Deep Reinforcement Learning in Continuous Action Spaces: a Case Study in the Game of Simulated Curling

Kyowoon Lee [*1] **Sol-A Kim** [*1] **Jaesik Choi** [1] **Seong-Whan Lee** [2]

Abstract

Many real-world applications of reinforcement learning require an agent to select optimal actions from continuous spaces. Recently, deep neural networks have successfully been applied to games with discrete actions spaces. However, deep neural networks for discrete actions are not suitable for devising strategies for games where a very small change in an action can dramatically affect the outcome. In this paper, we present a new self-play reinforcement learning framework which equips a continuous search algorithm which enables to search in continuous action spaces with a kernel regression method. Without any hand-crafted features, our network is trained by supervised learning followed by self-play reinforcement learning with a high-fidelity simulator for the Olympic sport of curling. The program trained under our framework outperforms existing programs equipped with several hand-crafted features and won an international digital curling competition.

1. Introduction

Learning good strategies from large continuous action spaces is important for many real-world problems including learning robotic manipulations and playing games with physical objects. In particular, when an autonomous agent interacts with physical objects, it is often necessary to handle large continuous action spaces.

Reinforcement learning methods have been extensively applied to build intelligent agents that can play games such as chess (Campbell et al., 2002), checkers (Schaeffer et al., 1992), and othello (Buro, 1999). Recently, deep convolutional neural networks (CNNs) (LeCun & Bengio, 1998) have achieved super-human performance in deterministic games with perfect information, such as Atari games (Mnih et al., 2015) and Go (Silver et al., 2016; 2017). In the latter game, board positions are passed through the convolutional layers as a 19-by-19-square image. These CNNs effectively reduce the depth and breadth of the search tree by evaluating the positions using a value network and by sampling actions using a policy network. However, in a continuous action space, the space needs to be discretized. Deterministic discretization would cause a strong bias in the policy evaluation and the policy improvement. Thus, such deep CNNs for large, non-convex continuous action spaces are not directly applicable.

To solve this issue, we conduct a policy search with an efficient stochastic continuous action search on top of policy samples generated from a deep CNN. Our deep CNN still discretizes the state space and the action space. However, in the stochastic continuous action search, we lift the restriction of the deterministic discretization and conduct a local search procedure in a physical simulator with continuous action samples. In this way, the benefits of both deep neural networks (i.e., learning the global structure) and physical simulators (i.e., finding precise continuous actions) can be realized.

More specifically, we design a deep CNN called the policy-value network, which gives the probability distribution of actions and expected reward given an input state. The policy-value network is jointly trained to find an optimal policy and to estimate the reward given an input instance. During the supervised training, the policy subnetwork is directly learned from the moves of a reference program in each simulated run of games. The value subnetwork is learned using d-depth simulation and the bootstrapping of the prediction to handle a high variance of a reward obtained from a sequence of stochastic moves. The network is then trained further from the games of self-play using kernel regression to precisely handle continuous spaces and actions. This process allows actions in the continuous domain to be explored and adjusts the policy and the value in consideration of uncertainty of the execution.

[*]Equal contribution [1]Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology, Ulsan, Republic of Korea [2]Department of Brain and Cognitive Engineering, Korea University, Seoul, Republic of Korea. Correspondence to: Jaesik Choi <jaesik@unist.ac.kr>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

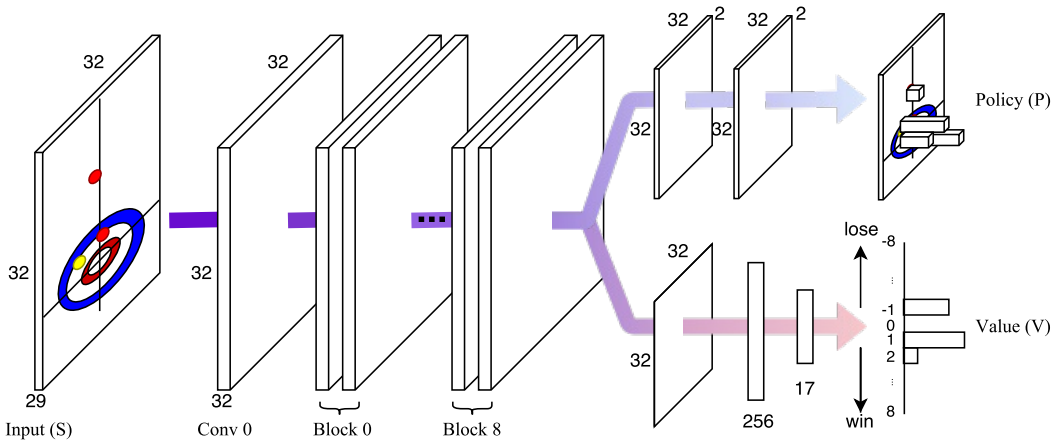

Figure 1. The architecture of our policy-value network. As input, a feature map (Table 2 in the supplementary material) is provided from the state information. During the convolutional operations, the layers' width and height are fixed at 32x32 (the discretized position of the stones) without pooling. The details of the layer information are provided described in Figure 2. We train the policy and the value functions in a unified network. The output of the policy head is the probability distribution of each action. The output of the value head is the probability distribution of the final scores [-8,8].

