Game AI and Reinforcement Learning

Wayne Tu, Michael Wan, Edison Chen

INFOR 29th, Taipei Municipal Chien Kuo High School

Abstract

In this thesis we introduce a program that is capable of playing two-player 3D-Bingo game. The program consists of two phases: value network training and policy search. The value network used in our program is a multi-layer neural network which back-propagates with respect to *temporal difference* procedure. Two types of policy searching are considered in this thesis, *Minimax* and *Monte-Carlo tree search*, the former is a traditional algorithm aims to minimize opponent's maximum reward, the latter generates policy based on numbers of game simulation. Discussion about pros and cons of two methods and results of different learning parameters is given at the end of the thesis.

1 Introduction

Game AI has been a popular topic in the field of computer science and game theory for decades and has gained great progress by researchers from all over the world. In 1997, a chess-playing computer called Deep Blue developed by IBM defeated the world champion Garry Kasparov 3.5-2.5 in six games; Not long ago, with the rapid development of deep neural networks and higher computing performances of GPUs and CPUs, the Go-playing artificial intelligence Alpha Go by Google Deep Mind beat Lee Sedol 4 to 1 in 2016. In recent years, numbers of machine learning libraries like Tensorflow or Theano have been developed, combined with Cloud services such as AWS or Microsoft Azure, building AIs becomes much more affordable and efficient for smaller teams or individuals.

2 Reinforcement Learning

One of the fundamental problems in artificial intelligence is that of sequential decision making, reinforcement learning is one of the solution. More specifically, reinforcement learning algorithms are able to let an agent learned from its experiences obtained from interaction with the environment and by doing so, the agent will eventually make correct or appropriate decision in some given states. In the chapter, we'll introduce the formal description of the environment and some notations that will be used later.

2.1 Markov Decision Process

Markov Decision Process (MDP) provides a formal description of an agent taking action and thus gaining reward from an environment. A typical definition of MDP M is a tuple $(S, D, A, P_{sa}(\cdot), \gamma, R)$ consisting of:

- S: Set of states of the environment.
- D: Initial state distribution.
- A: Set of actions.

- $P_{sa}(\cdot)$: State transition probability, for each $s \in S$ and $a \in A$ this gives the probability distribution over to which state by taking action a in state s.
- γ: Discount factor in $[0, 1]$.
- R: $S \mapsto \mathbb{R}$, reward function, bounded by $R_{max}(|R(s)| \leq R_{max}, \forall s \in S)$.

The MDP proceeds as follow. First, we are at state s_0 based on D, then, at each time step t, we take action a_t and advance to state s_{t+1} based on $P_{s_t a_t}(\cdot)$. By repeating the above events, we can obtain the total reward:

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots = \sum_{t=0}^{\infty} \gamma^t R(s_t)$$

In reinforcement learning, generally, our goal is to maximize the total reward by taking a sequence of action $a_0, a_1, a_2, \ldots a_{end}$. We also want to succeed as soon as possible. Thus, a discount factor $\gamma(\gamma < 1)$ is introduced to lower the impact of future reward and our agent will hence prefer to reach the goal (i.e., gain reward) faster. Sometimes, MDP with $\gamma = 1$ is also be of interest, which is called an undiscounted Markov Decision Process.

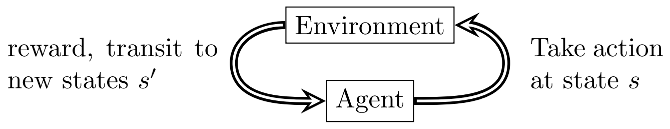


Figure 1: Markov Decision Process

2.2 V-function and Q-function

A policy in MDP is a function $\pi : S \mapsto A$, that is, for each $s \in S$, this gives the corresponding action to take, which is $\pi(s)$. A value function $V^\pi : S \mapsto \mathbb{R}$ is the expected reward of policy π starting from state s:

$$V^\pi(s) = \mathbb{E}_\pi[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \mid s_0 = s]$$

Value function for state-action pair is also defined, called Q-function ($Q : S \times A \mapsto \mathbb{R}$):

$$Q^\pi(s, a) = \mathbb{E}_\pi[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots \mid s_0 = s, a_0 = a, \forall t > 0 \; a_t = \pi(s_t)]$$

The above is the expected reward of policy π starting from state s and first taking action a.
We also define the optimal V-function $V^* : S \mapsto \mathbb{R}$, which is the maximum expected return starting from state s:

$$V^*(s) = \max_\pi V^\pi(s)$$

The optimal Q-function is similar:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

Also, the Bellman Equations introduce a recursive definition of the V-function:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s')$$

$$V^*(s) = \max_a R(s, a) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$$

Note that $R(s, a)$ is another typical expression of the reward function, representing the reward obtained by taking action a in state s.

3 Value Estimation

In the field of reinforcement learning, it is necessary to estimate the value of the given state in order to guide the agent on action selection which leads to the highest reward. The following methods, for the sake of simplicity, assume small enough state and action space size which can be directly stored in an array or a look-up table.

3.1 Dynamic Programming

The dynamic programming algorithm (or DP for short) is based on the Bellman equations described above, and furthermore assumes that the transition probability $P_{sa}(\cdot)$ is known.
First of all, we arbitrarily choose the initial approximation v_0 (e.g., return 0 for all state s), and repeatedly update the approximation by computing the following:

$$v_{k+1}^{\pi}(s) = \mathbb{E}_{\pi}[R_{t+1} + \gamma v_k^{\pi}(S_{t+1}) \mid S_t = s] = \sum_{s' \in S} P_{s\pi(s)}(s')[R(s, \pi(s)) + \gamma v_k^{\pi}(s')]$$

As $k \to \infty$, $v_k^{\pi}(s)$ will eventually converge to the true value of s.

