Reinforcement Learning

Chunheng Jiang

June 20, 2020

All important signals, including the current and visited states, the depth, the size of alternatives never been a winner, the number of winners, the percentage of winners and non-winners in the state, the predicted probability based on other features, are included in a new model that learned from the experience (episodes from root to leaf node). By learning such kind of model, we can get rid of the cache checking and pruning component, we choose the path according to the prediction results from this enhanced function to reduce the number of nodes to extend.

The visualization of the visited path helps to guide us to design a better heuristic algorithm. Based upon some observation, we can improve the definition of our current priority function.

We model the multi-round STV elimination approach as a reinforcement learning problem, where the states are the node (or subset of the alternatives), the action set comes from the eliminations (legal one should follows the rule that one alternative that receives the lowest plurality score will be eliminated), and the reward should be modeled based on the minimum number of nodes to extend along the current branch, and possible early discovery of as many winners as possible, and running time and so on. If a path contains a lot of successful cache checking, it should be punished; if the path contains multiple new winners, it should be assigned more rewards. Even the path is critical, we say that it contains a single winner, and the winner is hard to search, because from all other paths we get different winners from this winner.

When the winner set is empty, there is no need to compute the priorities for states. With a nonempty winner set, we could check whether the predicted winner(s) has/have been elected, if not, its priority will be higher than those have been elected. Once the start state has been fixed, then searching the path with its children states the high priority until a winner is found, or the predicted winner(s) is/are different from the prediction of its parent state.

Visited state should also been fully exploited, such that those nodes with such same states, we should record their corresponding winner(s) when it's available. In order to apply Reinforcement Learning (RL) or Inverse Reinforcement Learning (IRL) techniques to STV, the meanings of states, actions and rewards should be clarified.

1 State Space

A state should include the preference profile, a set of candidates, the visited set of candidates, the known winners. Another way to view the problem: all nodes in the fringe should be considered as the components of the state. It looks like in Go where there are valid actions (further extension) from many positions (nodes).

2 Action Space

There are three ways to define the action space: (1) selection of the remaining candidates: the remaining candidates could indicate a state, as well as a meta-action; (2) selection of one to-be-eliminated candidate: each candidate with the smallest plurality score will be eliminated from the current remaining candidates; (3) adding back or remove one candidate from the current remaining candidates: adding one candidate to the current remaining candidates works like an agent steps backward to the previous node (may be different state), remove one candidate from the current remaining candidates simulates the elimination procedure in STV election. These three action space is quite different, and dimensions are different as well. There are $\sum_{k=1}^{m} A_m^k = 2^m - 1$ possible actions in case (1), m possible actions in case (2), and $2m$ possible actions in case (3). For different states, the valid or legal actions will be varied, e.g. there is no legal action for a leaf node in case (2), because a leaf node may house a cowinner and the current searching path will be terminated; the legal actions for a state in case (3) will be determined by the previously eliminated candidates and to-be-eliminated candidates.

The valid actions for a given state should be the candidates that can be eliminated from the current set of candidates. One action will result in the elimination of a valid candidate from the set of candidates. However, there is a problem, when we reach at a leaf node, it does not implies a complete termination. We have to restart from some point we have visited, and choose another path like allowing regret. Therefore, we set the action space as: moving one step forward (eliminate one legal candidate) or moving $0 < k < m$ steps backward (track backward along the trace). Therefore, the maximum number of legal actions is $2m$: adding back m possible candidates, eliminating m possible candidates. Here, we have to construct a graph, such that each node may have multiple children and multiple parents.

To select a legal action based on the softmax prediction over all individual actions, how to define the probability distribution over a set of individual actions. The multi-label classification algorithms may provide some hints.

The speedup early termination may cause a larger last step, because there are few episodes for such kinds of episodes switching between the state without the last winner and the one with such winner.

