Reinforcement Learning: Final Project

Due on July 1, 2020 at 11:59pm

Professor Ziyu Shao

Yiwei Yang
2018533218

Problem 1

1. I'm currently a sophomore at CS major, having a deepmind to build something with Reinforcement Learning(RL) to make a world better place. For now, my interst focus on HFT and UAVs. Those territory can smoothly apply the RL thoughts by applying competetive MDP on reward updates by the surrounding environment. In the field of Control Systems, admittedly we have useful tools such as PID controllers. In my sense, if I wanted to make an agent to do certain movements to complete tasks, I'll apply PID, while if I wanted to make an agent to control a plane to shoot an certain targets, RL is better. So, RL really add a brilliant touch to the traditional automation Fields.

2. To make a conclusion to the course , I think SI252 RL really takes me into the door of exploration and exploitation. That's not only an outlook on life, but also a profound strategy to conquer the economics problems. However, during my research and study on Probability, Statistics, Stochastic Process, I self-studied a lot on how to utiliaze the language python to generate the RL code. I relearnt some of the Graph theory, Cryptography and Game thoery to get full understanding of a practical scenario.

3. For the shorts of the class:

   (a) More originated resources and more equation deduction.

   (b) Can add more talks on the Deep RL, because the balance between the math basics and the depth of the RL scenario is not well-considered.

Problem 2

The maximal-cut (max-cut) problem in a graph , as discussed in Goemans and Williamson (1995) Zhou et al. (2018), can be formulated as follows. Given a graph $G = G(V, E)$ with a set of nodes $V = \{v_1, \ldots, v_n\}$, a set of edges E between the nodes, and a weight function on the edges, $c : E \to R$. The problem is to partition the nodes into two subsets V_1 and V_2 such that the sum of the weights c_{ij} of the edges going from one subset to the other is maximized:

$$\max \left\{ \sum_{\substack{v_i \in V_1 \\ v_j \in V_2}} c_{ij} : V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset \right\}$$

$x = (x_1, \ldots, x_n)$, where $x_i = 1$ if node v_i belongs to same partition as v_1, and $x_i = 0$ else. For each cut vector x, let $\{V_1(x), V_2(x)\}$ be the partition of V induced by x, such that $V_1(x)$ contains the set of nodes $\{v_i : x_i = 1\}$. Unless stated otherwise, we let $v_1 \in V_1$ (thus, $x_1 = 1$)
Let X be the set of all cut vectors $x = (1, x_2, \ldots, x_n)$ and let $S(x)$ be the corresponding cost of the cut. Then,

$$S(x) = \sum_{\substack{v_i \in V_1(x) \\ v_j \in V_2(x)}} c_{ij}$$

When we do not specify, we assume that the graph is undirected. However, when we deal with a directed graph, the cost of a cut $\{V_1, V_2\}$ includes both the cost of the edges from V_1 to V_2 and from V_2 to V_1. In this case, the cost corresponding to a cut vector x is, therefore,

$$S(x) = \sum_{\substack{v_i \in V_1(x) \\ v_j \in V_2(x)}} (c_{ij} + c_{ji})$$

1. From of Science (2012) Dunning et al. (2018)' work, the gibbs sampling which is one part of the MCMC best fit the max cut problem.

Problem 2 continued on next page...

(a) Start with $\mu^0 = \overline{x}$ and $\delta^0 = \left(\delta_1^0, \ldots, \delta_p^0\right)^T$. Set $i = 1$

(b) Sample $\left(\Sigma^{-1}\right)^i$ from the Wishart distribution with scale matrix $\left(S_f + S + n\left(\overline{x}_f - \mu^{i-1}\right)\left(\overline{x}_f - \mu^{i-1}\right)^T\right)^{-1}$ and $v = n + 2t + p + 1 = N + n - 1$ degrees of freedom.

(c) Sample μ^i from a normal distribution with mean $g_{\delta^{i-1},\Sigma^l} = \left(\Psi_{\delta^{i-1}}^{-1} + n\left(\Sigma^{-1}\right)^i\right)^{-1}\left(\psi_{\delta^{i-1}}^{-1}\theta_{\delta^{i-1}} + n\left(\Sigma^{-1}\right)^i\overline{x}_f\right)$, and covariance matrix $V_{s^{i-1},\Sigma^i} = \left(\psi_{s^{i-1}}^{-1} + n\left(\Sigma^{-1}\right)^i\right)^{-1}$

(d) Define $\delta_j^i(r) = \left(\delta_1^i, \ldots, \delta_{j-1}^i, r, \delta_{j+1}^{i-1}, \ldots, \delta_p^{i-1}\right)^T$. For $j = 1, \ldots, p$, sample δ_j^i from the discrete distribution: $p(r) = \dfrac{\left|\psi_{s_j^i(r)}\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\left(\mu^i - \theta_{\delta_j'(r)}\right)^T \psi_{\delta_j^i(r)}^{-1}\left(\mu^i - \theta_{s_j'(r)}\right)\right\} p\left(\delta_j^i(r)\right)}{\sum_{q=-1}^{1}\left|\psi_{\delta_j^i(q)}\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\left(\mu^i - \theta_{\delta_j^i(q)}\right)^T \psi_{\delta_j^i(q)}^{-1}\left(\mu^i - \theta_{\delta_j^i(q)}\right)\right\} p\left(\delta_j^i(q)\right)}, r \in \{1\}$

2. The cross entropy method can be generally depicted as by Rubinstein and Kroese (2013):

(a) First, cast the original optimization problem of $S(x)$ into an associated rare-events estimation problem: the estimation of

$$\ell = P(S(X) \geq \gamma) = EI_{\{S(X)\geq\gamma\}}$$

(b) Second, formulate a parameterized random mechanism to generate objects $X \in X$. Then, iterate the following steps:

i. Generate a random sample of objects $X_1, \ldots, X_N \in X$(*e.g.* cut vectors.)

ii. Update the parameters of the random mechanism (obtained via CE minimization), in order to produce a better sample in the next iteration.

iii. Generation of cut vectors: The most natural and easiest way to generate the cut vectors is to let $X_2, \ldots, X_n$ be independent Bernoulli random variables with success probabilities $p_2, \ldots, p_n$

iv. Updating formulas: From CE minimization: the updated probabilities are the maximum likelihood estimates of the ρN best samples:

$$\hat{p}_{t,j} = \frac{\sum_{i=1}^N I_{\{S(X_i)\geq\hat{\gamma}_t\}} I_{\{X_{ij}=1\}}}{\sum_{i=1}^N I_{\{S(X_i)\geq\hat{\gamma}_t\}}}, \quad j = 2, \ldots, n$$

Formally, the algorithm can be written as:

3. Basic MCTS has the following procedures:

(a) Select

It is to select a node, at first, only the root node, no child nodes can be selected then skip to the next step, expand If there are child nodes, choose one based on a sample of the value of the child nodes, and then see if there are any child nodes left in the chosen one , if any, continue down the selection until you reach the leaf node. Then proceed to the next step.

(b) Expand

Once you have selected the leaf node, you can expand it to expand the children of the leaf node, either one or the other Multiple, depending on the situation. For example, the familiar alpha dog algorithm expands all the child nodes each time, and then gives each child a probability based on the neural network output Node assignment a priori probability, not much to say here, interested to understand the Alpha Dog paper. The normal MCTS, which is more common to expand one, is not essentially that different, because we record whether the node has been fully Expanded, nodes that are not fully expanded will continue to be expanded in the next simulation, so essentially there is little difference between expanding one and multiple as it will mostly unfold eventually.

Problem 2 continued on next page...

Algorithm 1 Cross Entropy Method on the Max-cut Problem

Input cost matrix C;
Initialize $\hat{p}_0 = (1/2, \ldots, 1/2), p[1] = 1, eps = 10^{-3}, Ne = 10$;
for $\max(\min(p, 1-p)) > eps$ **do**
  $x = (\text{rand}(N, D) < \text{ones}\ (N, 1) * p)$;
  $SX = S(x)$;
  sort SX = sortrows($[x\,\text{SX}], m + 1$);
  $p = \text{mean}(\text{sort}\,\text{SX}(N - \text{Ne} + 1 : N, 1 : m))$
end for
function S(x)
  $N = size(x, 1)$;
  for $i = 1$ to N **do**
    $V1 = find(x(i, :))$;
    $V2 = find(\ x(i, :))$;
  end for
  $perf(i, 1) = sum(sum(C(V1, V2)))$;

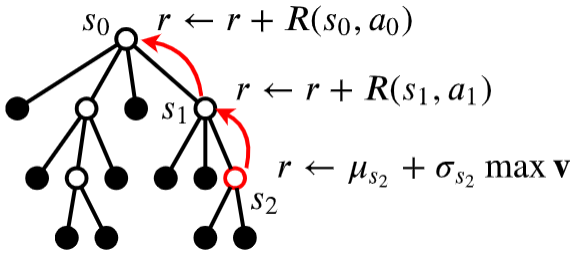


Figure 1: After estimating the reward of the expanded node, we iteratively update the rewards of its ancestors.

(c) Simulation

This is one of the more important steps in MCTS, and depending on a strategy, a random strategy usually works very well, borrowing from the Alpha Dog first author David Silver's quote from an intensive learning course at the University of London: don't assume random strategies is a very bad strategy, and it can often yield very good results. So here we generally just use a random strategy, starting with the leaf node we just expanded and simulating it until the end of the game. What exactly does that mean? It is from this leaf node situation that the game begins, with both players randomly landing from available spots until the game is won or lost, and this win or lost The outcome of the situation is to some extent a reflection of the situation, and if you win, you can at least have the kind of fallout to win once Isn't it, and of course we can imagine that this result is very unreliable, after all, random drop-offs are, in fact, really unreliable. But the advantage is that it's fast, and we can simulate it many times, thousands of times, so that even if it's random, if most of them win It's also enough to say that the situation is winnable, so essentially MCTS is an algorithm that approximates probability with frequency.

(d) Backup After simulating the win-loss result in the previous step, this result is 1, representing a win-loss draw, and then the result is returned. Update the value of the node on this path. For a concrete example, say the simulation results in a win, which is 1, then add 1 to the value of that leaf node in step 2. Of course there are other values that need to be updated, such as the number of selections should also be added by 1, and then the value of its parent node should be added

by -1 because it is The game is a win for the opponent, a loss for yourself, so take the opposite number, then the parent node is plus 1, and so on recursively to the root node to update the values of the nodes on the entire path.