Algorithm 1 Dynamic Programming

```
procedure DYNAMIC PROGRAMMING
    Initialize $v(s) \leftarrow 0$ for each $s \in S$
    repeat
        $\delta \leftarrow 0$
        for each $s \in S$ do
            $temp \leftarrow v(s)$
            $v(s) \leftarrow \sum_{s'} P_{s\pi(s)}[R(s) + \gamma v(s')]$
            $\delta \leftarrow \max(\delta, |temp - v(s)|)$
    until $\delta < \epsilon$
```

The policy iteration approach can be done with policy evaluation. For formally, we choose an arbitrary policy π_0 initially and evaluate it. Once the evaluation is done, π_0 can be optimized if there exists an action $a \neq \pi(s)$ satisfy $Q^{\pi}(s, a) \geq v^{\pi}(s)$, one can perform $\pi_1(s) = a$ and by repeating the process, obtain the optimal policy π^* eventually.

$$\pi_0 \to v^{\pi_0} \to \pi_1 \to v^{\pi_1} \to \pi_2 \to \cdots \to \pi^*$$

3.2 Monte-Carlo Method

Although the DP method holds a straightforward approach to estimate the value function, prior knowledge to the transition probability is usually unrealistic. Monte-Carlo method requires no knowledge to the dynamics of the environment, on the contrary, MC evaluates the state based on *experience*: sample sequences of states, actions and rewards from on-line or simulated interaction with the environment. In particular, given a set of episodes obtained by following policy π and passing through s, we call each occurrence a *visit* to s. The main idea behind Monte-Carlo is to average the return value of *visits*. There are two typical Monte-Carlo Method, one of which is *every-visit* MC, the other is called *first-visit* MC. The difference is quite obvious, *every-visit* MC averages every occurrences of s in episodes, while *first-visit* MC averages just the first occurrence of s in episodes.

Algorithm 2 First-Visit MC

```
procedure FIRST-VISIT MC METHOD
    Initialize $V(s) \leftarrow 0$ for each $s \in S$, $Return(s) \leftarrow 0$ for each $s \in S$
    repeat
        $E \leftarrow$ Generate an episode with respect to $\pi$
        for each $s \in E$ do
            $G \leftarrow$ return value of first occurrence of $s$
            append $G$ to $Return(s)$
            $V(s) \leftarrow average(Return(s))$
    until terminated
```

Monte-Carlo Method also converges when average time goes to infinity.

3.3 Temporal Difference

Temporal Difference is a combination of Dynamic Programming and Monte-Carlo Method. Like MC, TD requires no prior knowledge of the environment and evaluate states based on experience; like DP, TD updates $v(s_t)$ immediately after visiting the successive state s_{t+1}, which is called *bootstrapping* (i.e., estimate s based on another estimation).
Recall that the value function is defined:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s]$$

The Monte-Carlo samples through the first portion of the equation, thus makes it an estimation. However, DP is an estimation not for the expectation but for the uncertainty of $v_{k+1}(s_{t+1})$. TD is a combination of the two, and hence it's an estimation for both reasons. The simplest TD method, called TD(0), is

$$V_{k+1}(s_t) = V_k(s_t) + \alpha[R_{t+1} + \gamma V_k(s_{t+1}) - V_k(s_t)]$$

where α is a real number in $[0, 1)$ representing the portion of the value of successive state s_{t+1} added to $V(s)$.

Algorithm 3 TD(0)

```
procedure TD(0)
    Initialize $V(s) \leftarrow 0$ for each $s \in S$
    $s \leftarrow$ initial state
    for each time step $t$ in episode do
        $A \leftarrow \pi(s)$
        Take action $A$, get reward $R$, transform into new state $s'$
        $V(s) \leftarrow V(s) + \alpha[R + \gamma V(s') - V(s)]$
        $s = s'$
```

3.3.1 Eligibility trace and TD(λ)

Eligibility traces are the bridge connecting TD method and Monte-Carlo. In the previous section, we update the value estimation of the state by the reward and the value of the next state. More generally, if we consider update the value of current state based on the reward obtained by n time step combined with $V(s_{t+n})$ and called it a n-step backup, then the method in the previous section is essentially a one-step backup. The target of update G can be calculated according to the above definition:

$$G_t^{(1)} = R_{t+1} + \gamma V(s_{t+1})$$

$$G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(s_{t+1})$$

$$\vdots$$

$$G_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n V(s_{t+n})$$

Thus, the mathematical definition of n-step backup is:

$$V(s_t) = V(s_t) + \alpha[G_t^{(n)} - V(s_t)]$$

Backups can be done not just toward any n-step return, but toward any average of n-step return. More formally, an averge target of update can be defined as a linear combination of a set of n-step return, where the summation of the coefficient will always be equal to 1.

$$\bar{G_t} = \sum_{k=1}^{n} c_k G_t^{(k)}, \sum_{k=1}^{n} c_k = 1$$

The TD(λ) algorithm can be derived from the idea of averaging n-step return. In particular, every return is averaged, each weighted propotional to λ^{n-1}, where $0 \leq \lambda \leq 1$, a normalization factor of $1 - \lambda$ is added to ensure the sum of weights is equal to 1.

$$G_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

And the update of the value estimation is:

$$V(s_t) = V(s_t) + \alpha[G_t^{\lambda} - V(s_t)]$$

4 Neural Network

In the previous section we introduced three value estimation method and their tabular implementation. However, in real world problem, the state and action space size is usually intractably too large to store in an array or a look-up table (e.g., we'll have a 10^{20}-size state space in the game on which we implement reinforcement learning). Thus, deep learning, which involves a neural network approach to both state evaluation and feature selection, plays a significant role in modern reinforcement learning.

A neuron is a node consisting of several input wires and one output wire. The neuron takes the input data (X) and parametrizes them with weights (W) on the input wire, then sums up the weighted input and *activates* them. (Note that the non-linearity of the activation function is important, for that the linear combination of a linear combination is still a linear combination.)

