ROLLOUT POLICIES IN HEARTHSTONE

In this section we first describe the neural networks that we trained for making Hearthstone card play decisions in the MCTS rollout phase, and then present experimental results.

## A. Card-Play Policy Networks

A card-play policy network for Hearthstone maps a game state $n$ to a card probability vector. The probabilities indicate how probable it is for card $c_i$ to be in the turn card set

$$TCS(n) := \{c \mid c \text{ is played in turn starting with } n\ \}$$

Our goal is to train policy networks to mimic turn card sets computed by good Hearthstone players, which then can be used as high-level rollout policies in DUCT.

## B. State Features

Because Hearthstone's state description is rather complex we chose to construct an intermediate feature layer that encapsulates the most important aspects of states. Our state feature set consists of three groups:

- **Global features** are represented as a 1D vector one-hot encoding mana available until turn end, the opponent's available mana on the next turn, the Hero's health points (HP) (0-4 for each player, for a total of 25 different values), whether the active player is the starting player of the game, and whether the total attack of our minions is greater than the total health points of the opponent's minions.
- **Hand features:** we use a 2D vector $V_h[x][y]$ to one-hot encode hand features. Each distinct card, which appears in the decks, is given an index. The $j$th column ($y = j$) encodes features related to the card with index $= j$ ($C_j$). Let $NC_a(C_j)$ stand for the number of copies of $C_j$ in the active player ($P_a$)'s hand, and $NC_o(C_j)$ represent the same feature of the opponent ($P_o$)'s hand. The 1st row ($x$ = 0) indicates whether $(NC_a(C_j), NC_o(C_j))$ = $(0, 2)$ for each card, the 2nd row ($x$ = 0) encodes $(NC_a(C_j), NC_o(C_j))$ = $(0, 1)$ and so on. There are 9 different possible value pairs ((0,2), (0,1), (1,2), (0,0), (1,1), (2,2), (2,1), (1,0), (2,0)) of $(NC_a(C_j), NC_o(C_j))$. For instance, if both $P_a$ and $P_o$ has 1 copy of $C_5$ in hand, then $V_h[4][5]$ = 1. We use next 4 rows to encode the card playability of both players since there are (0,1), (1,1), (0,0), (1,0), 4 possible value pairs of $(P_a, P_o)$. The last 3 rows encode whether there is a following card-play if $C_j$ card is played: $\{x$ = 13: no following card-play, $x$ = 14: a low-mana card-play, $x$ = 15: high-mana card-play$\}$.
- **Board features** are one-hot represented as a 3D vector $V_b[x][y][z]$. Each minion on the board has a 2D index $(y, z)$ to abstract its status, where $z$ represents the index of the card that summons the minion, $y$ represents state of a minion's health points: ranging from 0 to 5. The mapping from $y$ to health points is $\{y$ = 5: 0-1, $y$ = 5: 1-2, $y$ = 5: 3-4, $y$ = 5: 5-6, $y$ = 5: 6+$\}$. For example, a minion $M$ with a card index = 3 and Health points = 5 will have the its index = $(3, 3)$. The first 9 layers encode the different states of the numbers of two players' minion, which is encoded the same way as the hand feature. The next 9 layers encode the 3 levels of the specialty of a minion. Lv.0: no special effects, Lv.1: aura minion and battle-cry minions, and Lv.2: legend minions.

Table III summarizes the features we use in our experiments. We also tried some hand-crafted features but they didn't show merit, and we skipped some features like a minion's buff and debuff (power-ups or power-downs) to keep the model simple.