The algorithm can be described as followed:

Algorithm 2 MCTS Method on the Max-cut Problem

Require Network f_θ ,root node s_0
Ensure Return π, Enhanced Policy
for $\sum_{a \in A_{s_0}} N(s_0, a) \leq c_{\text{iter}} \ |A_{s_0}|$ **do**
  $s = s_0$;
  {select};
  for s is expanded before and $s \notin S_{end}$ **do**
    $a = \underset{b \in A_n}{\operatorname{argmax}} \left(Q(s,b) + c_{\text{puct}} \ P(s,b) \frac{\sqrt{\sum_{b'} N(s,b')}}{1+N(s,b)} \right)$;
    $s \leftarrow T(s,a)$;
  end for
  {expand}
  if $s \notin S_{end}$ **then**
    $(p, v) = f_\theta(s)$;
    initialize $(N(s,a) = 0, W(s,a) = 0, Q(s,a) = 0, P(s,a) = p_a)$ for each $a \in A_s$
    calculate (μ_s, σ_s) by random sampling
  end if
{backup}
  $r = r_{estim(s)}$;
  for s is not s_0 **do**
    a = previous action $s \leftarrow$ parent of s $r \leftarrow r + R(s,a)$
    $r' = (r - \mu_s) / \sigma_s$
    $W(s,a) \leftarrow W(s,a) + r'$
    $N(s,a) \leftarrow N(s,a) + 1$
    $Q(s,a) \leftarrow \frac{W(s,a)}{N(s,a)}$
  end for
end for
Compute $\left(\pi_a = \frac{N(s_0,a)^{1/\tau}}{\sum_b N(s_0,b)^{1/\tau}} \right)$ for each a
return π

4. Pros and Cons of the method:

   All of those heurisitic algorithm is just the trade of the exploitation

   (a) For MCMC, the search method of maxcut is very similar to those in the queuing problem. The only difference is the reward and the live- death problem. So, those kind of problem can done by each transition updates a single variable of the sample(Gibbs Sampling). By this way, MCMC is scalable to be deployed on many questions and solve them without knowing the principles. However, combinatorial optimization often has a set of possible values for a variable can beseverely restricted by the value of others. Hence, the set of possible values for a variable can beseverely restricted by the value of others. At times,no single variable update is possible without violatingthe constraints, thus rendering the underlying Markovchain non-ergodic. Just like the C_MCMC case in solving maxcut, it stuck to local optimal value.

   (b) For CEM When adding experiments, the CE program will not obtain a better estimate of the cut by increasing the number of iterations, because it is not possible to obtain a better estimate of the

cut for a program with fixed The CE program with the number of samples is iteratively updated with the probabilistic parameter p as it reaches the optimal solution. The density of probability distributions describing the pattern of variables should be concentrated on the optimal solution, i.e., however randomly the samples are generated, The samples are all optimal solutions to the objective function, but due to the limited number of samples, the solution obtained by the CE program is This is only a partial optimal solution, so when the CE program obtains this partial optimal solution during the iteration, the CE program will not be able to solve the problem even if it is repeated. Increasing the number of iterations will not produce a better solution and the output will always be the same. This means that too many iterations of the CE program are not necessary.

(c) MCTS actually has a bunch of algorithm. The naive one is to use UCB1 to update the value function and then do back-play. In my test, MCTS is the faster than MCMC which manifest its profermance, slower than CE which implies it requires more information from the graph to update. Besides, we have Abe et al. (2019) using AlphaZero to solve the probblem which I didn't have the time to reproduce. The AlphaZero's self-play can be deployed on many models, which is very versatile and specific, but time consuming.

Problem 3

1. I have an apple(RL), I have an pen(DL), combine it, we get applepen(DQN) in Mnih et al. (2015).

   (a) Problem: how to perform Reinforcement Learning Reinforcement in high dimensional input situations. For example, in the game Atari, how to make the machine learn to control the target to make the game score the most points. The paper treats playing games as intelligences agent through a series of actions, observations, and The reward interacts with the environment (in this case, the Atari simulator). The internal state of the simulator is not available to the agent; the agent only gets the game screen and the corresponding score. Obviously the current state is not only dependent on the current game screen, but also on previous states and actions. The agent can use this to learn how to play the game, i.e. how to choose the current action that will result in the highest efficiency score in the future. In the Loss Function of the Q-network, we have $+\theta_i L\left(\theta_i\right)=E_{k,a,r,s'}\left[\left(r+\gamma \max_{d'} Q\left(s', a' ; \theta_i^{-}\right)-Q\left(s, a ; \theta_i\right)\right)+\theta_i Q\left(s, a ; \theta_i\right)\right]$ which can be solve by SGD or other DL methods. This kind of optimization's feature is model-free, since it is not fixed explicitly; also off-policy, ϵ- The greedy strategy lets ϵ fall linearly from 1 to 0.1 for the first 1 million times, and then stays constant at 0.1. The decay parameter is ϵ and the selection action parameter is 1 - ϵ. This starts out with a random search for Q updates and then slowly Use the optimal method.

   (b) Traditional procedures:

      i. Handcrafted features, i.e. manually extracted features (e.g. background modeling in the game). (extracting the target location);

      ii. using linear value equations or strategies for characterization.

   Due to the lack of robustness of manually extracted features, the performance of traditional methods depends mainly on how well the features are extracted; additionally, linear value The equations cannot simulate the nonlinearity in reality well.

   (c) Possible solution: The Deep-Q-Network proposed in this paper can solve the above problems, i.e., CNN + Q-Network. Learning = Deep Q Network is specifically a combination of convolutional neural networks and Q Learning is combined. The input to the convolutional neural network is the raw image data (as state) and the output is the value value corresponding to each action.

Problem 3 continued on next page...

Function to estimate future feedback Reward. i.e., use CNN to fit the optimal action valuation function (optimal action-value function).

(d) Technical Details(Steps):

i. Pre-processing of images

ii. When encoding the image, the image pixel is the maximum of the image pixel to be encoded and the image pixel of the previous frame, this is to eliminate the flickering phenomenon. Take the grayscale value of the RGB image, and resize to 84*84, take 4 frames (k=4 is the most robust) as input.

iii. Model architecture
Previous methods of deriving Q values used historical information and actions as input, which resulted in a Q value to be calculated for each action and computationally inefficient; the paper has only state as an input and the output is an estimate of the value of Q corresponding to each action, such that in the When calculating the Q values for different actions in a certain state, you only need to calculate the network forward once. The model is as follows

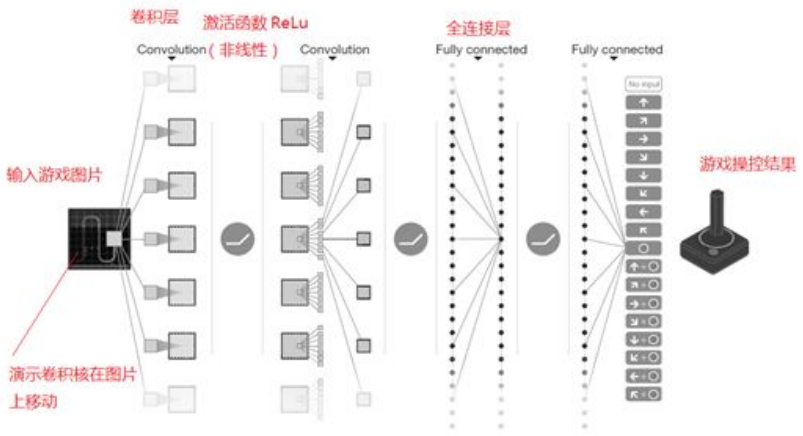


Figure 2: Graph Labeled for DGN network

iv. Details of the model

A. The previous approach uses different models for different games, the paper trains with only one model (based on a small amount of a priori knowledge, such as input images, game action types, number of lives, etc.), indicating that the paper's model is generalizable.

B. The rewards of the different games are normalized to 1 for positive, -1 for negative, and 0 for others. ratio and can be used to train different games using a uniform training speed.

C. Because of the large amount of training data and high redundancy or repetition, training with the RMSProp algorithm. The gradient descends by a look at the direction and a look at the step length, here using Divide the gradient by a running average of its recent magnitude. by introducing a decay coefficient ϵ such that the reward decays by a certain percentage each round. ϵ-greedy strategy makes ϵ preceded by 1 Millions of times linearly decreases from 1 to 0.1 and then stays constant at 0.1. This starts off with a random search for Q-value updates and then slowly uses the optimal method later. This approach is a good solution to the problem of premature endings in deep learning and is suitable for dealing with non-stationary targets, but introduces new parameters. The attenuation coefficient, ρ, still depends on the global learning rate.

Problem 3 continued on next page...

D. The reason for skipping four frames for all games is that it is more efficient to compute without affecting the result too much.

v. Algorithm

Algorithm 3 Deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for epispde = 1 to$M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1$ to T **do**
    With probability ε select a random action a_t otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{\alpha'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

(e) Experiments Result

i. As discussed in the previous work in Mikolov et al. (2013), they applied the algorithm to the Atari 2600 game, 6 out of 7 games tested exceeded the previous method and several exceeded the human level. In this paper they applied the algorithm to Atari 2600 games, 49 of them exceeded the human level, mainly because of the Fixed target Q-network change.

ii. The figure below shows that the training reward and the forecast of the reward fluctuate a lot, with very small changes in weights This can lead to dramatic changes in scores and strategy output. On the other hand, the variation in average Q is stable because the maximum value of Q is used in each Target calculation.

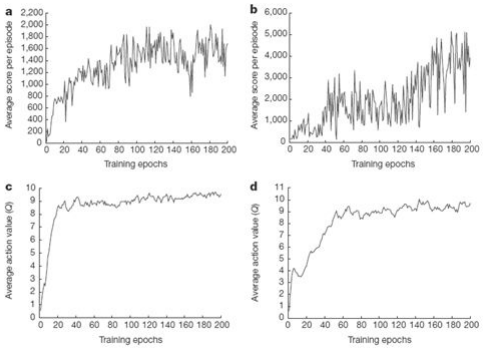


Figure 3: Result for DGN network

iii. Using the t-SNE algorithm to visualize high-dimensional data, showing the last hidden layer of the DQN to Square Enix Limited is an example of how similar states will be placed in close

proximity to each other. Sometimes the states may not be similar, but the desired rewards are similar (bottom half of Figure 3). The conclusion drawn is that the DQN network is able to learn from the high-dimensional raw input to support the representation of the adaptable rules.