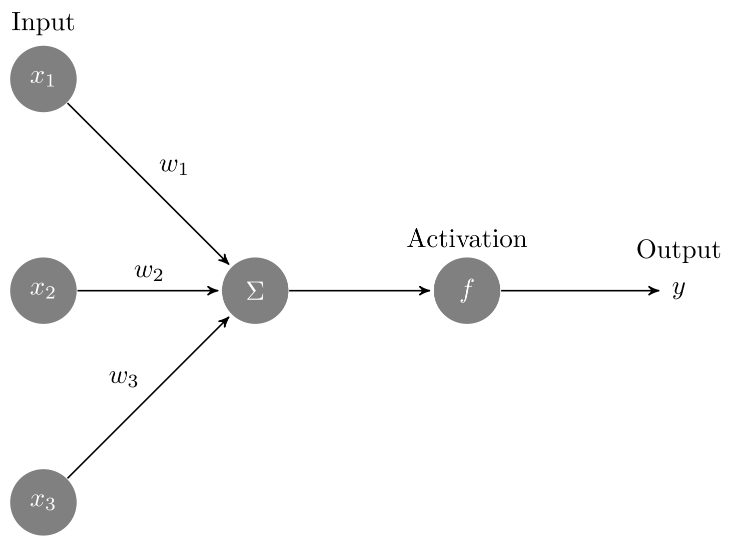
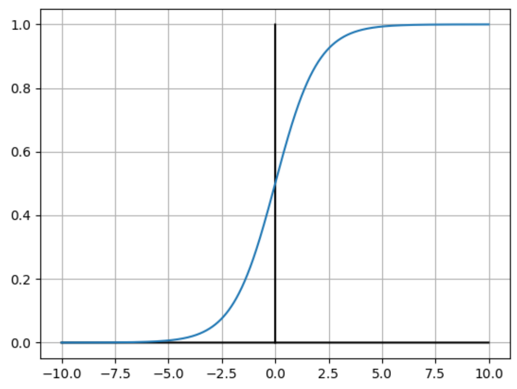


Figure 2: Simplified Neuron

If we regard input data as a vector X and weights as another vector W, the neuron described above is essentially calculating $z = W^T X, y = f(z)$.
In practice, an extra input unit with constant value of 1 is often added to the neurons, called a *bias* unit, denoted by x_0. Note that in neural network, the non-linearity of the activation function is crucial, for that the linear combination of another linear combination is stall a linear combination. *Sigmoid* and *Relu* are two of the widely-used activation functions:

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$Relu(x) = \max(x, 0)$$


(a) *sigmoid*

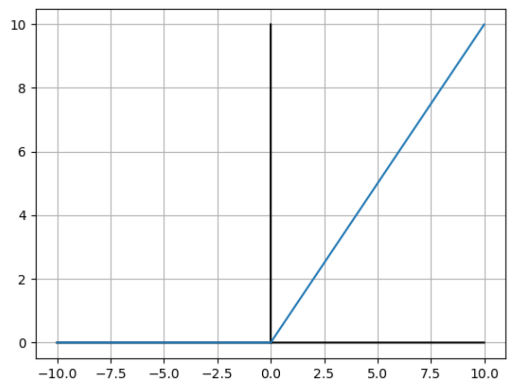

(b) *relu*

Figure 3: Examples of activation function

4.1 Forward Propagation

A neuron network comprises many neurons, which forms several layers, where each layer, besides the input layer, takes the output of the previous layer as the input of itself, shown as Figure 3 below:

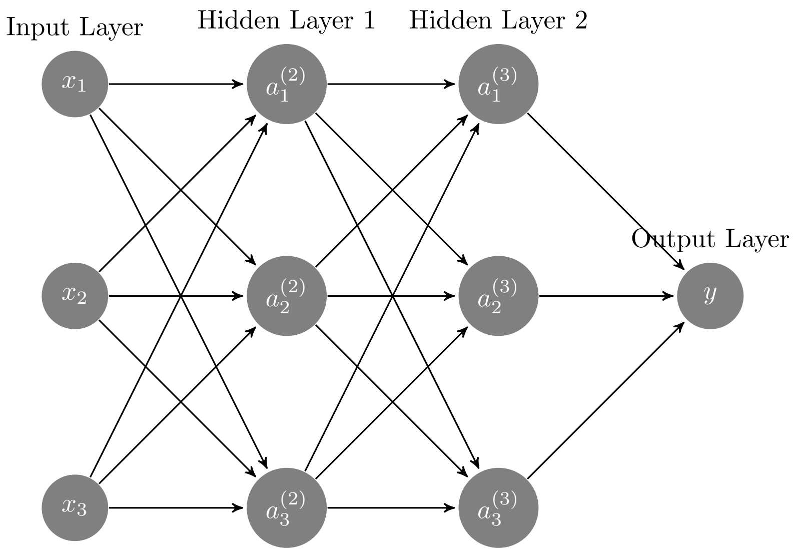


Figure 4: Neuron Network with two hidden layers

$W_{ij}^{(l)}$ denotes the weight between the jth node of layer l to the ith node of layer $l+1$. Thus, take Figure 3 for example, input of the second node of hidden layer 1 is $z_2^{(2)} = W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3$, after receiving the input, the neuron processes the input with the activation function, then output the activated sum, that is, $f(z_2^{(2)})$. According to the previous description, we can hence calculate the output of this neuron network with a vectorized implementation, where x is the input vector, $W^{(l)}$ is a $n_{l+1} \times (n_l + 1)$ matrix (n_i represents the number of unit in layer i).

$$z^{(2)} = W^{(1)}x, a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)}a^{(2)}, a^{(3)} = f(z^{(3)})$$
$$z^{(4)} = W^{(3)}a^{(3)}, y = f(z^{(4)})$$