## C. Training Data

To generate data for training our networks we let two DUCT-Sf+CNB players play three different mirror matches (using Mech Mage, Handlock, Face Hunter decks), each consisting of 27,000 open-hand games using 10,000 rollouts per move. Because drawing new cards in each turn randomizes states in Hearthstone we didn't feel the need for implementing

TABLE III
FEATURES FROM THE VIEW OF THE PLAYER TO MOVE

| Feature(Modal) | Value Range | #CNN Planes |
|---|---|---|
| Max Mana (Global) | 1-10 | — |
| Heroes' Hp (Global) | 4 states | — |
| If active player if $P_1$ (Global) | 0-1 | — |
| Total attack $\geq$ enemy's board Hp (Global) | 0-1 | — |
| Having each card (Hand) | 9 states | 9 |
| Each card playable (Hand) | 4 states | 4 |
| Next card after a cardplay (Hand) | 3 states | 3 |
| Having each minion (Board) | 9 states | 9 |
| Each minion's specialty (Board) | 9 states | 9 |

explicit state/action exploration, but we may revisit this issue in future work.

The training target is the turn card set $TCS(n)$ for state $n$. For each triple $(n, TCS(n), n_{end})$ in the stored data set, where $n$ is an intermediate game state and $n_{end}$ is the turn end state reached after $n$, we have one training sample $(n, TCS(n))$. In fact, all intermediate state-TCS pairs can also be used as training samples. In total, we used about 4M samples.

### *D. Network Architecture and Training*

For approximating high-level card play policies we employ two network topologies:

1. **"CNN+Merge."** In this network type the three state feature groups are separated at the beginning. The global features receive the input from the hand feature, then trained with fully connected network layers with the Leaky ReLU activation function ($\alpha = 0.2$). The board group is a 2D vector, thus convolution layers to capture the pattern of the input of the board for predicting the cards to be played. For instance, a pattern in Fig. 5 indicates that the active player is very likely to play spell cards dealing damage to opponent's archmage since the archmage cannot be killed by only mech yeti's attack. This method works successfully in Poker [16]. We tried to use 96 3x5 filters with followed by a 2x2 max pooling layer, then followed by 3 to 6 convolution layers with 96 3x3 filters. For the hand features, we use 4 to 6 1D convolution layers with 96 filters of the size 3. There is a merged model that concatenate the flattened output of the three groups and followed by fully connected layers with 0.5 drop-outs. The last layer is a 20 to 23 ways (depends on the match-ups) output with the binary cross-entropy to give the probability of the each card to be played.

2. **"DNN+Merge."** The network type also receives the inputs from the 3 feature groups, but the entire input is flattened into one long vector for each group. Each group vector is then followed by fully connected layers of leaky ReLU units ($\alpha = 0.2$). Similarly to the CNN+Merge type, the output of each group is fed into one merged layer and then followed by a fully connected layer with using 0.5 drop-outs. The output is the same as in the CNN+Merge networks.

When training both network types we used Xavier uniform parameter initialization [17]. We train several different models using similar settings. The largest one is a CNN+Merge network with 6 convolution layers has with 1.75M parameters; the smallest one is the DNN+Merge network that has only 140k parameters.

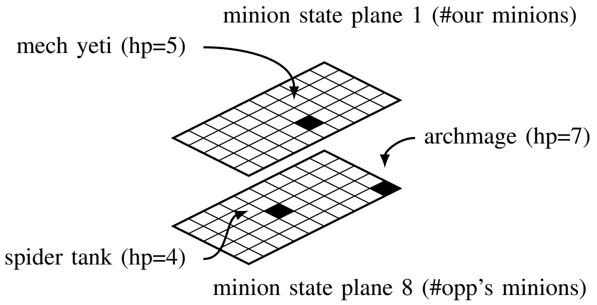
minion state plane 1 (#our minions)
mech yeti (hp=5)
archmage (hp=7)
spider tank (hp=4)
minion state plane 8 (#opp's minions)

Fig. 5. Board feature pattern example: black squares encode 1s and white squares 0s. Plane 1 encodes whether there is 1 minion on my board, while plane 8 encodes whether there is 1 minion on the opponent's board. This example indicates that we have 1 mech yeti while the opponent has 1 spider tank and 1 archmage on the board.