3 Reward Function

The reward should consider whether the current set of candidates has been visited, the size of the current set of candidates. Assuming the number of candidates is m, and the number of true co-winners is $1 \leq k \leq m$. Let $\{m_1, m_2, \cdots, m_k\}$ be the numbers of nodes have explore to find all co-winners, and m_i is the number of nodes explored for the ith winner. There is a baseline for us to measure the performance of the searching method. The baseline method can find a set of branches, each branch is of length at most m from up to bottom and has one different co-winner. Therefore, the baseline method can find the ith co-winner after visiting $m * i$ nodes, where $1 \leq i \leq k$. Finding a new winner, i.e. the ith winner, the agent will be awarded according to the rule

$$r(m_i) \propto m * i / m_i,$$

and the reward will increase as m_i reduces. The reward function is independent from the effective number of candidates, and may work well. Early discovery complies with the above rule, and a reward for a complete episode is defined as:

$$r(s_1, a_1, s_2, a_2, \ldots, a_T, s_T) = \bar{r} \sum_{i=1}^{k} \frac{1}{i} \frac{m * i}{m_i} = \bar{r} \sum_{i=1}^{k} \frac{m}{m_i} = m\bar{r} \sum_{i=1}^{k} \frac{1}{m_i},$$

where $\bar{r} > 0$ is a base reward for $m_i = m * i$.

The range of the rewarding function could be another factor that affects the performance of the agent. If the range is too small, it fails to guide the agent to learn the difference between positive and negative actions. If the range is too large, the feedback may disturb the performance.

Besides, the searching path for all co-winners is various for different profiles. To make a general rewarding policy, we are considering using the ground-truth winners for each node and define the reward function after evaluation several relationships. Suppose a state s_t, an action a_t in s_t are given, then the next state s_{t+1} will be deterministic. Assuming there are N co-winners, the existing winners are C, the true winners for s_t and s_{t+1} are C_t, C_{t+1} respectively (if we are in s_t and it has one remaining candidate, the candidate has been collected into C). The action a_t is awarded based on: freshness (whether there is fresh winners in C_t and C_{t+1}) and cost (what's the depths of s_t and s_{t+1}).

There are two types of actions: positive actions and negative ones. A positive action increases the freshness, e.g. $|C_t \cap C| < |C_{t+1} \cap C|$. A negative action will decrease the freshness, e.g. $|C_t \cap C| > |C_{t+1} \cap C|$. The level of freshness is closely related to the corresponding cost. There are several types of cost in searching, but we concern most about is the required nodes to visited before finding a new winner. To reduce the cost, we prefer an action that increase the freshness with as few cost as possible.

- $|C_t \cap C| > 0$: a positive action is that of $|C_{t+1} \cap C| > 0$ and $d_t < d_{t+1}$; a negative action is that of $|C_{t+1} \cap C| = 0$. All other cases are negative.
- $|C_t \cap C| = 0$: all actions that result in new winners are positive, i.e. $|C_{t+1} \cap C| > 0$, and the ones in deeper layers are preferred to those in shallower layers. All other cases are negative.

Based on the above analysis, we have the following definition:

1. Case 1: $|C_t \cap C| = 0$

$$r(s_t, a_t, s_{t+1}) = \begin{cases} (d_{t+1}+1)(1+\frac{|C_{t+1}\cap C|}{N}), & |C_{t+1}\cap C| > 0 \\ -5, & |C_{t+1}\cap C| = 0 \end{cases} \tag{1}$$

2. Case 2: $|C_t \cap C| > 0$

$$r(s_t, a_t, s_{t+1}) = \begin{cases} (d_{t+1}+1)(1+\frac{|C_{t+1}\cap C|}{N}), & |C_{t+1}\cap C| > 0 \text{ and } d_{t+1} > d_t, \\ -5, & |C_{t+1}\cap C| > 0 \text{ and } d_{t+1} \le d_t \\ d_t - m, & |C_{t+1}\cap C| = 0 \end{cases} \tag{2}$$

To use a single set of hyper-parameters to solve different tasks, the rewards and the temporal difference errors are clipped to $[1, +1]$, because the magnitudes and frequencies of rewards vary wildly between different games. The approach can be called reward clipping[1]. However, the clipping changes the problem that the learning agent is trying to learn and even affects the learned behaviors. **(author?)**[2] proposed a adaptive target normalization to control the magnitude of each update. Especially in neural network back-propagation, a too large update may harm learning.

Age-sensitive reward: each action has a life-span, when an action is performed, its life ends. To reduce the steps to find all co-winners, we expect that a good action should be performed as soon as possible. Therefore, the age of an action should also be taken into consideration when define the rewarding function.