We verify our framework with the sport of curling. Curling, often called *chess on ice*, has been viewed as the most intellectually challenging Olympic sport due to its large action space and complicated strategies. Typically, curling players put a stone in a large area of about 5m by 30m, with precise interactions which are typically less than 10 cm among stones. When discretized, the play area is divided into a 50x300 grid. Asymmetric uncertainty is added to the final location to which a stone is delivered.

The program trained under our framework outperforms state-of-the-art digital curling programs, AyumuGAT'17 (Ohto & Tanaka, 2017) and Jiritsukun'17 (Yamamoto et al., 2015). Our program also won in the Game AI Tournaments (GAT-2018) (Ito).

2. Related Work

In the game of go, *AlphaGo Lee*, the successor version of *AlphaGo Fan* (Silver et al., 2016), defeated Lee Sedol, the winner of 18 international titles. Although Go has a finite, discrete action space, its depth of the play creates complex branches. Based on the moves of human experts, two neural networks in *AlphaGo Lee* are trained for the policy and value functions. *AlphaGo Lee* uses a Monte Carlo tree search (MCTS) for policy improvement.

AlphaGo Zero (Silver et al., 2017), which is trained via self-play without any hand-crafted knowledge, has demonstrated a significant improvement in performance. *AlphaGo Zero* is expected to win more than 99.999% of games against *AlphaGo Lee*.[1] *AlphaGo Zero* uses a unified neural network for the policy and value networks to train the networks faster.

In the domain of curling, several algorithms have been proposed. As a way of dealing with continuous action space, game tree search methods (Yamamoto et al., 2015) have discretized continuous action space. The evaluation functions are designed based on the domain knowledge and rules of the game. With considering given execution uncertainty, the action value is calculated as the average of the neighboring values.

A MCTS method called *KR-UCT* has been successfully applied to continuous action space (Yee et al., 2016). KR-UCT exhibits effective selection and expansion of nodes using neighborhood information by estimating rewards with kernel regression (KR) and kernel density estimation (KDE) in continuous action spaces. Given an action, the upper confidence bound (Lai & Robbins, 1985) of the reward is estimated based on the values nearby. KR-UCT can be regarded as a specialized exploration of pseudo-count based approaches (Bellemare et al., 2016).

To handle continuous action space in the bandit problem, several algorithms have been proposed. For example, hierarchical optimistic optimization (HOO) (Bubeck et al., 2008) starts by creating a cover tree and recursively divides the action space into smaller candidate ranges at each depth. A node in the cover tree is considered as arms of the sequential bandit problem. The most promising node is exploited to create estimates of finer granularity, and regions which have not been sampled sufficiently are explored further.

An analysis of the dynamics of curling is important to build an accurate digital curling program. For example, the fric-

[1] Their difference in elo rating is greater than 2,000.

tion coefficients between the curling stones and the ice curling sheet have been analyzed (Lozowski et al., 2015), while pebbles, the small frozen droplets of water across the play sheet, have also been taken into account (Maeno, 2014). Unfortunately, modeling the changes in friction on the ice surface is not yet possible. Thus, in general, digital Curling simulators assume a fixed friction coefficient with noise generated from a predefined function (Ito & Kitasei, 2015; Yee et al., 2016; Ahmad et al., 2016).

The physical behavior of the stones has been modeled using physics simulation engines such as Box2D (Parberry, 2013), Unity3D (Jackson, 2015) and Chipmunk 2D. Important parameters including friction coefficients and noise generation functions are trained from games between professional players (Yee et al., 2016; Ito & Kitasei, 2015; Heo & Kim, 2013). In this paper, we use the same parameters used in a digital curling competition (Ito & Kitasei, 2015).

3. Background

In this section, we briefly overview the models and algorithms used in this paper.

3.1. Policy Iteration

Policy iteration is an algorithm that generates a sequence of improving policies, by alternating between policy evaluation and policy improvement. In large action spaces, approximation is necessary to evaluate each policy and to determine its improvement.

POLICY IMPROVEMENT: LEARNING ACTION POLICY

Action policy $p_\sigma(a|s)$ can be trained using supervised learning. This action policy outputs a probability distribution over all eligible moves a. The policy is trained on randomly sampled state-action pairs (s, a) using stochastic gradient ascent to maximize the likelihood of the expert action a being selected in state s,

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial\sigma}. \tag{1}$$

The action policy is trained further by using policy gradient reinforcement learning (RL) (Sutton et al., 1999). The action policy is then updated at each time step t by stochastic gradient ascent in the direction that maximizes the expected outcome:

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial\rho} r(s_t), \tag{2}$$

where $r(s_t)$ is the return, which is the discounted sum of rewards for one whole episode from the perspective of the current player at time step t.