4.2 Backward Propagation

Backward propagation is an algorithm of computing $\frac{\partial J}{\partial W_{ij}^{(l)}}$ (where J is the cost function) in the neuron network in order to minimize the error of our prediction using gradient descent which will be illustrated later.
We will be using Figure 3 as a concrete example to demonstrate the idea of backward propagation. In particular, suppose we know that the precise output given input X should be y', we then compute the error term of the output layer, which is $\delta^{(4)} = y - y'$ (assume that we're using the square-sum as the cost function), where $\delta^{(l)} \in \mathbb{R}^{n_l}$ denotes the error term of layer l.
After that, $\delta^{(3)}$ is computed:

$$\delta^{(3)} = (W^{(3)})^T \delta^{(4)} \circ f'(z^{(3)})$$

where $\circ$ is the element-wise multiplication of two vectors, $f'(z)$ is the derivative of f evaluated by z.
With the error term being calculated, we can now compute the partial derivative of each unit by

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta^{(l+1)}$$

4.3 Gradient Descent

Gradient descent is one of the most popular algorithm to perform optimization and to minimize the cost function in the neural network. The concept behind gradient descent is, suppose we're minimizing $J(w)$ parameterized by $w \in \mathbb{R}^d$, we'll update the parameters in the opposite direction of the gradient of the cost function (which is $\frac{\partial J(w)}{\partial w_j}$). In particular, we will introduce η which represents how far we *step* at each update and perform:

$$w_j := w_j - \eta \frac{\partial J(w)}{\partial w_j} \ (\forall 1 \leq j \leq d)$$

or with a vectorized implementation:

$$w := w - \eta \nabla_w J(w)$$

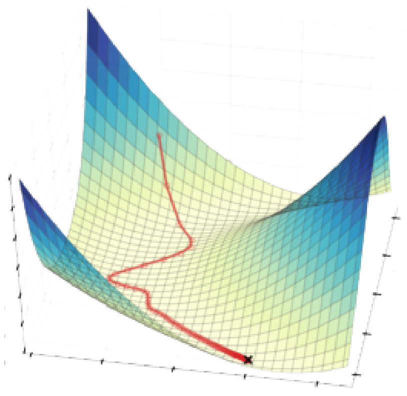


Figure 5: Gradient Descent

4.3.1 Adagrad

Adagrad is an algorithm for gradient-based optimization which adopt different learning rate to parameters. Adagrad performs larger updates for infrequent parameters and smaller updates for frequent parameters, thus it is suited for optimization dealing with sparse data. Recall that we update our parameter w with gradient descent:

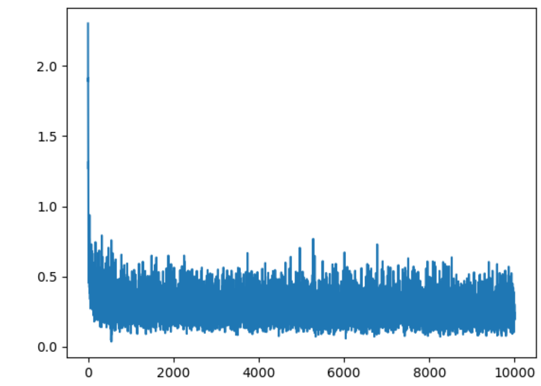
$$g_t = \nabla_w J(w_t), w_{t+1,i} = w_{t,i} - \eta \cdot g_{t,i} \ \forall 1 \leq i \leq d$$

However, in Adagrad, the learning rate for w_i is modified based on past gradient of w_i:

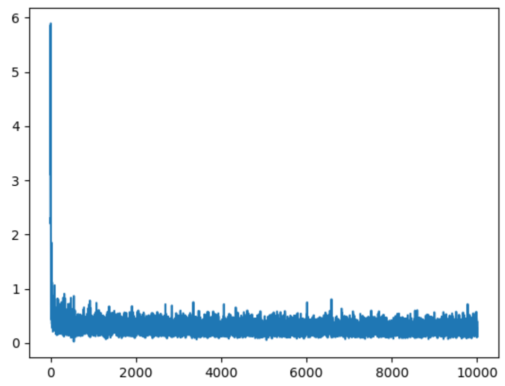
$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

where $G_t \in \mathbb{R}^{d \times d}$ is a matrix that each diagonal element $G_{t,ii}$ corresponds to the square sum of gradients with respect to w_i on time t. A real number ϵ (usually $\epsilon < 10^{-18}$) is added to the denominator to prevent division by zero. So the vectorized implementation of Adagrad simply performs the following:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \circ g_t$$

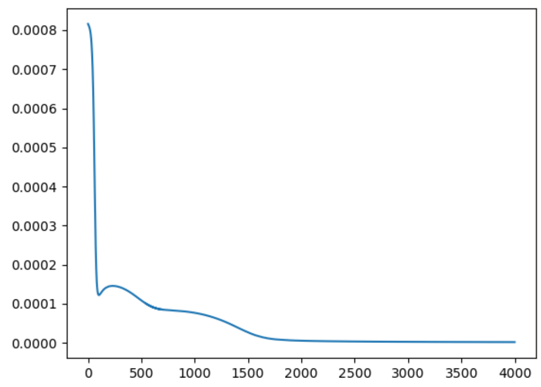


(a) SGD

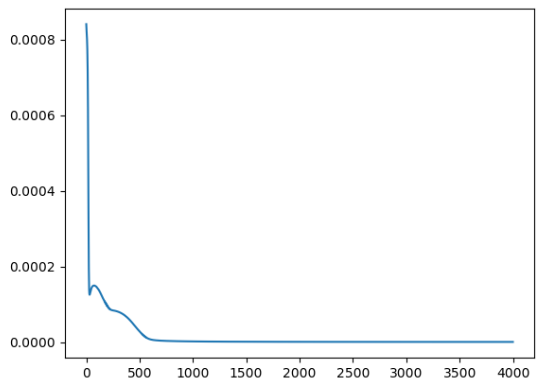


(b) Adagrad

Figure 6: Cross Entropy on MNIST handwritten digits recognition



(a) SGD



(b) Adagrad

Figure 7: Square error on 3D-Bingo implmentation

5 Policy Search

With the ability to evaluate the given state using neural network, we then have to choose action which leads to higher reward or a more promising state. Despite the fact that greedily choosing action which profits the most is an averaging good approach, in game playing, specially for games involving two or more players, actions that make long term profit are often preferred.