4 Meta Action

Given m individual actions that an agent can take, the agent can take multiple actions a simultaneously, like a robot turns left ($a_0 = 0$, raises its right hand ($a_1 = 1$), and smiles ($a_2 = 1$). Let the output of the actor being $x \in \mathbb{R}^m$ without any activation. To make each element corresponding to one single action being a probability in $[0,1]$, we use the sigmoid activation function $\sigma(x_i) = 1/(1+e^{-x_i})$ to the output. How to produce the probability distribution over a meta-action which has at least one active individual actions? Here, we assume that all individual actions are independent from each other, then the probability distribution for a meta action could be the joint probability of each individual action in the given meta-action. Suppose the actor network outputs the probability over all individual action $p_i = \sigma(x_i)$, then the probability of a meta action $a \in \mathbb{B}^m$ could be defined as

$$Pr(a|x) = \prod_{i=1}^{m} [p_i^{a_i}(1-p_i)^{1-a_i}].$$

The one with the highest probability will be selected to take. Equivalently, we choose the one with maximum log-likelihood

$$\log Pr(a|x) = \sum_{i=1}^{m} [a_i \log p_i + (1-a_i)\log(1-p_i)] = \sum_{i=1}^{m} [a_i x_i - \log(a + e^{x_i})].$$

4.1 Continuous Action

Each meta action in STV actually has a unique plurality score, which could be directly used to describe the action. If the actor can fit the plurality score perfectly, is it a better policy network?

4.2 Board Action

We create a board-like action space, with the candidates and its positional counts in the profile being the two variables on $x-y$ axis. Could we take advantage of method used in AlphaGo?

5 Q-Learning Method

A Markov Decision Process (MDP)[3] relies on the Markov property: the probability of the next state s_{t+1} only depends on current state s_t and the action a_t, but not on the preceding states or actions. Given an MDP episode, i.e. a sequence of states, actions and rewards: $\{s_0, a_0, r_1; s_1, a_1, r_2; \ldots; s_t, a_t, r_{t+1}; \ldots; s_{n-1}, a_{n-1}, r_n; s_n\}$, where $s_t \in \mathcal{S}$ is the state of the agent at step t, $a_t \in \mathcal{A}$ is an action the agent takes at step t, and r_{t+1} is the immediate reward that agent receives from the environment after it takes action a_t at state s_t. The state s_n is a terminal state, which indicates the termination of the agent. Given an MDP, we can calculate the total reward R for an episode $R = r_1 + r_2 + \ldots + r_n$ and the total future reward R_t from step t onward $R_t = r_t + r_{t+1} + \ldots + r_n$. To make sure the agent can receive the immediate or near rewards, a common approach is to place more weights on the near rewards. Therefore, we have the discounted future reward

$$R_t = r_t + \gamma_{t+1}^{r} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-t} r_n,$$

where $\gamma \in [0,1]$ is a discount factor, the more into the future the reward is, the less we take it into consideration. If $\gamma = 0$, the agent is short-sighted, only relies on the immediate reward. If $\gamma = 1$, there is no difference between the immediate reward and all future rewards. To keep a balance between the immediate reward and future rewards, it's commonly set $0 < \gamma < 1$. After a rearrangement, we have

$$R_t = r_t + \gamma[r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-t-1} r_n] = r_t + \gamma R_{t+1}.$$

The discounted rewarding method reveals a greedy strategy for the agent – taking an action with a maximum discounted reward. To explicitly describe the dependence of R_t on s_t and a_t, we define the Q-value and have the well-known Bellman equation:

$$Q(s_t, a_t) = r_t + \gamma \max R_{t+1} = r(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1} \in \mathcal{A}} Q(s_{t+1}, a_{t+1}).$$

Suppose there exists a Q-function $Q(s,a)$ satisfying the Bellman equation. Given an environment and an agent, Q-Learning tries to approximate the Q-function with samples. We define the simple squared error loss function [1] [2]

$$L(s,a) = [(r(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q(a', s')) - Q(s,a)]^2,$$

and minimize it with stochastic gradient descent method and therefore iteratively approximate the Q-function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a)],$$

where $0 \leq \alpha \leq 1$ is a learning rate, and $r(s,a,s') + \gamma \max_{a'} Q(s', a') - Q(s,a)$ is the temporal difference (TD) error.