POLICY EVALUATION: LEARNING VALUE FUNCTIONS

The value function predicts the outcome from state s of games played using policy p for both players,

$$v(s) = \mathbb{E}[r(s)|s_t = s, a_t \sim p]. \tag{3}$$

The value function is approximated by the value estimator $v_\theta(s)$ with parameters θ. The value estimator is trained by state-reward pairs $(s, r(s))$ using stochastic gradient descent to minimize the mean squared error (MSE) between the predicted regression value $v_\theta(s)$ and the corresponding outcome $r(s)$,

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(r(s) - v_\theta(s)). \tag{4}$$

3.2. Monte Carlo Tree Search

A Monte Carlo tree search (MCTS) (Browne et al., 2012; Coulom, 2007a; Kocsis & Szepesvári, 2006) is a tree search algorithm for decision processes for finite-horizon tasks. It iteratively analyzes and expands nodes (states) of the search tree based on random sampling (actions) of the search space. From a root to a leaf node, it selects an action towards most promising moves based on a *selection function*. At the leaf node, the node is expanded by adding a new leaf to the tree. Then, a *rollout policy* plays a playout until reaching a terminal state to obtain reward. Then, the obtained reward is used to update information of the selected nodes from the root to the leaf.

Upper confidence bounds applied to trees (UCT) (Kocsis & Szepesvári, 2006) is a commonly used MCTS algorithm using an upper confidence bound (UCB) selection function. It is computed by the Chernoff-Hoeffding bound:

$$\arg\max_a \bar{v}_a + C\sqrt{\frac{\log \sum_b n_b}{n_a}}, \tag{5}$$

which is one-sided confidence interval on the expected value $\bar{v}_a$ with the number of visits n_a for each action a. The exploration-exploitation tradeoff is controlled by the constant C.

3.3. Kernel Regression

Kernel regression is a non-parametric estimator which uses a kernel function as a weight to estimate the conditional expectation of a random variable. Given the choice of kernel K and data set $(x_i, y_i)_{i=0}^n$, kernel regression is defined as follows:

$$E[y|x] = \frac{\sum_{i=0}^n K(x, x_i) y_i}{\sum_{i=0}^n K(x, x_i)}. \tag{6}$$

Here, kernel K is a function which defines the weight given a pair of two points. The denominator of kernel regression

is related to kernel density estimation which is also a non-parametric method for estimating the probability density function of a random variable. The kernel density is defined by $W(x) = \sum_{i=0}^{n} K(x, x_i)$.

One typical kernel function is the Gaussian probability density which is defined as follows:

$$K(\mathbf{x}, \mathbf{x}') = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^T \Sigma^{-1} (\mathbf{x} - \mathbf{x}')\right)}{\sqrt{(2\pi)^k |\Sigma|}}. \quad (7)$$

4. Deep Reinforcement Learning in Continuous Action Spaces

4.1. The Policy-Value Network

Recently, deep CNNs have produced remarkable performances in playing Atari games (Mnih et al., 2015) and Go (Silver et al., 2016). We employ a similar architecture for curling. We pass the position of the stones on the ice sheet as a 32x32 image and use convolutional layers to construct a representation of the position. We use these neural networks to reduce the effective depth and breadth of the search tree evaluating the position using a value network and sampling actions using a policy network as shown in Figure 1.

The Shared Layers

Our policy-value network takes the following inputs; the stones' location, the order to *tee*[2], the number of shots, and flags to indicate whether each grid cell inside of the *house*[3] is occupied by any stone. After the first convolutional block, the nine residual blocks (He et al., 2016) follow, which are shared during training procedure.

The Policy Head

The policy head $\mathbf{p}_\theta$ outputs $\mathbf{p}$ which is the probability distribution of actions for selecting the best shot out of 32x32x2 discretized actions (clockwise or counter-clockwise spin). In supervised training, the network is trained to follow actions of the reference program AyumuGAT'16 (Section 6.2). In reinforcement learning (self-play) the network is trained to follow the actions generated by **Algorithm 1**. The policy head has two more convolutional layers (to make a total 21 convolutional layers) where ReLU activation is used. The best policy is trained and selected in the last layer using the softmax activation function.

The Value Head

The value head $\mathbf{v}_\theta$ outputs the probability distribution of each score at the conclusion of each end, in which both teams throw eight stones in turn. Thus, +8 and -8 are respectively the maximum and minimum scores. The value head has two fully connected layers on top of a convolutional layer (a total of 20 convolutional layers and 2 fully connected layers). The last layer of the value head outputs a vector $\mathbf{v}$, the probability distribution of 17 different outcomes $[-8, 8]$ using the softmax activation function.

[2]The center point of the house.

[3]The three concentric circles where points are scored.