5.1 Minimax Tree Search

Minimax is a decision rule used in decision theory and game theory for minimizing the possible loss for the worst case (i.e., maximum loss) which covers both the cases where players take alternative and those where they make simultaneous move. The search tree consists of two kinds of nodes: the *Max* nodes and the *Min* nodes, where the agent chooses action in order to both maximizing its own profit and minimizing opponent's reward. In particular, we recursively span the search tree using DFS (depth-first search), when the given depth is reached or the state is terminal, we evaluate the state and return the estimated value. In *Max* node, it receives the *maximum* value of its children. On the contrary, *Min* node receives the *minimum* value of its children, that is, minimizing opponent's interest. The process of this algorithm is shown as Figure 7:

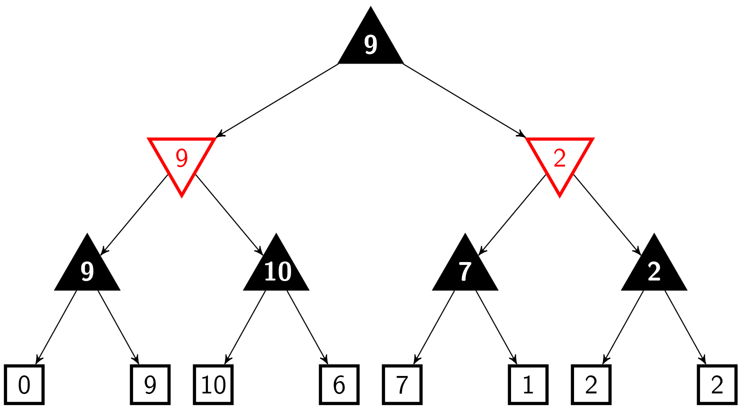


Figure 8: Minimax Tree Search

Note that in Figure 7, black nodes correspond to the *Max* nodes while red nodes correspond to the *Min* nodes and rectangle nodes are leaf nodes.

Algorithm 4 Minimax

```
procedure MINIMAX($s, depth, level$)
    if $depth = 0$ or $s$ is terminal then return evaluate($s$)
    if level is $Max$ then
        $best \leftarrow \infty$
        for each valid action $a$ do
            $s' \leftarrow$ successive state of $s$ taking $a$
            $best \leftarrow \max(best, Minimax(s', depth - 1, Min))$
        return $best$
    else
        $worst \leftarrow -\infty$
        for each valid action $a$ do
            $s' \leftarrow$ successive state of $s$ taking $a$
            $worst \leftarrow \min(worst, Minimax(s', depth - 1, Max))$
        return $worst$
```

5.2 Alpha-Beta Pruning

Alpha-Beta pruning is a searching algorithm that seeks to decrease the number of evaluated nodes in Minimax search tree. As described before, the time complexity of Minimax with branching

factor b and maximum search depth d is $O(b^d)$. When b is large, the approach to evaluate every nodes in tree becomes impractical, this is when alpha-beta pruning comes in handy.
The main idea behind alpha-beta pruning is to eliminate branches which are no longer promising. In practice, we maintain two value during searching, α: the best value along the path from root to maximizer, β: the worst value along the path from root to minimizer. Whenever $\alpha \geq \beta$, we prune the branch and stop further evaluating and searching.