(f) Summary of papers and future work

In this paper, DQN is used to solve the problem of game manipulation under high-dimensional image input.

i. The advantages of DQN are as follows.

A. Algorithms are more universal, the same network can learn different games (for different Atari games)

B. Adopt end-to-end training method, no need to manually extract Feature, use CNN instead of handcrafted features. Use a fixed length of historical data, such as 4 frames of images as a state input. Solve the problem that deep learning inputs must be of fixed length.

C. Using reward to construct labels through continuous test training, endless samples can be generated in real time for supervised learning.

D. through the experience replay method to solve the correlation and non-static distribution problems. That is, to establish a pool of experience, the experience of each time are stored, to train when the time to randomly take out a sample training.

E. The choice of action is based on the conventional ϵ-greedy policy. That is, the random action is chosen with small probability, and the random action is chosen with large probability. Probability selects the optimal action.

ii. The disadvantages of DQN are as follows.

A. Since the state of the input is short term, it is only suitable for dealing with problems that require only short term memory, and cannot deal with problems that require long term experience.

B. Using CNN to train may not necessarily converge, and requires fine settings of the network's parameters. (For example, I am doing target detection in my current project, using the model trained in imagenet as a pre-initialized network) parameters, which makes CNN making convergence not too difficult, and the paper is manually configuring CNN from scratch. (Also the Loss function is more complex.)

C. Because physical memory is limited, only N experiences are stored in the experience pool, the limitation of this method is that this experience pool doesn't distinguish between important transfer transitions, it always overwrites the latest transitions.

D. The input image pixels are still very low, how to scale to future complex games.

iii. Directions for improvement.

A. Use LSTM or RNN to enhance network memorability.

B. Improve Q-Learning algorithms to improve network convergence.

C. Train using preferred experiences to improve performance. I.e., guided pool of experiences; physically increase the capacity of stored data d.

D. Look for biological inspiration.

E. Changes in the structure of the CNN model need to be investigated, e.g. to see if the full connectivity layer can be replaced and if some of the tricks trained in other deep learning networks can be used here.

2. "Deep learning is a type of machine learning, it's a sophisticated assembly line where the whole pig is driven in from this side and the sausage comes out from that side on the It's okay." Silver first come up with MCTS algorithm to solve the Go Game in Silver et al. (2016), which defeats the Lee Seldo.

Problem 3 continued on next page...

(a) Problem: the Deep Blue defeat the top player in Chess in 1995, but based on brute force search so that the machine can easily defeat the people. When it comes to go, the search space of 391! does not accept full search. A good cut for the search is necessary. The MCTS is the good way to get optimal value in a limited time to converge.

(b) Some confusing ideas for researchers at that time:

i. Almost endless state space

ii. The MCTS tree search algorithm is invalid.

iii. The space for exploration must be narrowed.

iv. A human-like approach to chess.

(c) The most supprising contributions in this paper is applies MCTS and neural network on Go strategy.

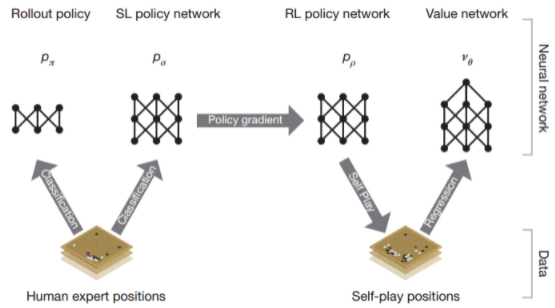


Figure 4: The basic procedures of the Alpha Go

i. This system, AlphaGo, consists of several main components.

A. policy network, to predict/sample the next move given the current position.

1. randomly picking a board (state/position) from the game

2. 30 million positions from the KGS Go Server (KGS is a Go website). Data is arguably the core, so it's still too early to say that AI has beaten humans, and AlphaGo is still standing at the human The expert advances on his shoulder. The board is treated as a 19x19 black and white binary image, then with convolutional layers (13 layers). This is better than the image. rectifier nonlinearities

3. Output all legal moves

4. raw input accuracy: 55.7%. all input features. 57.0%. Later methods have a mention of exactly what features. A bit of Go knowledge is needed, e.g., liberties means gas.

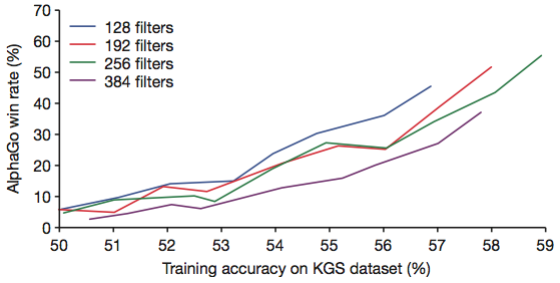


Figure 5: the training accuracy of policy network

B. fast rollout, with the same objectives as 1, but at an appropriate sacrifice of quality, which is 1000 times faster than 1.

1. Non-linear $->$ Linear
2. Local Features $->$ Full Board

Accuracy drops to 24.2%, but time 3ms-¿ 2μs. earlier MCTS mentioned the need to go to the bottom of the assessment The advantage of speed is evident.

Extended Data Table 4 | Input features for rollout and tree policy

Feature	# of patterns	Description
Response	1	Whether move matches one or more response pattern features
Save atari	1	Move saves stone(s) from capture
Neighbour	8	Move is 8-connected to previous move
Nakade	8192	Move matches a *nakade* pattern at captured stone
Response pattern	32207	Move matches 12-point diamond pattern near previous move
Non-response pattern	69338	Move matches 3×3 pattern around move
Self-atari	1	Move allows stones to be captured
Last move distance	34	Manhattan distance to previous two moves
Non-response pattern	32207	Move matches 12-point diamond pattern centred around move

Features used by the rollout policy (first set) and tree policy (first and second set). Patterns are based on stone colour (black/white/empty) and liberties (1, 2, ≥3) at each intersection of the pattern.

Figure 6: The feature of fast rollout policy

C. Policy Networks

1. The result of the preceding policy networks is taken as the initial value $\rho = \mu$
2. Randomly selects the policy network from a previous round to duel against to reduce overfitting.
3. $z_t = \pm 1$is the final winner. After the winner is determined, the direction of the gradient is optimized for each previous move, with a win increasing the predicted p and a loss decreasing p. The final objective is the win, while the original SL Policy Networks predicts the win.
4. The corrected final objective is the win, while the original SL Policy Networks predicts The same accuracy as the EXPERT walk. So the duel result 80%+ wins SL. , given the current situation, estimates whether a white or black win.

D. Value Network, given the current situation, estimates whether a white or black win.

Determine what the probability is that black or white will each win on a given surface, so the only parameter is s. Of course, you can also enumerate a to get $p(a|s)$. Of course, you can also enumerate a and get $p(a|s)$. The difference is that you know the relative probability of winning for each side.

1. using policy p for both players (Difference RL Policy Network: Random in front) (A P and the latest P vs.
2. $v_\theta(s) \approx v^{P_\rho}(s) \approx v^*(s) \cdot v^*(s)$ is the fraction obtained by the theoretically optimal lower method. It is obviously impossible to obtain, but only with our current strongest P_ρ algorithm to approximate. But this is not known until the end of the walk, so we have to use Value Network $v\theta(s)$ to learn about it.

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}\left(z - v_\theta(s)\right)$$

(The above equation should be $\min(z - v_\theta(s))^2$, and converting it to max removes the negative sign of the derivation) Since the preceeding hyphen is strongly correlated, enter Only one piece is different, z is the final result, which is always the same, so directly counting like this will overfitting. The solution is to extract only the position from the game. The solution is to extract only the position of the game, which actually generated 30 million distinct positions. that's how many innings of game there are.

By add the value network, the MSE of teh trainingset and test set get close. And if assembling the components as MCTS, the strength adds up.

Problem 3 continued on next page...

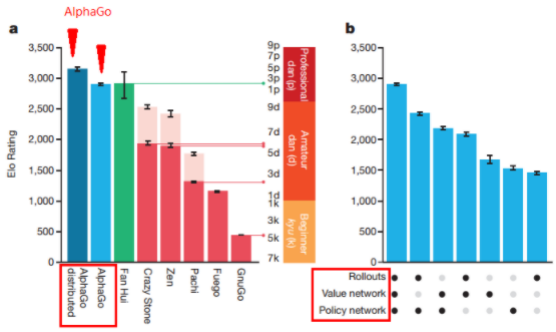


Figure 7: The Components of the Alpha Go compared to others

E. The Monte Carlo Tree Search (MCTS), to put the These three parts are connected to form a complete system.

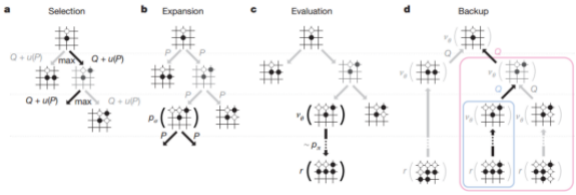


Figure 8: The MCTS of Alpha Go

The choicde of MCTS Simulation:

$$\begin{aligned} a_t &= \arg\max_a \left(Q\left(s_t, a\right) + u\left(s_t, a\right)\right) \\ &= \arg\max_a \left(Q\left(s_t, a\right) + C * \frac{P\sigma}{1 + \text{Search Times} N(s, a)}\right) \end{aligned}$$

, where $V(\theta_L)$ is the evaluated value of the leaf node and Q is the expected $V(\theta_L)$ after many simulations. Interestingly, the experimental result $\lambda = 0.5$ is the best. $V\left(\theta_L\right) = (1-\lambda) v_\theta\left(s_L\right) + \lambda z_T$, where v_θ is value network, fast rollout go to end result z_L

If you haven't qqexpanded Q to 0 in the first place, then SL's Pσ is prior It also reduces the width of the search, and the average point score is very low. It is harder to be SEECTED. Interesting conclusion, the comparison comes out better with SL than RL here! SL results that mimic human chess moves are more suitable for MCTS search because humans choose a diverse beam of promising moves. while RL's learned is the optimal lower method (whereas RL optimizes for the single best move). So mankind is winning at this point for now! But on the other hand, RL learned value networks work well for evaluation. So each has its own strengths.

N a lot of searches will deduct points, encourage exploration other branches.

ii. Experiment: They can have the professional level even with a single machine. Google is willing to apply few thousands of machine to train for the competition with Lee Seldo, it will of course win the game only to modify few vairables.

Problem 3 continued on next page...