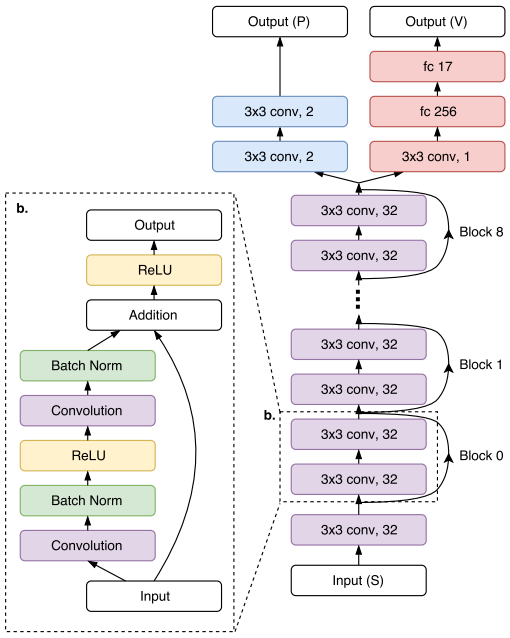

Figure 2. A detail description of our policy-value network. The shared network is composed of one convolutional layer and nine residual blocks. Each residual block (explained in **b**) has two convolutional layer with batch normalization (Ioffe & Szegedy, 2015) followed by the addition of the input and the residual block. Each layer in the shared network uses 3x3 filters. The policy head has two more convolutional layers, while the value head has two fully connected layers on top of a convolutional layer. For the activation function of each convolutional layer, ReLU (Nair & Hinton) is used.

4.2. Continuous Action Search

When the action space is not discretized, it is difficult to specify or select an action from the huge continuous space. When the action space is discretized, it could lead to a significant loss of information. Our search algorithm starts from actions initialized by the policy network in the discretized space. It then explores and evaluates actions with the value network using an Upper Confidence Bound (UCB) in the continuous space.

The search procedure of our algorithm follows the MCTS algorithm (selection, expansion, simulation, and backpropagation), specifically UCT (Kocsis & Szepesvári, 2006), which uses the UCB as the selection function. For information sharing between actions, we used kernel regression

Algorithm 1 KR-DL-UCT

1: $\mathbf{p}_\theta \leftarrow$ the policy network
2: $\mathbf{v}_\theta \leftarrow$ the value network
3: $s_t \leftarrow$ the current state
4: $A_t \leftarrow$ a set of visited actions in s_t
5: $expanded \leftarrow$ false
6: **if** s_t is terminal **then**
7: **return** Score(s_t), false
8: $a_t \leftarrow arg\max_{a \in A_t} \mathbb{E}[\bar{v}_a|a] + C\sqrt{\frac{\log \Sigma_{b \in A_t} W(b)}{W(a)}}$
9: **if** $\sqrt{\sum_{a \in A_t} n_a} < |A_t|$ **then**
10: $s_{t+1} \leftarrow$ TakeAction(s_t, a_t)
11: $\vec{reward}, expanded \leftarrow$ KR-DL-UCT(s_t)
12: **if not** $expanded$ **then**
13: $a'_t \leftarrow arg\min_{K(a_t,a)>\gamma} W(a)$
14: $A_t \leftarrow A_t \cup a'_t$
15: $s_{t+1} \leftarrow$ TakeAction(s_t, a'_t)
16: $A_{t+1} \leftarrow \cup_{i=1}^{k} \{a_{init}^{(i)}\}$ s.t. $a_{init}^{(i)} \sim \pi_{a|s_{t+1}}$ // Policy net
17: $\vec{reward} \leftarrow \mathbf{v}_\theta(s_{t+1}|s_t, a'_t)$ // Value net
18: $\vec{v}_{a_t} \leftarrow \frac{1}{n_{a_t}+1}(n_{a_t}\vec{v}_{a_t} + \vec{reward})$
19: $n_{a_t} \leftarrow n_{a_t} + 1$
20: **return** $\vec{reward}$, true

based UCT (KR-UCT) (Yee et al., 2016). In KR-UCT, the expected value and the number of visits is estimated by kernel density estimation and kernel regression respectively.

$$\mathbb{E}(\bar{v}_a|a) = \frac{\sum_{b \in A_t} K(a,b)\bar{v}_b n_b}{\sum_{b \in A_t} K(a,b) n_b}, \; W(a) = \sum_{b \in A_t} K(a,b) n_b \quad (8)$$

For the selection, this information sharing enables our model to make decisions by considering the result of similar actions. By sampling actions within the execution uncertainty of the selected action a_t and choosing less explored space, an action can be selected to lead an effective expansion. The details of **Algorithm 1** with respect to the steps for an MCTS is described below.[4]

Selection As a selection function, our algorithm uses a variation of the UCB formula (line 8). The scores and the number of visits for each node are estimated using the information from already visited sibling nodes b which are in A_t. They are denoted by $\mathbb{E}(\bar{v}_a|a)$ and $W(a)$ respectively. The expected probability distribution of values $\bar{v}_a$ is weighted by the winning percentage, which is described in Section 5.3. For an one-end game, which is used for self-play games, the expected value of the distribution would be $\bar{v}_a$. For the final selection, the algorithm chooses the most visited node and selects the corresponding action.

$$a^* = \arg\max_{a \in A_t} W(a). \quad (9)$$

[4]Source codes are available at https://github.com/leekwoon/KR-DL-UCT

Expansion Before expanding a node, we used *progressive widening* (Coulom, 2007b; Yee et al., 2016) (line 9) to overcome the problem of vanilla UCT in a large action space (e.g. continuous space), in which exploring all possible actions results in a shallow search. With this approach, a new node is expanded only when existing nodes are visited a sufficient number of times.