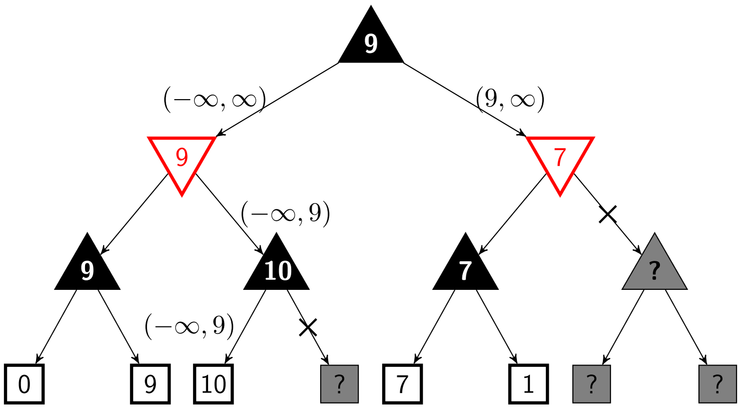


Figure 9: Minimax Tree Search with Alpha-Beta Pruning

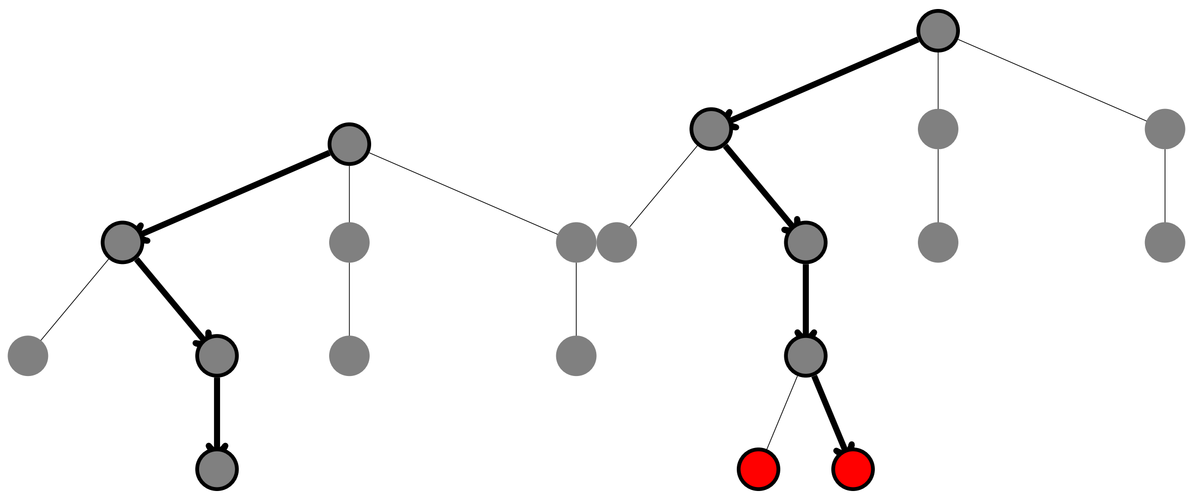
The $(-\infty, \infty)$ pair by the edge on Figure 8 is the (α, β) pair, the gray nodes are eliminated.

Algorithm 5 Alpha-Beta Pruning

```
procedure MINIMAX($s, depth, level, \alpha = -\infty, \beta = \infty$)
    if $depth = 0$ or $s$ is terminal then return evaluate($s$)
    if level is $Max$ then
        $best \leftarrow \infty$
        for each valid action $a$ do
            $s' \leftarrow$ successive state of $s$ taking $a$
            $best \leftarrow \max(best, Minimax(s', depth - 1, Min, \alpha, \beta))$
            $\alpha \leftarrow \max(\alpha, best)$
            if $\alpha \geq \beta$ then
                break
        return $best$
    else
        $worst \leftarrow -\infty$
        for each valid action $a$ do
            $s' \leftarrow$ successive state of $s$ taking $a$
            $best \leftarrow \min(worst, Minimax(s', depth - 1, Max, \alpha, \beta))$
            $\beta \leftarrow \min(\beta, worst)$
            if $\alpha \geq \beta$ then
                break
        return $worst$
```

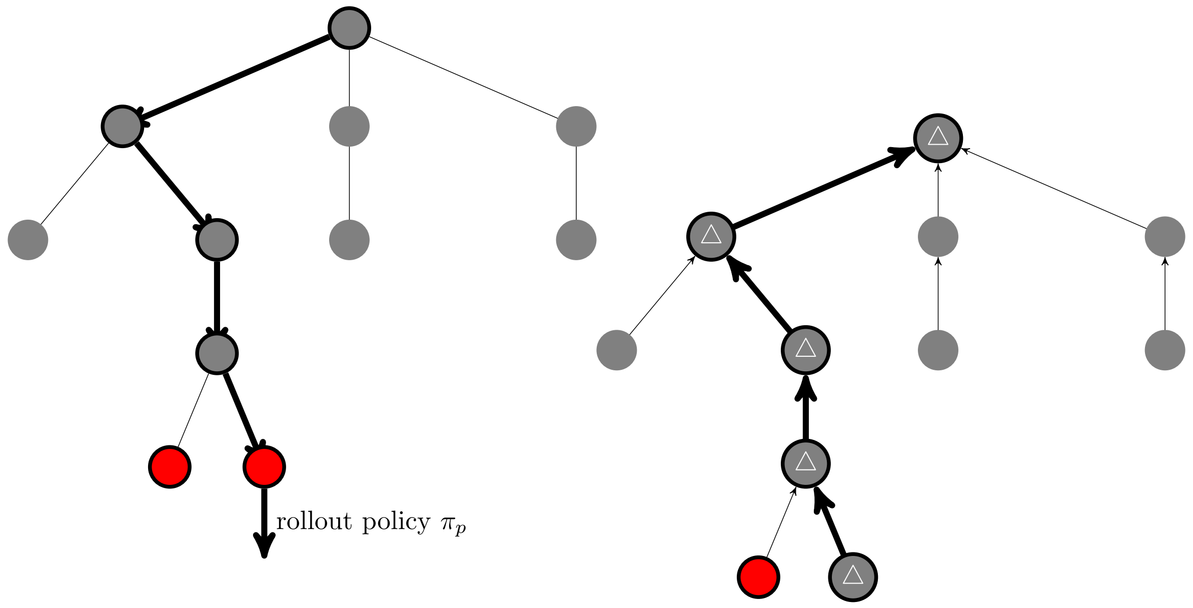
5.3 Monte-Carlo Tree Search

Although the alpha-beta pruning reduces the nodes being evaluated in the Minimax search tree, the time complexity of the algorithm is still overwhelmingly intractable for computing. Thus, a concept of averaging through simulations as a guidance on action selection has been developed, called Monte-Carlo tree search. The process of MCTS consists of four steps: selection, expansion, simulation and back-propagation. At each playout, we first select actions according to some fixed policy, when reaching the leaf node (i.e., unexplored node), we expand the node (i.e., add children to that node), then we simulate the game result following another fixed policy, which is called the rollout policy (e.g., randomized). Finally we back-propagate along the path from the leaf node to the root and update the value of these nodes using the simulated game result.



(a) Select (b) Expand

Figure 10: Monte-Carlo Tree Search



(a) Simulate (b) Backpropagate

Figure 11: Monte-Carlo Tree Search

6 Implementation

We choose to implement reinforcement learning and tree search on 3D-Bingo game. Two players take turn placing their cubes on the plate, whoever obtains a line with four cubes of their color at any direction or angle wins. So we end up with about 10^{20} states total, and at most 16 different actions can be performed at each turn. We implement our algorithm with Tensorflow, an open source numerical computation library using data flow graph, developed by Google.