Extended Data Table 1 | Details of match between AlphaGo and Fan Hui

Date	Black	White	Category	Result
5/10/15	Fan Hui	*AlphaGo*	Formal	*AlphaGo* wins by 2.5 points
5/10/15	Fan Hui	*AlphaGo*	Informal	Fan Hui wins by resignation
6/10/15	*AlphaGo*	Fan Hui	Formal	*AlphaGo* wins by resignation
6/10/15	*AlphaGo*	Fan Hui	Informal	*AlphaGo* wins by resignation
7/10/15	Fan Hui	*AlphaGo*	Formal	*AlphaGo* wins by resignation
7/10/15	Fan Hui	*AlphaGo*	Informal	*AlphaGo* wins by resignation
8/10/15	*AlphaGo*	Fan Hui	Formal	*AlphaGo* wins by resignation
8/10/15	*AlphaGo*	Fan Hui	Informal	*AlphaGo* wins by resignation
9/10/15	Fan Hui	*AlphaGo*	Formal	*AlphaGo* wins by resignation
9/10/15	*AlphaGo*	Fan Hui	Informal	Fan Hui wins by resignation

The match consisted of five formal games with longer time controls, and five informal games with shorter time controls. Time controls and playing conditions were chosen by Fan Hui in advance of the match.

Figure 9: The Result of the Alpha Go

iii. Some method to come up to improve the processure.

A. When Alphago's MCTS does rollout, it uses a fast walker as well as an existing search tree. In part, it looks like AMAF/RAVE in reverse: AMAF is the information that conducts the fast walker to other irrelevant parts of the tree. In part, Alphago is using other unrelated parts of the tree to enhance fast moves. I doubt that this is one of the reasons why it's stronger than other DCNN+MCTS.

B. "The better quality of rollout moves may lead to a decrease in chess power."
There are two cases to distinguish here, tree policy and default policy. in the AlphaGo's article, the distribution of tree policy should not be too sharp, otherwise the search for the If you pay too much attention to some seemingly good moves at times, you may lose your power. But apart from this reason, in general it's better to have a tree policy.
On the default policy side, i.e. (semi-)random moves to the end and then judging the score, it's more complicated and the quality gets better. All default policy needs to ensure is that each piece is dead or alive in roughly the right way. Don't play dead chess to live or vice versa, but rather less demanding of the big picture. It is perfectly possible for both sides to play each piece in tandem and then move on to another piece, rather than, say, taking the lead elsewhere before the other side.

3. Silver et al. (2016) says by telling the machine only the basic rules of the game, but not Go tactics such as stereotypes that humans have been fumbling with for thousands of years, let the Machines rely entirely on self-learning to beat humans. The title is not only fresh, it's hot.

(a) The core ideas: AlphaGo Zero differs from AlphaGo Fan and AlphaGo Lee. In the following areas.

i. The first and most important point is that AlphaGo Zero only uses reinforcement learning for training.AlphaGo Zero learns from the whiteboard, without using human supervised data, and trains for reinforcement learning in a self-playing game.

ii. Second point: uses only Othello on the board as input features. No artificially designed features of the Go domain are required.

iii. Third point: Use only a single neural network. No use of separate strategy and value networks.

iv. Fourth point: Use a simple tree search that relies only on a single neural network as described above, and perform both board situation evaluation and Move Move Selection. Monte Carlo Rollouts (Rollout Policy in AlphaGo) is abandoned.

To achieve these goals, this paper develops a new reinforcement learning training algorithm that incorporates Monte Carlo tree search trees into the training phase. The new algorithm results in faster performance improvement and more accurate and stable learning.

(b) Technical details:

Problem 3 continued on next page...

i. Policy Iteration:

A. Network structure: deep residual neural network f_θ, parameter θ. Each convolutional layer consists of many residual blocks, using batch Normalization and rectifier nonlinearities.

B. Input layer: Raw Board Representation of the position and its history. only the Othello features of the chess surface are used.

C. The output layer: $(p, v) = f_\theta(s)$, the probability distribution and prediction of the next move. pa is a vector of The probability distribution of the next move p(a—s), v is a value representing the probability that the currently playing party will win. Role: Acts as both a strategy network and a value network.

D. Training: a deep neural network in AlphaGo Zero using reinforcement learning algorithms in a self-opposed game Training. For each board s, the neural network $f\theta$ obtained in the previous round is used to guide the MCTS Monte Carlo tree search. The Monte Carlo search tree outputs a probability distributionπ for each move, called search probability searchprobabilities. search probabilities are better distributed than the output p-probability of the neural network alone. able to select stronger moves. From this point of view, MCTS can be viewed as a strategy boosting operation in the Policy Iteration algorithm (policy improvement operator). This means that the improved, Monte Carlo search-tree-based strategyπ is constantly used to make a decision move during the self-playing process , which will ultimately determine the winner as a sample value of the eventual winner z, can be seen as Policy Policy evaluation in Iteration operator), an assessment of the situation on the current board, as a prediction of the probability of winning the game on the current board. The key to the reinforcement learning algorithm here is to repeat during policy iteration These search operations, policy improvement and policy evaluation. after the Monte Carlo search, update the neural network parameters so that the output of the updated neural network : moving probability and predicted value $(p, v) = f\theta(s)$, which can more closely approximate the search probability and self-game obtained by MCTS of the winning side (π, z). This new neural network will continue to guide the Monte Carlo search in the next round of self-opposition, making the search more robust.

E. MCTS search simulation process: a neural network $f\theta$ is used to guide the simulation process. Each edge of the search tree stores the prior probability $P(s, a)$, the number of visits $N(s, a)$ and the action benefit value $Q(s, a)$.

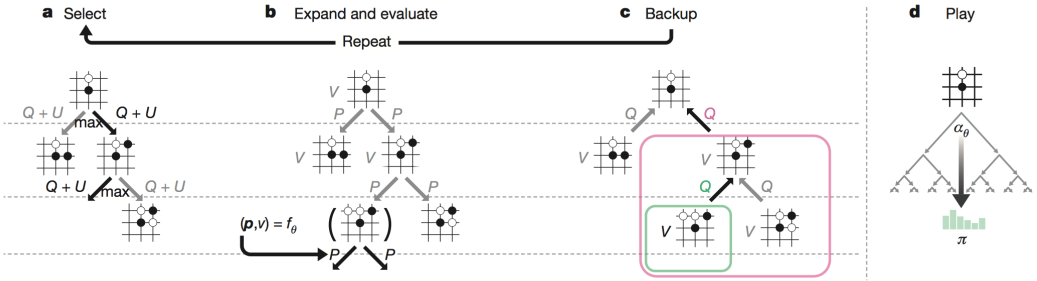


Figure 2 | MCTS in AlphaGo Zero. a, Each simulation traverses the tree by selecting the edge with maximum action value *Q*, plus an upper confidence bound *U* that depends on a stored prior probability *P* and visit count *N* for that edge (which is incremented once traversed). **b**, The leaf node is expanded and the associated position *s* is evaluated by the neural network $(P(s, \cdot), V(s)) = f_\theta(s)$; the vector of *P* values are stored in the outgoing edges from *s*. **c**, Action value *Q* is updated to track the mean of all evaluations *V* in the subtree below that action. **d**, Once the search is complete, search probabilities $\boldsymbol{\pi}$ are returned, proportional to $N^{1/\tau}$, where *N* is the visit count of each move from the root state and τ is a parameter controlling temperature.

Figure 10: The diagram of the Alpha Zero

Problem 3 continued on next page...

F. Data generation (self-playing) process: using the reinforcement learning network from the previous round to guide Monte Carlo during each game of self-playing Search the tree to play chess. For each time step t of the game on the board s_t, the Monte Carlo search simulation process generates a probability distribution π_t for the next move, thus can generate data pairs (s_t, π_t). The game is played until the moment at time step T, when the game is decided and the payoff r_T is obtained. r_T is then returned and labeled as a The preceding time step $t(< T)$ produces data pairs (s_t, π_t, z_t) on z_t, where $z_t = \pm r_T$, $\pm$ The symbols are determined by the corresponding moves of the current game face. From this, one game can generate many training data (s_t, π_t, z_t). During training, s_t is the input to the neural network: the features extracted from the current board, (π_t, z_t) are the labels of the sample s_t, The output (p, v) of the neural network is to be fitted to that label.

G. Optimization process: the neural network is first initialized using a random weight θ_0. In each subsequent iteration of Iteration $i \geq 1$ (each iteration corresponds to a number of complete games of Go), the Neural networks are optimized for training using Policy Iteration, a reinforcement learning algorithm that will The Monte Carlo search tree is incorporated into the training process using the previous round of the neural network $f_{\theta i-1}$ at each time step t of that round of iteration i to Instruct the Monte Carlo search $\pi_t = \alpha\theta_i - 1(s_t)$ until time step T produces a winner, constructing the round as described above The data generated by the game (s_t, π_t, z_t). Next, training data (s, π, z_t) is obtained by normal random sampling from the data generated at all time steps of the round, and the neural network Using this training data for parameter updates, the goal is to enable the output of the new neural network $(p, v) = f_{\theta i(s)}$ to be fit (π, z), i.e., minimizing the error between v and z and maximizing the similarity between p and π. The optimization loss function is as follows, combining MSE and cross-entropy loss (which also includes a regularization term). The optimization loss function is as follows, combining MSE and cross-entropy loss (also including the regularization term).

$$\ell(\theta) = (z - v)^2 - \pi^T \log p + c\|\theta\|^2$$

ii. The training process consists of three parallel core subprocesses: first from the 500,000 board data generated so far by the α_{θ_*} Medium-normal sampling followed by random gradient descent optimization (mini-batch:2048; impulse parameter:0.9 ; regularization parameter:10-4.)

Next, every 1000 rounds, a checkpoint is set for evaluation. The current best AlphaGoZero α_{θ_*} and the current round of AlphaGoZero α_{θ_i} are performed 400-game matchup. (1,600 MCTS simulations per move)

Finally, Self-play. using the best current AlphaGoZero α_{θ_*} to generate a self-playing game Data. 25,000 complete games are played in each iteration, using 1600 MCTS simulations per move. In the first 30 moves of each game, the temperature parameter is set to $\tau = 1$, and the probability of a move is proportional to the number of visits to ensure the diversity of moves. Subsequent step $\tau \to 0$, the probability of the action with the most visits tends to be 1, and the remaining probability is 0. In addition, to increase the probability of exploration exploration, adding the Dirac noise $P(s, a) = (1 - \epsilon)$ to the root node s_0 a priori probability $p_a + \epsilon_{\eta_a}$, where the parameter value is $\eta\ Dir(0.03), \epsilon = 0.25$.