From the selected action a_t, actions are sampled that satisfy the inequality '$K(a_t, a) > \gamma$' and an action which minimizes $W(a)$ is selected from among these (line 13). In our implementation, we generate samples from the Gaussian kernel by setting the mean to a_t with a specified variance, instead of setting the hard bound γ. This sampling allows our algorithm to explore and search for actions in the continuous space.

TakeAction(s_t, a_t) requests the curling simulator to generate the next state by delivering a stone a_t given the current state s_t, and then the simulator returns the position of all remaining stones (line 10, 15).

For all expanded states, including the root, our algorithm initializes the k actions (line 16) for each state with the policy π_a,

$$\pi_{a|s_{t+1}} = \frac{\mathbf{p}(a|s_{t+1})^{1/\tau}}{\Sigma_b \mathbf{p}(b|s_{t+1})^{1/\tau}}. \quad (10)$$

With the temperature parameter τ, the k initial actions are sampled using the unbiased distribution $\pi_{a|s_{t+1}}$.

Simulation To evaluate the initialized or expanded states, our value network $\mathbf{v}_\theta$ is used instead of a simulation for several depth following a default policy (e.g. random policy). Without any domain knowledge (i.e., rule-based strategies), this makes the search procedure faster (i.e., it does not need to simulate for a certain depth) than using hand-crafted functions. For the given new state s_{t+1}, a vector of probabilities is returned with a score ranging between [-8,8] (line 17). At the terminal state, the score is measured and returned as an one-hot vector (line 6-7).

Backpropagation With the new value vector *reward*, our algorithm updates v_{a_t} and n_{a_t} along the selected path (lines 18-19).

5. Learning Pipeline

5.1. Supervised Learning

During supervised training, we jointly train our policy-value network using 0.4 million state-action pairs (s, a) from the reference program (Section 6.2). The policy subnetwork is directly learned from a move a at state s in the training data. Instead of the final game outcome (win or lose), the value subnetwork is learned by d-depth simulations and bootstrapping of the prediction to handle the high variance in rewards resulting from a sequence of stochastic moves.

Thus, our framework does not use any hand-crafted rollout policy.

We sample m state-action pairs (s, a) from the training data. For each state-action pair (s_t, a_t), we generate d-depth state s_{t+d} randomly by considering execution uncertainty on a_{t+d-1} only. Using the value subnetwork, s_{t+d} is evaluated and prediction $\mathbf{z}_t$ is used for learning the value.

The policy-value network $f_\theta(s)$ is adjusted to maximize the similarity of two pairs of vectors $(\mathbf{p}, \mathbf{v})$ and $(\boldsymbol{\pi}, \mathbf{z})$, where $\boldsymbol{\pi}$ and $\mathbf{z}$ are the policy and the value of the reference program. To maximize the similarity of the neural network pair $(\mathbf{p}, \mathbf{v})$ to the pair $(\boldsymbol{\pi}, \mathbf{z})$, we use stochastic gradient descent on the multi-task loss function l, which sums the cross-entropy losses with a batch size of 256 examples, a momentum of 0.9, and a L2 regularization parameter c of 0.0001.

$$(\mathbf{p}, \mathbf{v}) = f_\theta(s), \tag{11}$$

$$l = -\mathbf{z}^T \log \mathbf{v} - \boldsymbol{\pi}^{\mathbf{T}} \log \mathbf{p} + c||\theta||^2. \tag{12}$$

Here, we set $\pi_a = 1$ for the action a selected by the reference program. We find that eliminating a part of the action space beyond the *backline*[5] was very effective in learning strong shots, like *takeout*[6] shots. Combining the policy and value networks in the policy-value network (a multi-task learning scheme) was also effective in improving performance. The network was trained for roughly 100 epochs. The learning rate was initialized at 0.01 and reduced twice prior to termination.

5.2. Self-Play Reinforcement Learning

Policy improvement starts with a sample policy followed by executions of the MCTS using the proposed KR-DL-UCT algorithm. The searched policy is then projected back into the function space of the policy subnetwork.

The outcomes of self-play games are also projected back into the function space of the value subnetwork. These projection steps are achieved by training the policy-value network parameters to match the search probabilities and self-play game outcomes.

Classification based modified policy iteration (CBMPI) (Scherrer et al., 2015) is severely limited in evaluating policy in continuous action spaces because CBMPI cannot be used unless the action set is discretized. We handle this problem by exploring actions using a physics simulator in the continuous action space.