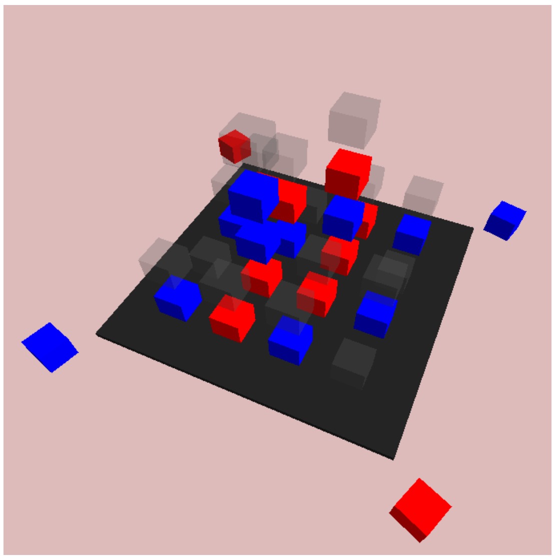


Figure 12: 3D-Bingo

6.1 Architectures

Our learning algorithm comprises two phases: value network training and policy searching. For value network, we use a fully-connected neural network, which takes the raw game board, pattern recognition of the board and one extra node representing the player, for that value may differs from player to player. The pattern of the board are captured from both current player and opponent: the number of cubes on the main diagonals (e.g., from $(0, 0, 0)$ to $(3, 3, 3)$), the number of promising two-cubed lines and the number of promising three-cubed lines, where the word *promising* means the probability of winning from this line. The number of hidden layers and the number of nodes in each hidden layers are user-defined, the output of the neural network is one real number, calculated as the weighted sum of the previous layer mapped by $tanh$ (ranging from -1 to 1, which is ideal for state evaluation).

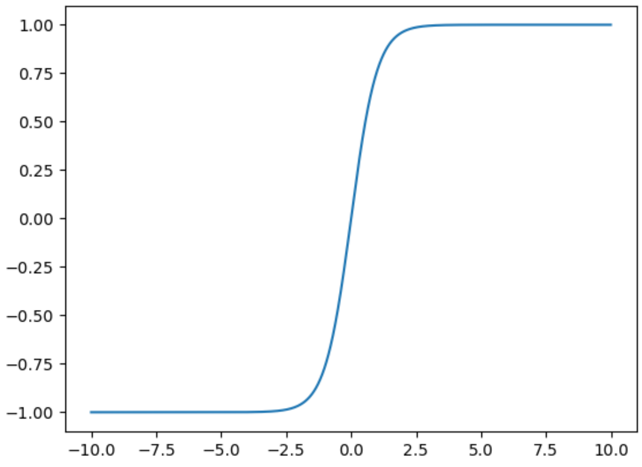


Figure 13: $tanh$

After the neural network is trained by supervised game record, we then apply reinforcement training by making self-play of the AI and update the neural net following temporal difference procedure.
As for policy search, we construct two types of AIs: one uses Minimax algorithm for action selections while the other uses Monte-Carlo tree search. The rollout policy applied to the MCTS

is randomized for both player. For playout node selection, we maintain the UCT value for each nodes, which is quite similar to the UCB (*upper confidence bound*) in bandit problems, UCT stands for the procedure which takes advantage of the UCB on tree search:

$$UCB1 = \bar{v_j} + \sqrt{\frac{2\ln n}{n_j}}$$

$$UCT = \bar{v_j} + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

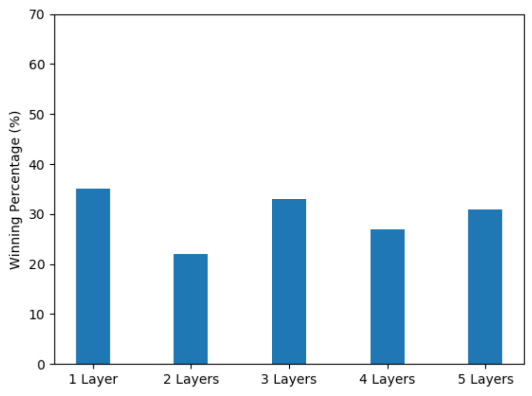
where $\bar{v_j}$ is the average value of node j, C_p is a constant value in $[0, \infty]$ which controls the exploration vs exploitation tradeoff, n is the number of visits of the parent node, n_j is the number of visit of node j. At each playout, we greedily choose the child node which gives the highest UCT, while the actual action selection is based on number of visits for that higher UCT leads to higher frequency of visiting. The calculation of v_j is a combination between state evaluation and eligibility trace, which is the TD(λ) algorithm in Chpter 3:

$$v_j = (1 - \lambda) \cdot V(s) + \lambda \cdot z$$

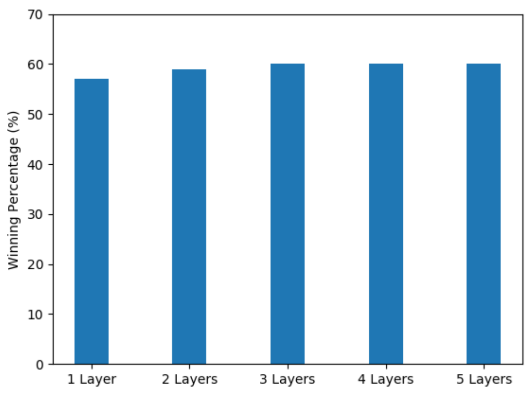
$V(s)$ is the state evaluation and z is the game simulation result (1 for victory and -1 of the opposite).