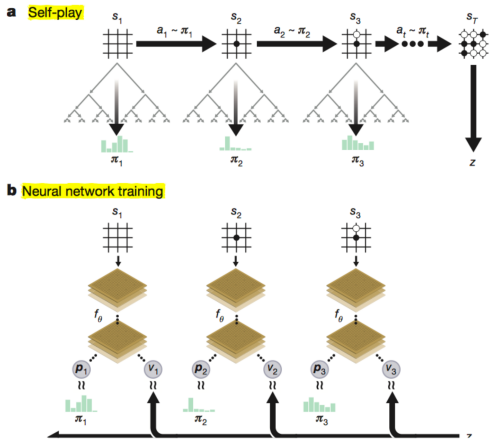


Figure 11: The training set of the neural network in Alpha Zero

(c) Comparative experiment

The reinforcement learning pipeline approach was used to train AlphaGo Zero. The training started with completely random behavior and lasted for 3 days, without human intervention. During this process, a total of 4.9 million self-playing chess games were generated, with 1600 simulations performed per Monte Carlo search. The corresponding average cost per move is 0.4 s. The parameters of the neural network use 700,000 mini batches, each with The mini batch contains 2048 planes to be updated and the residual neural network contains 20 residual blocks. residual blocks.

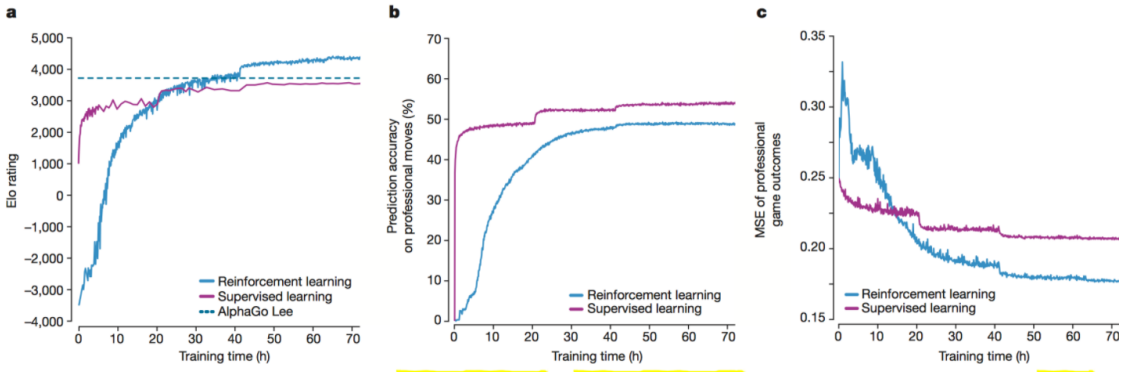


Figure 3 | Empirical evaluation of AlphaGo Zero. a, Performance of self-play reinforcement learning. The plot shows the performance of each MCTS player α_{θ_i} from each iteration i of reinforcement learning in AlphaGo Zero. Elo ratings were computed from evaluation games between different players, using 0.4 s of thinking time per move (see Methods). For comparison, a similar player trained by supervised learning from human data, using the KGS dataset, is also shown. **b**, Prediction accuracy on human professional moves. The plot shows the accuracy of the neural network f_{θ_i} at each iteration of self-play i, in predicting human professional moves from the GoKifu dataset. The accuracy measures the percentage of positions in which the neural network assigns the highest probability to the human move. The accuracy of a neural network trained by supervised learning is also shown. **c**, Mean-squared error (MSE) of human professional game outcomes. The plot shows the MSE of the neural network f_{θ_i} at each iteration of self-play i, in predicting the outcome of human professional games from the GoKifu dataset. The MSE is between the actual outcome $z \in \{-1, +1\}$ and the neural network value v, scaled by a factor of $\frac{1}{4}$ to the range of 0–1. The MSE of a neural network trained by supervised learning is also shown.

Figure 12: The result of the Alpha Zero

Testing AlphaGo Zero using KGS Server Dataset human expert chess playing data Predict the ability of human experts to play chess. That is, to test it, enter the data from the human chess playing surface, output the next move, and see how it compares to actual human play Whether it is the same, calculate the prediction accuracy. As a comparison, plot the predictive power of supervised learning, using the same network and algorithm, but the data is not generated by self-gaming of, but rather from human data. As shown in Figure above, it can be seen that in terms

of predicting the ability of human experts to play chess, the reinforcement learning of AlphaGo Zero algorithm is weaker than the supervised learning algorithm. It is worth noting that while AlphaGo Zero is slightly weaker at predicting human experts' ability to play chess, its Go ability and the The ability to predict eventual wins and losses, however, outperforms supervised learning, as can be seen in the MSE plot in Figure 3c below. This shows that AlphaGo Zero learned certain Go strategies on its own, and these strategies are qualitatively different from the way humans play Go.

(d) The possible method of improving:

i. Actually using a single network enables both Expansion and Evaluation. The value network in AlphaGo is also capable of indirectly doing Expansion (selecting actions with large values).

ii. Here AlphaGo Zero reinforcement learning uses Policy Iteration, and the Policy Gradient is used in AlphaGo. policy Iteration is similar to Q-learning in that it is value-based reinforcement learning. algorithm with the goal of maximizing the Bellman equation, Policy Iteration maximizing the Bellman expectation equation, Q -learning maximizes the Bellman optimal equation. The final result is a deterministic policy (near-deterministic policy). In contrast, Policy Gradient is a policy-based reinforcement learning algorithm that directly optimizes the The strategy function, a function estimation idea where the optimization method maximizes the desired output based on the reward payoff, will eventually yield a strategy Distributed functions can learn random polices over a large policy space.

iii. I think the AlphaGo Zero Monte Carlo search yields what would have been a deterministic strategy (policy Iteration is still essentially an optimal bellman equation) with a final strategy distribution π. It was actually only calculated by Monte Carlo searching for the number of visits to the simulation process nodes, see above Monte Carlo Play stage. Alternatively, one can think of AlphaGo Zero's reinforcement learning algorithm as a combination of value based and combination of policy based methods.

iv. I think the difference between the Value based approach and policy based is equivalent to the difference between maximum likelihood estimation and the Bayesian estimation of the difference, maximum likelihood estimation is based on a best solution for the training sample provided by the designer; Bayesian estimation is a weighted average of many feasible solutions, reflecting the degree of uncertainty about the models used Remaining Uncertainty. Looking back at AlphaGo's use of the Policy based method, the Policy Gradient, AlphaGo Zero uses the Value based method of I would venture to guess that the reason here is that the Monte Carlo search process introduces training. process, making the policy obtained by the Value based approach significantly more deterministic and reliable, without the need for excessive Consider residual uncertainty. In addition, the Value based method here is fast to train and does not require significant resources to ensure convergence speed . Of course, AlphaGo Zero ultimately builds on Policy Iteration and still uses the The probability distribution of the Monte Carlo search tree output strategy also shows the trade-offs and combinations of the two approaches.

v. We can also improve core elements like Policy Iteration, MCTS. This includes single improvements as well as improvements in the way multiple elements are combined. For example, the design of the final payoff function rT in the reinforcement learning algorithm, whether there is a better network architecture in deep learning, such as SENet etc., can Monte Carlo search trees be used in deep learning algorithms in addition to reinforcement learning algorithms to guide the back propagation of errors, etc. ; whether there is a better way to calculate action probability distributions after Monte Carlo simulations are completed. On the other hand, of course, the question of whether there are new areas that can replace the centrality of the Monte Carlo search tree, and whether forward-looking questions are only It takes a lot of

simulation to get feedback, is there another way to do it better.

4. For Silver et al. (2018) "I can't disguise my satisfaction that it plays with a very dynamic style, much like my own!"

   (a) The most enlightning part, Alpha zero can be deployed in any interllegent system in the society. For human chess:

      i. In chess, AlphaZero defeated the 2016 TCEC (Season 9) world champion Stockfish, winning 155 games and losing just six games out of 1,000. To verify the robustness of AlphaZero, we also played a series of matches that started from common human openings. In each opening, AlphaZero defeated Stockfish. We also played a match that started from the set of opening positions used in the 2016 TCEC world championship, along with a series of additional matches against the most recent development version of Stockfish, and a variant of Stockfish that uses a strong opening book. In all matches, AlphaZero won.

      ii. In shogi, AlphaZero defeated the 2017 CSA world champion version of Elmo, winning 91.2% of games.

      iii. In Go, AlphaZero defeated AlphaGo Zero, winning 61% of games.

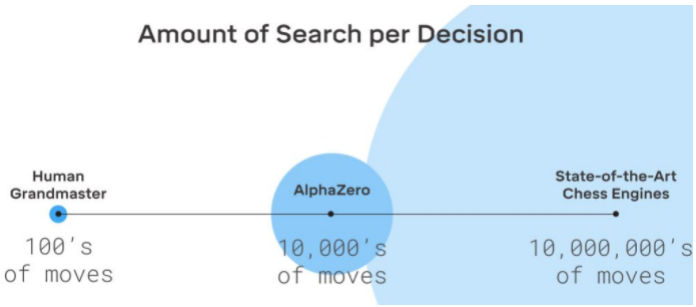


Figure 13: The diagram of the advancement with time

Deepmind applies 5000 TPU1 to train few hours can beat human power, and even machine based on the human knowledge: Stockfish.

This manifest the scalability of the AlphaZero algorithm, which mentioned in the paper, itcan be applied in Intelligent healthcare, Smart driving, NLP and Robotics

   (b) I think the paper does not make any more advancement in the thoery. Just a little remedy. MCTS is already a very scalable algorithm.

   (c) The experiment between AlphaZero and Stockfish, I think just release the result is not convincinig for the conclusion that AI is unbeatable by minmax. This is actucaully the difference between machine and human. Admittedly, MCTS can do pattern matching search and solve anything, sounds like brute and elegant conquer of human knowledge. The subtle knowledge of chess unearthed by human masters over 1000 years is not needed, nor is the subtle pruning search constructed by algorithm masters, nor should it be any opening library residue library? Some people think it is because the calculating power of Stockfish is not good, but the truth is the runtime AlphaZero just need to search 70,000 times per sec compared to 80,000,000. The problem is the opening of the game. Because the first few step contains most space and least calculable. The perspective of Alphazero is probability while Stockfish is search based on knowledge. The result shows all the problems are triggered by first 15 step. and though the chess is easy to get toe, the Stockfish still can't get to toe from post-15 steps.