To tailor networks to different deck choices and maximum mana values we train them on data gathered from 3 mirror matches which we divided into 10 different sets with different initial maximum available mana values. For training we use the adaptive moment estimation (ADAM) with $\alpha = 10^{-3}$, decay $\sqrt{t/3}, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. The mini-batch size was 200, and for one model, it approximately took between 500 and 1,000 episodes for the training process to converge.

### *E. Experiment Setup*

We trained and tested our neural networks with an NVIDIA GeForce GTX 860M graphics card with 4GB RAM using CUDA 7.5 and cuDNN4. The Hearthstone game simulator is written in C# and the networks are executed using Keras 1.1.2 [18]. For transmitting data between C# and Python 2.7.12 we used PythonNet [19] which introduced negligible delays.

### *F. High-Level Move Prediction Accuracy*

This section we compare the predicted card selection of our learned high-level policy networks with the following move selectors:

• **Silverfish:** this is regular Silverfish with 3-ply search depth and 1 second search time limit.

• **Greedy:** This action selector uses cost-effect action evaluation heuristic $H(a)$, which we adapted from Silverfish's heuristics, is defined as:

$$H(a) = \frac{value(a)}{cost(a)}, \quad \text{where} \tag{1}$$

$$value(a) = \sum_{m \in M_p} G(m, a) + \sum_{m \in M_o} L(m, a) \tag{2}$$

$$cost(a) = (\sum_{m \in M_p} L(m, a)) + a.ManaCost + 1 \tag{3}$$

$$L(m, a) = HpLoss(m, a) \cdot (m.ManaCost + 1)/m.MaxHp \tag{4}$$

$$G(m, a) = HpGain(m, a) \cdot (m.ManaCost + 1)/m.MaxHp \tag{5}$$

TABLE IV
HIGH-LEVEL POLICY PREDICTION

| Mana: | 1-2 | 3-4 | 5-7 | 7-10 |
|---|---|---|---|---|
| CNN+Merge (1.75m params) | 91.9% | 74.9% | 76.6% | 79.3% |
| CNN+Merge (290k params) | 91.5% | 71.7% | 75.4% | 77.8% |
| DNN+Merge (230K params) | 89.9% | 66.6% | 69.2% | 73.2% |
| Silverfish | 86.7% | 66.2% | 67.0% | 73.8% |
| Greedy | 82.7% | 50.3 | 50.4% | 55.5% |

TABLE V
WIN RATE OF CNN + GREEDY

| Opponent | % Win Rate | % Std. Deviation |
|---|---|---|
| Random | 99.7 | 0.3 |
| Greedy | 71.3 | 2.6 |
| Silverfish 1-turn | 54.6 | 2.8 |
| Silverfish 3-turns | 18.8 | 3.3 |

with $a$ being the action to be evaluated, $M_p, M_o$ representing the player to move's and the opponent's minion set, respectively, and $HpLoss(m, a)$ and $HpGain(m, a)$ denoting the loss and gain of minion $m$'s health points when executing action $a$. $H(a)$ is a local heuristic that uses mana cost as scale for unifying the evaluation of gains and losses considering card-minion interactions. $H(a)$ is not very accurate for comparing actions from different levels, but it is fairly good for comparing actions with the same precondition, such as finding the best target for a given card. The greedy action selector chooses the actions $a$ with the highest $H(a)$ values in the current turn.

For estimating the card selection quality we generated a total of around 1000 games the same way as the training data. We then picked ten states from each game with 1 to 10 available mana crystals, respectively. The accuracy metric we used is strict $TCS$ equality, i.e., a card set prediction is accurate if $TCS_{pred}(n) = TCS(n)$.

The results are presented in Table IV. They show that except for the beginning of the game, the trained networks are consistently better than Silverfish and Greedy at predicting turn card sets generated by high-level open-hand play, and that large networks are slightly better than the smaller networks. It is also interesting that near the end of the game the accuracy of all card selectors rises again. In the Face Hunter and Mech Mage games this may be caused by players running out of cards towards the end of the game which makes it makes easier to predict cards. The results also suggest that our CNN outperforms the DNN when using a similar number of parameters. However, the smaller DNN network only takes 60 microseconds to do one mini-batch evaluation, whereas the CNN takes 10 times longer.