6.2 Results

In this section we'll evaluate the strength of AIs by its winning percentage against another simple game-playing bot. The bot make action selections following the procedure that, if there's action which leads to direct victory, that action will be made; otherwise, if there's action which prevents our AI from winning directly, the bot will stop our AI from making that action. If none of above holds, the bot will just arbitrarily takes some random action. The winning percentage is averaged for 1000 game playing.

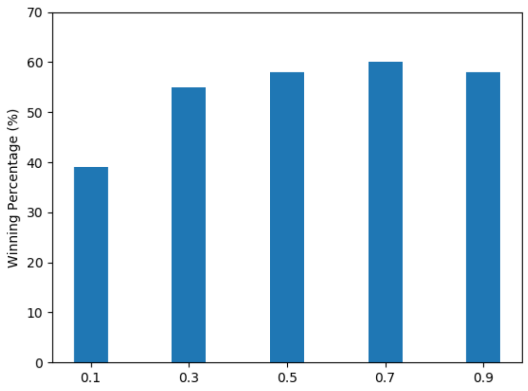


(a) Randomized rollout, 50 playout

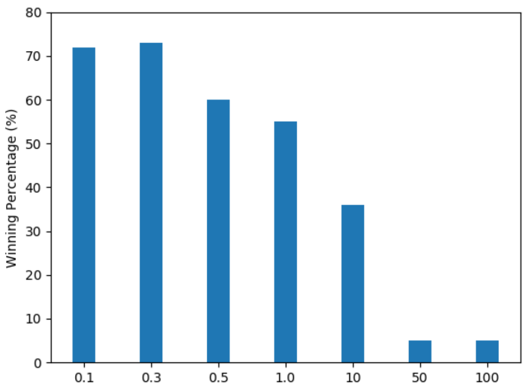


(b) rollout π_p, 20 playout

Figure 14: Winning percentage of MCTS with different layers and rollout policy, $\lambda = 0.7$ (π_p is the same policy adopted by testing bot)



(a) λ



(b) C_p

Figure 15: Winning percentage of MCTS with different λ and C_p

Based on the above results, we decide to conduct further experiments with an AI with 3-layer neural net which uses the same policy as testing bot and $\lambda = 0.7, C_p = 0.3$.

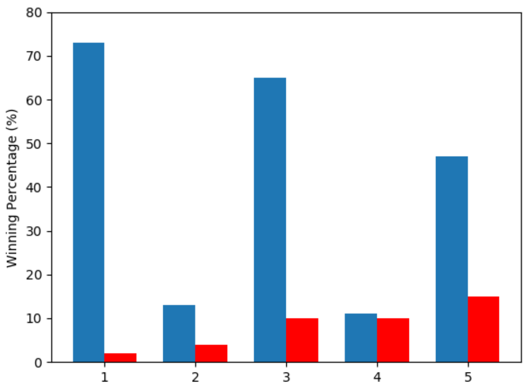


Figure 16: Minimax vs MCTS with different playout/search depth

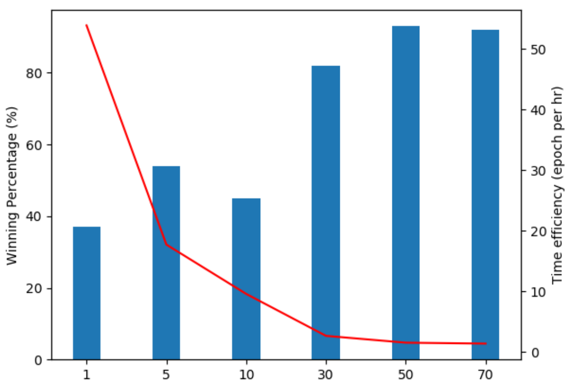


Figure 17: MCTS winning percentage and time efficiency

7 Discussion

On the basis of our results, the effectiveness of neural network is actually quite far from ideal, this reflects in the fact that higher λ leads to higher winning percentage, which means the low dependency on state evaluation is the key to victory. The potential causes include the insufficiency of the training data and the lack of computation power which brings about less epochs being made while the TD method converges when iteration goes to infinity.
The Minimax algorithm performs poorly in winning but is good at preventing opponent from taking the victory, thus, longer games are played by Minimax and more ties are made as well. As for MCTS, higher C_p refers to higher tendency of exploration, making the AI to choose *new* actions rather than those with good game simulation result, thus leading to poor performance. Another trivial fact is the positive correlation between number of playouts and winning, but the checks and balances of *good* move and *fast* move is still taken into further consideration.
The AI is still not capable of dominating the game, pattern recognition can be improved and a better neural network may be trained if we can obtain more, higher quality training data. Furthermore, optimization for implementation is yet another area of improvement. Generalization of the project is required in further research for conquering more complex games and solving real-world problems.

8 References

1. Csaba Szepesvari.*Algorithm for Reinforcement learning*
2. Jordan Boyd-Graber,Kevin Kwok, Hal Daume III.*Opponent Modeling in Deep Reinforcement Learning*
3. Henk Mannen.*Learning to play chess using reinforcement learning with database games*
4. Michael Gherrity.*A Game-Learning Machine*
5. Andrew Ng.*Shaping and Policy Search in Reinforcement learning*
6. Richard S. Sutton, Andrew G. Barto.*Reinforcement learning:An Introduction*
7. Google DeepMind.*Mastering the Game of Go with Deep Neural Networks and Tree Search*
8. Brian Tanner,Richard S. Sutton.*TD(λ) Networks:Temporal-Difference Networks with Eligibility Traces*
9. Marco A. Wiering,Hado van Hasselt.*Two Novel On-policy Reinforcement Learning Algorithms based on TD(λ)-methods*
10. Cameron Browne,Edward Powley,Daniel Whitehouse,Simon Lucas,Peter I. Cowling,Philipp Rohlfshagen,Stephen Tavener,Diego Perez,Spyridon Samothrakis,Simon Colton.*A Survey of Monte Carlo Tree Search Methods*