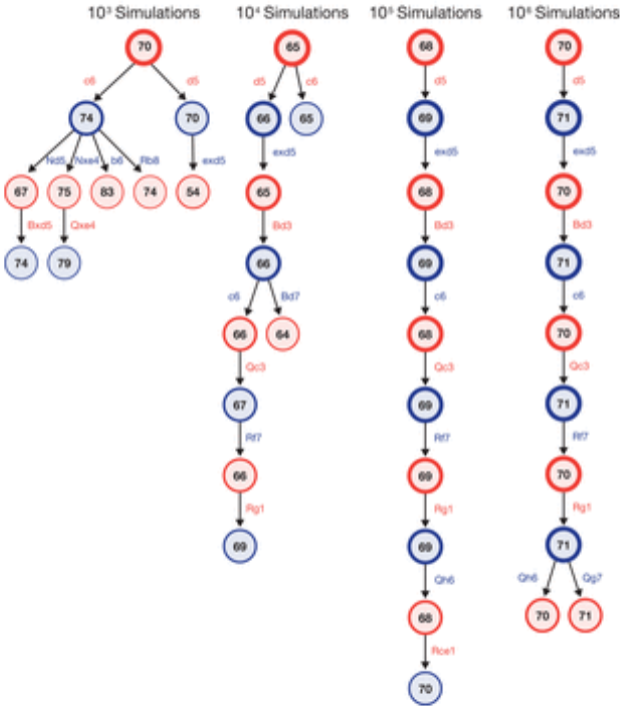


Figure 14: This search tree is also suitable to Stockfish

The problem is SF has calculated a whole bunch of variations, but the vast majority of them are just shipping pieces back and forth and then giving valuations based on the number of pieces in the force, with no actual plan. It's no surprise that one of the big reasons why players previously thought that maneuvers were better than pure soft was that SFs didn't address closed positions. The search algorithm does not have good performance in the first 15 steps is the key problem. (https://zhuanlan.zhihu.com/p/32166417)

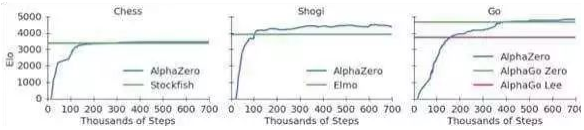


Figure 15: My thoughts of not good experiments

5. I prepared to take **?** & **?** together to talk, because they are basically the same thing. Imperfect-Information Games is Texas Deck. The main point is that A game of incomplete information is a game in which players have no common knowledge of the game being played, i.e., it cannot be played only through sub-games of information to solve the subgame, which is usually not globally optimal. In related studies, it is common practice to approximate the game's Nash equilibrium strategy, i.e., each player cannot solve the subgame by changing only his own of strategies to improve their own returns. In a zero-sum game, if each player chooses the optimal strategy against his opponent, then the game as a whole achieves Nash equilibrium, with each The player's strategy is the optimal strategy (Nash equilibrium strategy).

   (a) DeepStack's core algorithm is called Continual Re-solving, which is repeatedly traversing the Searching the tree and using Counterfactual Regret Minization (CFR)) algorithm continuously

optimizes the strategy and evaluates the nodes that exceed the specified depth in combination with value network. Its partial search tree is shown in the figure below.

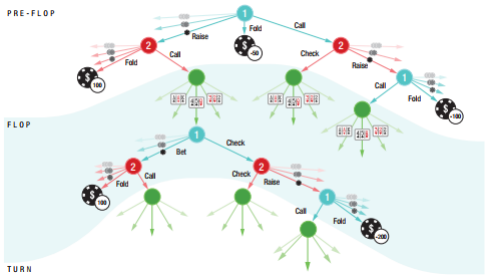


Figure 16: The diagram of Continual Re-solving

(b) Specifically, DeepStack does not save previous strategies and calculates them instantly each time it needs to act. For calculating the corresponding policy in any public state, you only need to provide your own range and your opponent's counterfactual values are sufficient, i.e. they are sufficient to summarize the current state of the All information that is known to the player. Note that the counterfactual values here are not one value, but a series of values corresponding to the In order to calculate the strategy on the fly. We need to track our own range and our opponent's counterfactual from the start of the game. values, at the beginning of the game, range is initialized to uniform distribution, while counterfactual values are initialized to each type of As the game progresses, you need to keep updating their values.

i. When acting on your own: update counterfactual values according to the calculated strategy; update range according to the strategy and Bayes' law

ii. On the flop: update counterfactual values according to the last strategy; if the flop comes with four identical cards, set the probability of the corresponding card in the range to zero.

iii. When the opponent acts: No need to update.

function RE-SOLVE($S, \mathbf{r}_1, \mathbf{v}_2, T$)
  $\sigma^0 \leftarrow$ arbitrary initial strategy profile
  $\mathbf{r}_2^0 \leftarrow$ arbitrary initial opponent range
  $R_G^0, R^0 \leftarrow \mathbf{0}$ ▷ Initial regrets for gadget game and subtree
  for $t = 1$ to T **do**
    $\mathbf{v}_1^t, \mathbf{v}_2^t \leftarrow$ VALUES($S, \sigma^{t-1}, \mathbf{r}_1, \mathbf{r}_2^{t-1}, 0$)
    $\sigma^t, R^t \leftarrow$ UPDATESUBTREESTRATEGIES($S, \mathbf{v}_1^t, \mathbf{v}_2^t, R^{t-1}$)
    $\mathbf{r}_2^t, R_G^t \leftarrow$ RANGEGADGET($\mathbf{v}_2, \mathbf{v}_2^t(S), R_G^{t-1}$)
  end for
  $\overline{\sigma}^T \leftarrow \frac{1}{T}\sum_{t=1}^{T}\sigma^t$ ▷ Average the strategies
  $a \sim \overline{\sigma}^T(\cdot|I)$ ▷ Sample an action
  $\mathbf{r}_1(a) \leftarrow \langle \mathbf{r}_1, \sigma(a|\cdot)\rangle$ ▷ Update the range based on the chosen action
  $\mathbf{r}_1(a) \leftarrow \mathbf{r}_1(a)/||\mathbf{r}_1(a)||_1$ ▷ Normalize the range
  $\mathbf{v}_2(a) \leftarrow \frac{1}{T}\sum_{t=1}^{T}\mathbf{v}_2^t(a)$ ▷ Average of counterfactual values after action a
  return $a, S(a), \mathbf{r}_1(a), \mathbf{v}_2(a)$
end function

Figure 17: The Re-Solve algorithm

(c) Specifically look at the search process, search is based on depth-first search, and limit the depth, if the restriction depth of 4, the complexity can be reduced to 10^{17}, DeepStack also further application of action abstraction, that is, only part of the action, the complexity is further reduced to 10^7

(d) After evaluating counterfactual values the current strategy needs to be calculated, using the algorithm CFR. which defines a regret value that means that in the current state, I choose behavior

A over behavior B of the Regret level. For example, in rock, paper, scissors, my opponent played scissors and I played cloth, at which point I would regret not playing rock even more, as opposed to Not so much regretting not having scissors out.

A regret is defined as the state reached by the action you regret not choosing. counterfactual value and counterfactual of the current state. value of the difference. For example, in the above example, in the case where the other party produces scissors, I produce the utility of the cloth (counterfactual value) is -1; the utility of scissors is 0; the utility of rock utility is 1. So I choose cloth over scissors for the regret value R(scissors -¿ cloth) = 1; and not stone. The regret value R(stone -¿ cloth) = 2.

After defining the regret value, we can apply the method of Regret Matching to the decision, with the idea that The tendency is to choose actions with a greater regret value. For example, in rock-paper-scissors, if we play n games and then add the regret value of all actions in each game to the up, and then do a normalization, we get a probability distribution as our strategy.$R(a) = \sum_{i=1}^{n} R_i\left(a_i\right)$

The value of regret at each step is determined by the opponent's behavior, so the CFR algorithm is actually changing our own strategy based on the opponent's strategy.

function UPDATESUBTREESTRATEGIES($S, \mathbf{v}_1, \mathbf{v}_2, R^{t-1}$)
  for $S' \in \{S\} \cup$ SubtreeDescendants(S) with Depth(S') < MAX-DEPTH **do**
    for action $a \in S'$ **do**
      $R^t(a|\cdot) \leftarrow R^{t-1}(a|\cdot) + \mathbf{v}_{\text{Player}(S')}(S'(a)) - \mathbf{v}_{\text{Player}(S')}(S')$ ▷ Update acting player's regrets
    end for
    for information set $I \in S'$ **do**
      $\sigma^t(\cdot|I) \leftarrow \frac{R^t(\cdot|I)^+}{\sum_a R^t(a|I)^+}$ ▷ Update strategy with regret matching
    end for
  end for
  return σ^t, R^t
end function

function RANGEGADGET($\mathbf{v}_2, \mathbf{v}_2^t, R_G^{t-1}$) ▷ Let opponent choose to play in the subtree or receive the input value with each hand (see Burch et al. (17))
  $\sigma_G(\mathrm{F}|\cdot) \leftarrow \frac{R_G^{t-1}(\mathrm{F}|\cdot)^+}{R_G^{t-1}(\mathrm{F}|\cdot)^+ + R_G^{t-1}(\mathrm{T}|\cdot)^+}$ ▷ F is Follow action, T is Terminate
  $\mathbf{r}_2^t \leftarrow \sigma_G(\mathrm{F}|\cdot)$
  $\mathbf{v}_G^t \leftarrow \sigma_G(\mathrm{F}|\cdot)\mathbf{v}_2^{t-1} + (1 - \sigma_G(\mathrm{F}|\cdot))\mathbf{v}_2$ ▷ Expected value of gadget strategy
  $R_G^t(\mathrm{T}|\cdot) \leftarrow R_G^{t-1}(\mathrm{T}|\cdot) + \mathbf{v}_2 - v_G^{t-1}$ ▷ Update regrets
  $R_G^t(\mathrm{F}|\cdot) \leftarrow R_G^{t-1}(\mathrm{F}|\cdot) + \mathbf{v}_2^t - v_G^t$
  return $\mathbf{r}_2^t, R_G^t$
end function

Figure 18: The UPdate algorithm

(e) experiment actuall does not point out the real problem. This made this algorithm not able to beat all human. This is because DeepStack also didn't improve its match data with the previous AI, so I'm afraid it's not as strong as it says it is, I It is believed that training with randomly generated data may be its bottleneck. The figure below is given by the authors of its competitor Libratus, which is highly referable.