## G. Playing Games

We combined our deeper one-shot card-play policy networks with the low-level Greedy action chooser with the cost-effect heuristic (Eq. 1). We use the card-play policy networks to get the card to play by $\arg\max P_{turn}(c_i|s)$ and choose the best-valued action that follows our policy. In this experiment, we feed the open-handed states to the networks, and there is no search in this simple algorithm.

We play against different opponents including random player, Greedy player with H(a) heuristic and Silverfish with 1-play and 3-turn look-ahead search with 3 seconds of thinking time, the win rates is shown in Table V.

The weakness of Greedy action chooser is that it ignores the management of mana and the inference of opponent's hand. Card-play policy networks are complementary to such high-level decisions. However, it still cannot beat the search-based 3-turn look ahead Silverfish as expected.

## H. Incorporating Card-Play Networks into DUCT

To make use of high- and low-level rollout policies in DUCT we replace the original Rollout function with Algorithm 2. This algorithm is tailored for games with multi-action turns and uses policies $\pi_l$ and $\pi_h$ to choose high- and low-level actions, respectively. It will execute multiple turns until either the turn limit or a terminal state is reached. If both high-level and low-level actions are available, it randomly selects either type and invokes the respective policy to generate an action. Otherwise, if an action is still available it uses the respective policy to generate one. Finally, the end-turn action is generated if no other actions are available. In the case of Hearthstone high-level policy $\pi_h(n)$ selects a card and low-level policy $\pi_l(n)$ then selects a suitable target.

In our implementation we apply the SoftMax function to the less accurate but fast DNN outputs to define $\pi_h$ based on card evaluations, and the fast action evaluator $H$ to form $\pi_l$ based on heuristic target action evaluations.

To reduce data transmission overhead when communicating between C# and Keras's Python code, we allocate a Numpy array and just send the indices of the entries to be filled. We also take advantage of the fact that the high-level policy

**Algorithm 2** Rollout with Multi-Level Policy

```
1: // $n$: current state, $d$: turn limit
2: // $\pi_h$: high-level policy, $\pi_l$: low-level policy
3: procedure Rollout($n, d$)
4:     $t \leftarrow 0$
5:     while $n$ not terminal and $t < d$ do
6:         if $n$ is chance node then
7:             $a \leftarrow$ SampleSuccessor($n$)
8:         else if high- and low-level actions available then
9:             if $Random(0,1) > 0.5$ then
10:                $a \leftarrow \pi_h(n)$
11:            else
12:                $a \leftarrow \pi_l(n)$
13:            end if
14:        else if high-level actions available then
15:            $a \leftarrow \pi_h(n)$
16:        else if low-level actions available then
17:            $a \leftarrow \pi_l(n)$
18:        else
19:            $a \leftarrow et$                                  ▷ end turn
20:            $t \leftarrow t + 1$
21:        end if
22:        $n \leftarrow$ Apply($n, a$)
23:    end while
24:    return Eval($n$)
25: end procedure
```

TABLE VI
DUCT-SF+CNB+HLR WIN RATE AGAINST DUCT-SF

| Mirror Match | % Win Rate | % Std. Deviation |
|---|---|---|
| Mech Mage | 53.4 | 2.2 |
| Hand Warlock | 62.6 | 2.1 |
| Face Hunter | 55.6 | 2.2 |
| Combined | 57.2 | 1.3 |

network only has to be evaluated once when the turn begins. The multi-level policy rollout function we implemented is 5 to 10 times slower than regular rollouts, but 10 times faster than the bigger CNNs. To test the effect of the high-level rollout (HLR) policy, we incorporated it into the strongest search-based AI without neural networks, namely DUCT with Silverfish's evaluation function and chance node bucketing (DUCT-Sf+CNB), and ran 500 games against DUCT-Sf+CNB for each mirror match, allowing 10 seconds thinking time per move and using $d = 5$, $numWorlds = 10$, $c = 0.7$, $\psi = 50$ and $\beta = 2/3$. One mirror match took one day to run on a single computer. The results presented in Table VI indicate a significant improvement over the already strong DUCT-Sf+CNB player.