For each time-step t in self-play games, with root state s_t, our KR-DL-UCT returns two vectors, $\mathbf{z}_t$ and $\boldsymbol{\pi}_t$. $\mathbf{z}_t$ represents the estimated probability distribution of the game score obtained from similar actions computed by kernel regression, while $\boldsymbol{\pi}_t$ represents the probability distribution of actions and is proportional to the estimated visit counts based on kernel density estimation, $\pi_a \propto W(a)^{1/\tau}$, where τ is a temperature parameter.

The parameters θ of the policy-value network are initialized by supervised learning (Section 5.1) and continually updated by data $(\mathbf{s}, \boldsymbol{\pi}, \mathbf{z})$ sampled uniformly (with a fixed size) at random from the most recent history of the self-play. Here, we use the same loss function as in supervised learning.

5.3. Learning Long Term Strategies

The game of curling differs from typical turn-based two player games like Go and chess, because curling usually consists of eight or ten rounds called ends. The winner is decided based on the accumulated scores after finishing all ends. Thus, a strategy optimized for one-end games (8 turns per team) would not be the best strategy to win a multi-end game (Ahmad et al., 2016). Both the position and other features, such as the number of remaining ends and the difference between the accumulated scores, should be considered.

To select the best strategy in multi-end games, we construct a winning percentage table, WP table, which is updated using data from self-play games. The table is consist of two entries; the number of remaining ends n, and the difference between the accumulated scores until the current end δ. For example, $P_{win}(n{=}1, \delta{=}-1)$ is the winning probability (for the team who shoots first of the end) when one end remains and the difference of the accumulated scores is -1 (i.e., the team is down by one point). Using an one-end score distribution and $P_{win}(n, \delta)$, $P_{win}(n{+}1, \delta)$ can be computed iteratively.

The expected winning percentage of multi-end games is efficiently computed from the WP table and the probability distribution of the one-end score $[-8, 8]$.

6. Experimental Results

6.1. Simulated Curling

Curling is a two-team, zero-sum game. The goal is to throw a stone down a sheet of ice toward the house (the scoring area). Games are divided into rounds called ends, in which each team alternates throwing a total of eight stones each. When each end finishes, the team with the stone closest to the center of the house, called the tee, scores points. The number of points they receive is calculated by the number of stones in the house those are closer to the tee than any opponent stone.

When curling is played on an actual ice sheet, deciding the best strategy given the game situation is difficult because the ice conditions continuously change and each player has

[5] A line beyond the house area.

[6] An action to make a stone that hits another stone to remove it.

Program	First play			Second play			Total
	Win	Draw	Lose	Win	Draw	Lose	Winning Percentage
GCCSGAT'17	53	25	22	73	19	8	$74.0 \pm 6.22\%$
AyumuGAT'16	56	8	36	69	8	23	$66.5 \pm 6.69\%$
AyumuGAT'17	43	15	42	65	17	17	$62.3 \pm 6.87\%$
JiritsukunGAT'16	80	4	16	87	7	6	$86.3 \pm 4.88\%$
JiritsukunGAT'17	38	12	50	58	18	24	$55.5 \pm 7.04\%$

Table 1. The 8-end game results for KR-DRL-MES against other programs alternating the opening player each game. The matches are held by following the rules of the latest GAT competition.

different characteristics.

In this paper, we use simulated curling software (Ito & Kitasei, 2015) which assumes a stationary curling sheet. Thus, ice conditions are assumed to remain unchanged, the *sweeping*[7] is not considered, and asymmetric Gaussian noise is added to the every shot. The simulator is implemented using the Box2D physics library, which deals with the trajectory of the stones and their collisions.

6.2. Dataset

We use data from the public program AyumuGAT'16 (Ohto & Tanaka, 2017), which is based on a high-performance MCTS algorithm and a champion program of the Game AI Tournaments digital curling championship in 2016 (GAT-2016) (Ito).

6.3. Domain Knowledge

The following is the only domain knowledge we used.

- In a game of curling, there is an official rule called the *free-guard zone*[8] (FGZ) rule.[9] To handle this rule, we encode the number of turns for each end.
- The input features describing the position of the stones are represented as a 32x32 image. To overcome the loss of information from the discretization, we add additional features for each stone: whether the stone is in the house and their order in terms of their distance to the center of the house.
- The strategy in game of curling is invariant under reflection; thus, we utilize this knowledge to augment the data.

Any form of domain knowledge beyond the information listed above is not used. We summarize input features used in the policy-value networks in the supplementary material.

[7] An activity, brushing the ice surface after throwing a stone, to adjust stone's trajectory close to the intended one.

[8] Area between the tee line and the hog line which the stone must completely cross to be considered in play, excluding the house.

[9] The rule states that an opponent's rock resting in the free-guard zone cannot be removed from play until the first four rocks of an end have been played.

6.4. Settings

Our first algorithm, kernel regression-deep learning (KR-DL), is trained as described in Section 5.1. Our second algorithm, kernel regression-deep reinforcement learning (KR-DRL), follows the learning pipeline in Section 5.2. During self-play, each shot is selected after 400 simulations. For a continuous search, we set $C = 0.1$ and $k = 20$. For the first three shots during which the FGZ rule is applied, we set $\tau = 1$ to allow for more stones to be in playground during an end. Otherwise, we set $\tau = 0.33$ to follow the promising actions with high probability.