(f) Improvements: Libratus in **?** consists of three parts: the blueprint strategy, the subgame solution and the self-enhancement module. The blueprint strategy is a macro strategy calculated before the match, and due to the high complexity of the derby, some abstractions are made during the calculation; the subgame solution is a subgame in the Instantly solving sub-games based on blueprint strategies and known information as the game progresses to get a better strategy; self-enhancement module is Refers to adding branches to the blueprint that are not in the blueprint encountered in the actual battle.

Problem 3 continued on next page...

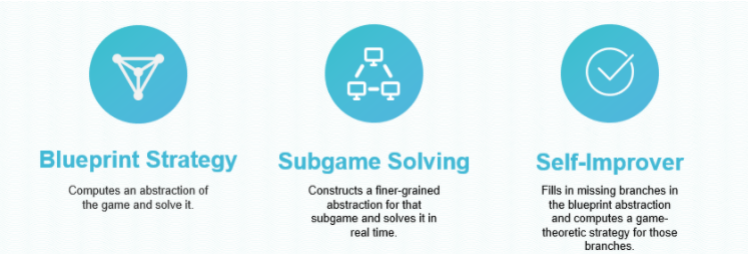


Figure 19: The Libratus structure

In **?**, the author did some convincing experiments to prove their lengths and shorts:

i. +: The theory is complete, and the article demonstrates that the method can converge to a Nash equilibrium; a superior strength, the only Texas Hold'em AI capable of beating top human players.

ii. -: The amount of computation is so large that Libratus runs on a supercomputing center, requiring 200-600 nodes with 14 cores each. 128 gigabytes of memory, 12 million core hours of computation...

Problem 4

For the forbidden rules as discussed in scblqj (2011), I realize them both in the *mcts_pure.py* and *mcts_alphaZero.py* by modifying

```
if(board.current_player==1):
    sensible_moves = board.availables
else:
    now_state=board.current_state()
    sensible_moves = board.availables
    #case1:line,upper-left case
    if (now_state[0][1,6]==1 and now_state[0][2,5]==1 and\
    now_state[0][4,3]==1 and\
    now_state[0][5,2]==1 and\
    now_state[0][6,1]==1
    ):
        sensible_moves.remove(3*8+4)
    #case2:3-3 forbidden,middle-right case
    tmp=now_state[0]
    tmp=np.flatnonzero(tmp)
    for all_loc in sensible_moves:
        if(((all_loc//8)>=2) and ((all_loc%8)>=2)):
            if(((all_loc-1) in tmp) and ((all_loc-2) in tmp)
                    and ((all_loc-8) in tmp)
                    and ((all_loc-16) in tmp)):
                sensible_moves.remove(all_loc)
    #case3:down-right case
    for all_loc in sensible_moves:
        if (((all_loc//8)>=0) and ((all_loc%8)>=1)):
            if(((all_loc-8) in tmp) and ((all_loc+1) in tmp)
```

Problem 4 continued on next page...

```
                    and ((all_loc+2) in tmp)
                    and ((all_loc+3) in tmp)
                    and ((all_loc+8) in tmp)
                    and ((all_loc+16) in tmp)):
                sensible_moves.remove(all_loc)
```

Basic settings, because this is a self-play game, we can't use a human knowledge to help. But I found a good benchmark AI fo
Train environment:

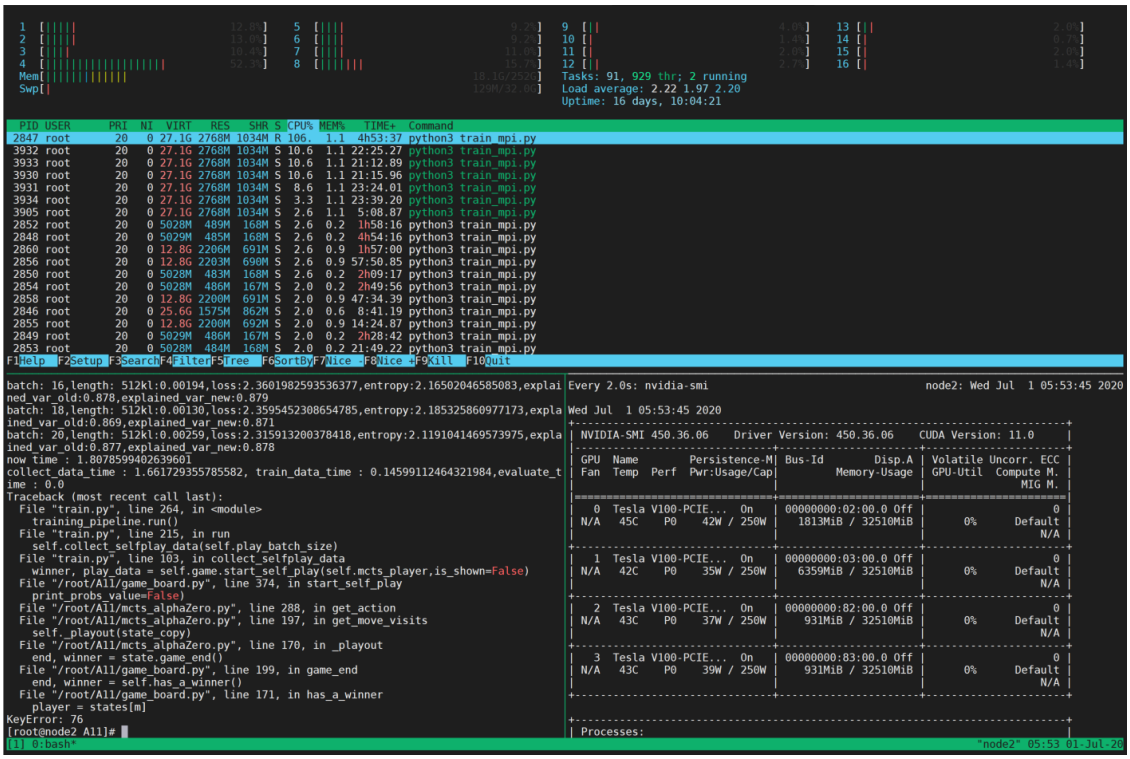


Figure 20: The ocupancy of V100

1. AlphaZero for Gomokuwith for-bidden rulesin a 8 x 8 board

   (a) **baseline** The first modified version of the 8x8 graph is using pytorch and simply add the composed distributed option of pytorch and trained on 4 Tesla V100 for one night. Roughly, The raw output result is: $batchi : 2330, episode\_len : 9, kl : 0.01711, lr\_multiplier : 0.444, loss : 2.674341917037964, entropy : 2.241215705871582, explained\_var\_old : 0.461, explained\_var\_new : 0.565$. The data does not change dramatically afterwards. We set the model as baseline and improves it.

   (b) **improvement** alphazerompi provided a good idea of improving the MCTS practically. The original ideas are all from two paper discussed before. This is based on tensorlayer and tensorflow network. The converge time speeds up to roughly 4 hours to beat the 8-depth minmax tree. The structure is similar to APV-MCTS.

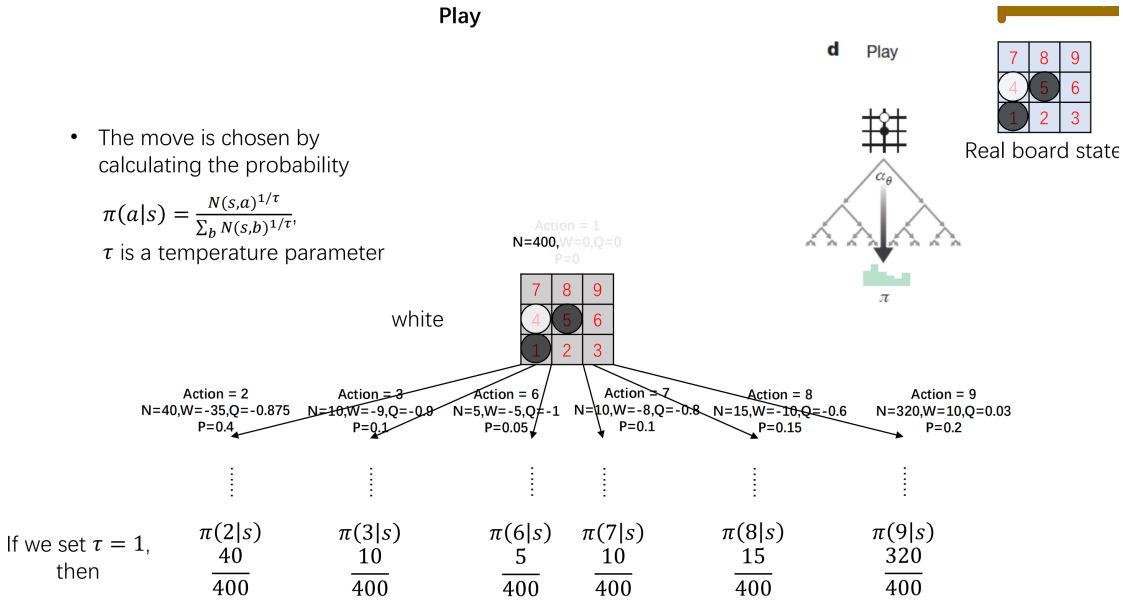


Figure 21: The diagram of the APV-MCTS

i. Network Structure
Current model uses 19 residual blocks, more blocks means more accurate prediction but also slower speed
The number of filters in convolutional layer shows in the follow picture

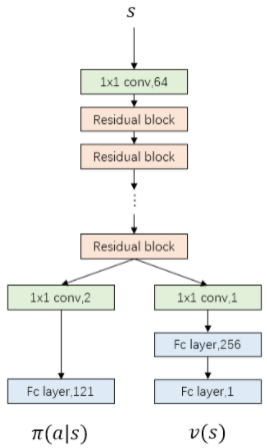


Figure 22: The Network

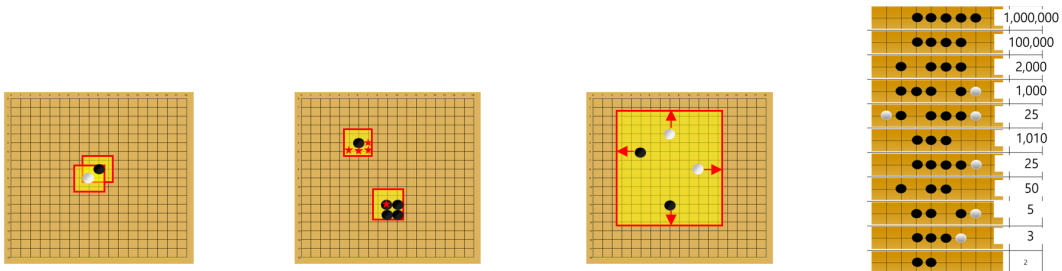
ii. Dirichlet Noise
I add dirichlet noises in each node, it's different from paper that only add noises in root node. I guess AlphaGo Zero discard the whole tree after each move and rebuild a new tree, while here I keep the nodes under the chosen action, it's a little different.
Weights between prior probabilities and noises are not changed here (0.75/0.25), though I think maybe 0.8/0.2 or even 0.9/0.1 is better because noises are added in every node.

iii. MPI
I modified the initial-h's 3 all-to-all to 12 ring-Bcast implementation, which adds up the speed.