## VII. Conclusions and Future Work

In this paper we have presented two improvements of MCTS applied to Hearthstone, and potentially other CC games. We use bucketing and pre-sampling to deal with the issue of large branching factors caused by chance nodes. By using the optimized DUCT algorithm and Silverfish's evaluation function, our new search agent DUCT-Sf+CNB defeats the original Silverfish by 72% of the time.

We then define a high-level policy for CC games and present features for evaluating Hearthstone states that we feed into different neural networks trained from game data. Lastly, we apply the trained high-level networks in conjunction with low-level action heuristics to perform stochastic MCTS rollouts. Our experiments show that the new AI system is even stronger than DUCT-Sf+CNB.

This paper combines improved MCTS's in-tree policies with learned rollout policies. Both parts can potentially be improved further. For instance, machine learning could be applied to the bucketing and sampling strategies instead of relying on manual tuning. Moreover, rollout policies can possibly be improved by learning low-level action policies and applying reinforcement learning. Also, our policy networks rely on perfect information state. There are possible future works can be done by using recurrent networks that receive partially observed state combined with the move history as the input.

There are also newer Hearthstone simulators like Metastone [12] which are updated frequently to reflect changes in the original game. We are considering to use this simulator for future research because it frees us from tedious implementation issues. Along with other successes of using search and deep learning techniques in modern video games we are optimistic that we will see stronger AI systems for complex CC games soon.

## References

[1] D. Churchill and M. Buro, "Hierarchical portfolio search: Prismata's robust AI architecture for games with large search spaces," in *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*, 2015.

[2] N. A. Barriga, M. Stanescu, and M. Buro, "Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games," in *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[3] N. Jouandeau and T. Cazenave, "Monte Carlo tree reductions for stochastic games," in *Technologies and Applications of Artificial Intelligence*. Springer, 2014, pp. 228–238.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[5] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline Monte Carlo tree search planning," in *Advances in neural information processing systems*, 2014, pp. 3338–3346.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] noHero123, "Silverfish," https://github.com/noHero123/silverfish, 2015.

[8] M. Moravčík *et al.*, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, 2017. [Online]. Available: http://dx.doi.org/10.1126/science.aam6960

[9] C. D. Ward and P. I. Cowling, "Monte Carlo search applied to card selection in Magic: The Gathering," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009, pp. 9–16.

[10] M. Lanctot, A. Saffidine, J. Veness, C. Archibald, and M. H. M. Winands, "Monte carlo *-minimax search," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, pp. 580–586. [Online]. Available: http://dl.acm.org/citation.cfm?id=2540128.2540213

[11] D. Taralla, Z. Qiu, A. Sutera, R. Fonteneau, and D. Ernst, "Decision making from confidence measurement on the reward growth using supervised learning: A study intended for large-scale video games," in *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016)-Volume 2*, 2016, pp. 264–271.

[12] demilich1, "Metastone," https://github.com/demilich1/metastone, 2017.

[13] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information set Monte Carlo tree search." *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 4, no. 2, pp. 120–143, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/tciaig/tciaig4.html#CowlingPW12

[14] T. Furtak and M. Buro, "Recursive Monte Carlo search for imperfect information games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[15] 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, pp. 241–257, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/tciaig/tciaig4.html#CowlingWP12

[16] N. Yakovenko, L. Cao, C. Raffel, and J. Fan, "Poker-CNN: a pattern learning strategy for making draws and bets in poker games," *arXiv preprint arXiv:1509.06731*, 2015.

[17] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[18] F. Chollet, "Keras," https://github.com/fchollet/keras, 2017.

[19] D. Anthoff, "PythonNet," https://github.com/pythonnet/pythonnet, 2017.