During a week of data collection, using 5 GPUs, 5 million game positions were generated and KR-DRL, initialized by KR-DL, was continually updated using data $(\mathbf{s}, \boldsymbol{\pi}, \mathbf{z})$ sampled uniformly at random from the most recent one million positions during self-play. Finally, the WP table (Section 5.3) generated from KR-DRL self-play is used for our third algorithm, kernel regression-deep reinforcement learning-multi ends strategy (KR-DRL-MES).

We compare our proposed algorithms with vanilla KR-UCT and programs which received first, second prize and third prize in the 2016 and 2017 GAT (Ito). For the vanilla KR-UCT, we used the same hyperparameter with 1,600 simulation as in (Yee et al., 2016). Unfortunately, the generator that creates list of promising shot from particular state is not publicly available, we used different generator based on handmade simple policy function (Ohto & Tanaka, 2017), such as drawing shot to the center of house, or hitting the opponent stone for the rollout policy and initialization of candidate actions. All the matches held with the rules of the last GAT competition: all programs are allowed about 3.4 seconds computation time per move and play 8-end game with known Gaussian execution uncertainty.

6.5. Results

First, to demonstrate the significance of our work, we compare the performance of a program trained by our proposed algorithm (KR-DL-UCT) with a baseline program trained

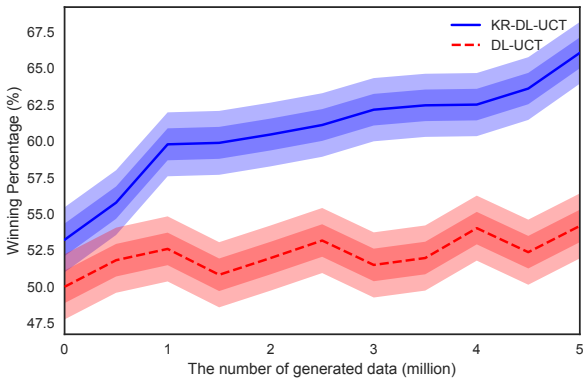

Figure 3. Learning curve for KR-DL-UCT and DL-UCT. The plot shows the winning percentages of 2,000 two-end games against DL-UCT with supervised learning only, 1,000 games as the first player and 1,000 games as the second player respectively. We compute the winning percentages by increasing the number of training shots.

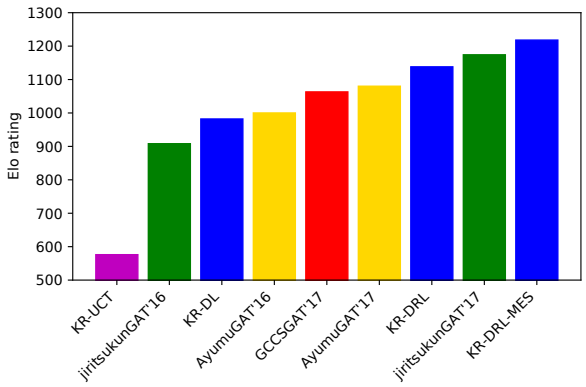

Figure 4. Elo rating and winning percentages of our models and GAT rankers. Each match has 200 games (each program plays 100 pre-ordered games), because the player which has the last shot (the hammer shot) in each end would have an advantage. Programs colored blue are our proposed programs.

by UCT only (DL-UCT).

We initialize both models with the supervised learning and then train further from shots of self-play games with two different algorithms. For the case of DL-UCT, each shot is selected with a constant (Silver et al., 2016) determining the level of exploration ($C_{puct} = 1.0$). We evaluate the programs with the winning percentages of two-end games against DL-UCT with supervised learning only. We also compared programs by increasing the number of training shots generated by self-play games.

Figure 3 shows the performance of KR-DL-UCT compared to DL-UCT. We could observe that KR-DL-UCT outperforms DL-UCT even without the self-play RL. With the supervised training only, KR-DL-UCT wins 53.23% against DL-UCT. KR-DL-UCT expedites the training procedure by improving the overall performance compared to DL-UCT under the self-play RL framework. After gathering 5 million shots from self-play, KR-DL-UCT wins 66.05% which is significantly higher than the winning percentage of DL-UCT.

Second, to evaluate our algorithm, we run an internal match among our proposed algorithm and other programs following GAT-2017 rules. Each algorithm played an equal number of games with and without the hammer shot against each opponent in the first end.

Figure 4 presents the Elo ratings of the programs trained by our framework (KR-DL, KR-DRL and KR-DRL-MES) and existing programs (KR-UCT, JiritsukunGAT'16, AyumuGAT'16, GCCSGAT'17, AyumuGAT'17 and JiritsukunGAT'17). The vanilla KR-UCT does not perform well compared to other programs optimized with many hand-crafted features. Our first model, KR-DL, trained using supervised learning from the shots of AyumuGAT'16 does not perform better than the reference program. KR-DRL, trained further from self-play reinforcement learning, outperforms the reference program and other recently successful programs. KR-DRL-MES, equipped with long-term strategy, achieves the state-of-the-art performance.