Problem 4 continued on next page...

2. Implementation without neural network on $\geq 8x8$.

   Because the problem is not well-explained whether we should use MCTS or just use a technique without MCTS. So I discussed the pure MCTS and minmax compared with the corresponding 8x8, 11x11 and 15x15 MCTS with neural network. For lack of time, the 11x11 and 15x15 MCTS is trained within 4 hours.



Checkerboard space & reward to apply the evaluation function

   (a) minmax algorithm with Alpha-Beta pruning technique:

   To make the Concave AI agent find the Optimal Decision of the Concave game, put the Go virtually in advance and try to find the optimal path

   We implemented the minmax algorithm to find the optimal path by placing the number of opponents in advance.

   In order to prevent the number of searches from increasing exponentially, the search was cut off by setting the number as many as the specified number (depth) in advance, and then the evaluation function was applied to evaluate the route.

   By applying the Alpha-Beta pruning technique, sections that do not need to be searched are removed.

```
def minmax(self, depth, player): # Here to change depth.
    if depth == 0 or self.stoneCnt == GO_BOARD_X_COUNT
                  * GO_BOARD_Y_COUNT:
        return (self.evaluate(), None, None)
    if player == AI:
        maxValue = -INF
        x = 0
        y = 0
        for k in self.searchSpace:
            i, j = k
            if self.goBoard[i][j] == EMPTY:
                self.goBoard[i][j] = AI
    self.stoneCnt += 1
                ret = self.minmax(depth - 1, PLAYER1)
                if ret[0] > maxValue:
                    maxValue = ret[0]
                    x = i
                    y = j
    self.stoneCnt -= 1
                self.goBoard[i][j] = EMPTY
```

```
21            if depth == 1:
22                return (maxValue, x, y)
23            else:
24                return (maxValue, None, None)
25            pass
```

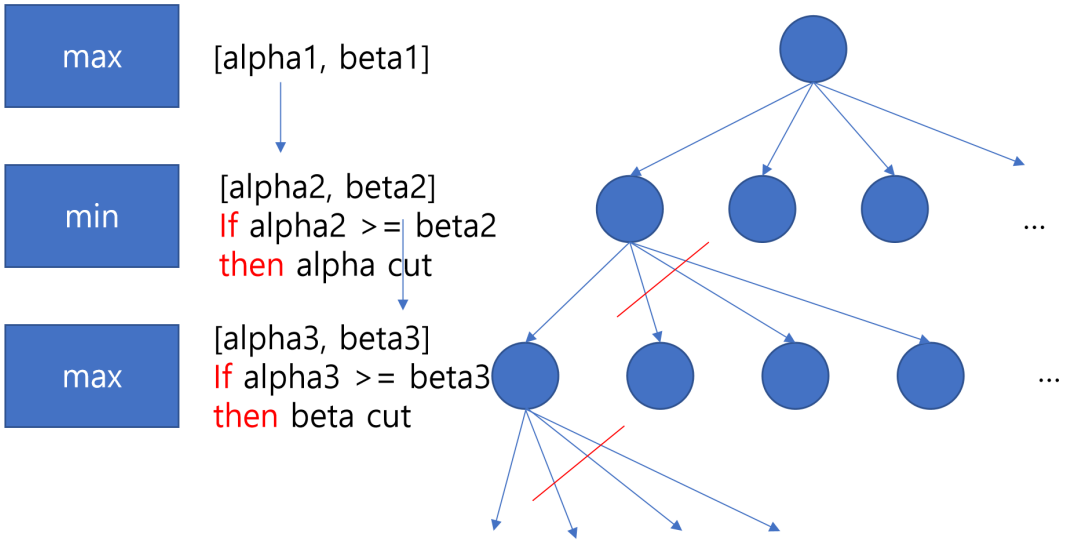


Figure 23: Graph of $minmaxWithAlphaBeta(self, alpha, beta, depth, player)$

The result is that If we keep add up to the search tree, the better the AI. Roughly 8-layer minmax has the ability to match (victorious and defeated by each other in 5 games.) in 11x11 & 15x15. They roughly have the same time of the runtime calculation on my laptop with 2070 maxq and 9750h, but minmax does not make gpu computation. In 8x8 cases, the nueral network performs better. This may because the time that 8x8 takes for Alphazero is ok for it to become a monster, but 11x11 and 15x15, 4 hours don't make it happen.

(b) Pure MCTS

The MCTS algorithm without neural network is just utilize UCB1 algorithm and make iterations. The result of the Pure-MCTS is very poor. In all 8x8 11x11 and 15x15, Pure_MCTS is defeated by the AlphaZero.

3. some summary

(a) experience

I was dealing with the bugs between different input of the tensorflow, it some times crashes in my MPI program. This is because the dimm transported between node is not asychronologically, so may give invalid input to next layer or next step.

(b) insight

Depth Limited minmax is quite good to be a NPC of the game for its lightly used property(you don't have to train and can used to solve small problem), while the RL or DRL based algorithm is likely to be the challenger to the intelligences if given enough time and hardware. Alpha Zero is to converge to the optimal with given times set and can also get a good result if just given small amount of time. MCTS is more scalable but takes longer to converges.

Problem 4 continued on next page...

The overall reason is because the search depth of the algorithm, all the tree algorithm is about to be limited by the computing power. The truth is for small problem minmax and MCTS is more efficient, while MCTS with neural network is more for bigger and harder problem.

(c) lessons

The minmax without AlphaBeta pruning is very slow, as slow as "LianDan", and always lose the game, so pruning is very necessary.

4. Innovative part:

(a) User Interface: I utilize the pygame to draw the playing UI. The UI is maintain in the class GUI.

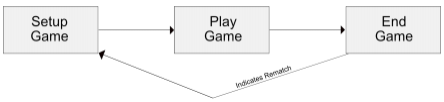


Figure 24: The diagram of GUI

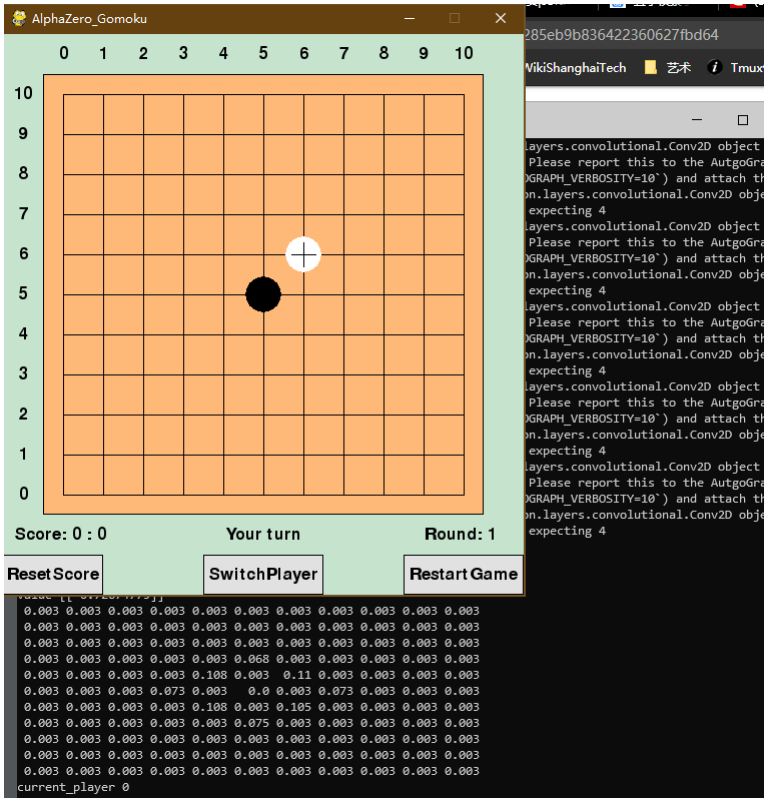


Figure 25: GUI graph

(b) MPI implementation refering to Chaslot et al. (2008). make training time shorter from roughly 8 hours to 4 to the same effects.

(c) Test both pytorch and tensorflow layer(legacy from original AlphaZero_Gomoku).

References

Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving np-hard problems on graphs with extended alphago zero. *arXiv preprint arXiv:1905.11623*, 2019.

Noam Brown and Tuomas Sandholm. Safe and nested subgame solving for imperfect-information games. In *Advances in neural information processing systems*, pages 689–699, 2017.

Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

Iain Dunning, Swati Gupta, and John Silberholz. What works best when? a systematic evaluation of heuristics for max-cut and qubo. *INFORMS Journal on Computing*, 30(3):608–624, 2018.

Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995. ISSN 0004-5411. doi: 10.1145/227683.227684. URL `https://doi.org/10.1145/227683.227684`.

Tomas Mikolov, Ilya Sutskever, Kai Chen, G Corrado, and Jeff Dean. Advances in neural information processing systems 26, nips, 2013. *arXiv preprint arXiv:1310.4546*, pages 3111–3119, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

Ethan Fetayan Weizman Institute of Science. Stochastic optimization markov chain monte carlo, 2012. URL `http://www.wisdom.weizmann.ac.il/ ethanf/MCMC/stochasticoptimization.pdf`.

Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

scblqj. The complete explanation of gomoku forbidden rules, 2011. URL `https://wenku.baidu.com/view/bd330beb856a561252d36f12.html`.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Leo Zhou, Sheng-Tao Wang, Soonwon Choi, Hannes Pichler, and Mikhail D Lukin. Quantum approximate optimization algorithm: performance, mechanism, and implementation on near-term devices. *arXiv preprint arXiv:1812.01041*, 2018.