We play games between our KR-DRL-MES and notable programs. Table 1 presents details of the match results. Our KR-DRL-MES wins against AyumuGAT'16 and AyumuGAT'17 by a significant margin. Only JiritsukunGAT'17 which uses a deep neural network and hand-crafted features, shows a similar level of performance but KR-DRL-MES is still the victor.

In the supplementary material, we provide a video of a game played between KR-DRL-MES and JiritsukunGAT'17.

7. Conclusion

We present a new framework which incorporates a deep neural network for learning game strategy with a kernel-based Monte Carlo tree search from a continuous space. Without the use of any hand-crafted feature, our policy-value network is successfully trained using supervised learning followed by reinforcement learning with a high-fidelity simulator for the Olympic sport of curling. The program trained under our framework outperforms existing programs equipped with several hand-crafted features and won an international digital curling competition.

Acknowledgements

We wish to thank Viliam Lisy and other authors of the KR-UCT paper sharing the part of their source code.
This work was supported by Institute for Information and communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2017-0-01779, A machine learning and statistical inference framework for explainable artificial intelligence, and No.2017-0-00521, AI curling robot which can establish game strategies and perform games).

References

Ahmad, Z., Holte, R., and Bowling, M. Action selection for hammer shots in curling. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI*, pp. 561–567, 2016.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. Unifying count-based exploration and intrinsic motivation. In *Proceedings of the Neural Information Processing Systems, NIPS*, pp. 1471–1479, 2016.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, pp. 1–43, 2012.

Bubeck, S., Stoltz, G., Szepesvári, C., and Munos, R. Online optimization in x-armed bandits. In *Proceedings of the Neural Information Processing Systems, NIPS*, pp. 201–208, 2008.

Buro, M. From simple features to sophisticated evaluation functions. In *Computers and Games*, pp. 126–145, 1999.

Campbell, M., Hoane Jr., A., and Hsu, F.-H. Deep blue. *Artificial Intelligence*, pp. 57–83, 2002.

Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pp. 72–83, 2007a.

Coulom, R. Computing Elo ratings of move patterns in the game of go. *International Computer Games Association, ICGA*, pp. 198–208, 2007b.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 770–778, 2016.

Heo, M.-H. and Kim, D. The development of a curling simulation for performance improvement based on a physics engine. *Procedia Engineering*, pp. 385–390, 2013.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning, ICML*, pp. 448–456, 2015.

Ito, T. The 4th uec-cup digital curling tournament in game artificial intelligence tournaments. URL http://minerva.cs.uec.ac.jp/curling/wiki.cgi?page=GAT_2018.

Ito, T. and Kitasei, Y. Proposal and implementation of digital curling. In *Proceedings of the IEEE Conference on Computational Intelligence and Games, CIG*, pp. 469–473, 2015.

Jackson, S. *Unity 3D UI Essentials*. 2015.

Kocsis, L. and Szepesvári, c. Bandit based monte-carlo planning. In *Proceeding of European Conference on Machine Learning, ECML*, pp. 282–293, 2006.

Lai, T. and Robbins, H. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, pp. 4–22, 1985.

LeCun, Y. and Bengio, Y. The handbook of brain theory and neural networks. pp. 255–258, 1998.

Lozowski, E., Szilder, K., Maw, S., Morris, A., Poirier, L., and Kleiner, B. Towards a first principles model of curling ice friction and curling stone dynamics. pp. 1730–1738, 2015.

Maeno, N. Dynamics and curl ratio of a curling stone. *Sports Engineering*, pp. 33–41, 2014.

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

Nair, V. and Hinton, G. Rectified linear units improve restricted boltzmann machines.

Ohto, K. and Tanaka, T. A curling agent based on the monte-carlo tree search considering the similarity of the best action among similar states. In *Proceedings of Advances in Computer Games, ACG*, pp. 151–164, 2017.

Parberry, I. *Introduction to Game Physics with Box2D*. 2013.

Schaeffer, J., Culberson, J., Treloar, N., Knight, B., Lu, P., and Szafron, D. A world championship caliber checkers program. *Artificial Intelligence*, pp. 273–289, 1992.

Scherrer, B., Ghavamzadeh, M., Gabillon, V., Lesner, B., and Geist, M. Approximate modified policy iteration and its application to the game of tetris. *Journal of Machine Learning Research*, pp. 1629–1676, 2015.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, pp. 484–489, 2016.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature*, pp. 354–359, 2017.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the Neural Information Processing Systems, NIPS*, pp. 1057–1063, 1999.

Yamamoto, M., Kato, S., and Iizuka, H. Digital curling strategy based on game tree search. In *Proceedings of the IEEE Conference on Computational Intelligence and Games, CIG*, pp. 474–480, 2015.

Yee, T., Lisý, V., and Bowling, M. Monte carlo tree search in continuous action spaces with execution uncertainty. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI*, pp. 690–697, 2016.