For a small and simple case, a Q-table like an memoizatioin table in dynamic program- ming (DP), could be enough to search an optimal path for the agent. When it comes to a complex situation (continuous action space, large-scale state space), it becomes in- tractable using the tabular-like method. **(author?)** [1],[4] in DeepMind proposed an alternative approach Deep Q-network (DQN). It implemented the Q-learning by approximating the Q-function with a Convolutional Neural Network (CNN or ConvNet). It is a form of Q-learning with function approximation (using a neural network), which means it tries to learn a state-action value function Q (given by a neural network in DQN) by minimizing temporal-difference errors, i.e. trying to make the value $Q(s,a)$ close to $r + \gamma \max_{a'} Q(s', a')$ after observing a transition (s,a,r,s'), where

[1] https://www.nervanasys.com/demystifying-deep-reinforcement-learning/
[2] http://users.isr.ist.utl.pt/ mtjspaan/readingGroup/ProofQlearning.pdf

the actions can be chosen arbitrarily (the algorithm is off-policy), typically using the ϵ-greedy approach based on the current Q function.

Model-free algorithms directly optimize the optimal policy or value function through algorithms such as policy iteration or value iteration. It is much computationally efficient, but requires vast amount of training data.

6 Policy Gradient Method

Assuming policy $\pi_\theta(s,a) = \mathbb{P}(a|s;\theta)$ is differential whenever it's non-zero, and the policy gradient can be computed using a surrogate gradient

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a),$$

the gradient of the likelihood function $\log \pi_\theta(s,a)$, which is also known score function, derived from the maximum likelihood estimation (MLE).

- A softmax policy weights actions using a linear combination of features $\phi(s,a)^T\theta$,and the probability of an action is
$$\pi_\theta(s,a) = \frac{\exp\{\phi(s,a)^T\theta\}}{\sum_a \exp\{\phi(s,a)^T\theta\}},$$
and the likelihood function $\log \pi_\theta(s,a) = \phi(s,a)^T\theta - \log\sum_a \exp\{\phi(s,a)^T\theta\}$. The score function will be
$$\begin{aligned}\nabla_\theta \log \pi_\theta(s,a) &= \phi(s,a) - \textstyle\sum_a[\frac{\exp\{\phi(s,a)^T\theta\}}{\sum_a \exp\{\phi(s,a)^T\theta\}}\phi(s,a)]\\ &= \phi(s,a) - \textstyle\sum_a[\pi_\theta(s,a)\phi(s,a)] = \phi(s,a) - \mathbb{E}_{\pi_\theta(s,\cdot)}.\end{aligned}$$
Intuitively, an action will move along the direction of the difference between the action feature and the average action feature, and the scale is determined by its probability.

- A Gaussian policy in continuous actions state assumes that the action a satisfies a Normal distribution. More specifically, $a \sim N(\mu(s), \sigma^2)$, where mean $\mu(s)$ is defined as a linear combination of the state features, i.e. $\mu(s) = \phi(s)^T\theta$, variance σ^2 is fixed and can also be parametrized. The likelihood function
$$\log \pi_\theta(s,a) \propto \log \exp\{-\frac{(a-\mu(s))^2}{2\sigma^2}\}$$
and the score function is $\nabla_\theta \log \pi_\theta(s,a) = (a-\mu(s))\phi(s)/\sigma^2$. Could we make a multivariate Gaussian distribution to present the meta-action distribution?

Theorem 1. *For any differential policy $\pi_\theta(s,a)$ and a policy objective function*

$$J(\theta) = \mathbb{E}_{\pi_\theta} Q^\pi(s,a) = \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a) Q^{\pi_\theta}(s,a),$$

its policy gradient is $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \times Q^{\pi_\theta}(s,a)]$.

Monte-Carlo policy gradient has high variance. To reduce the variance, the most commonly used algorithms are the actor-critic algorithms. The algorithms have a critic to estimate the state-action function $Q(s,a;w) \approx Q^{\pi_\theta}(s,a)$ and update parameters w, and have an actor to estimate the policy function $\pi_\theta(s,a)$ and update the parameters θ in direction suggested by the critic. An actor-critic algorithm follows an approximated policy gradient

$$\begin{aligned} \nabla_\theta J(\theta) &\approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q(s,a;w)], \\ \delta\theta &= \alpha \nabla_\theta \log \pi_\theta(s,a) Q(s,a;w), \end{aligned}$$

while the critic network is updated as in Q learning (temporal difference). To improve the stability of the training, both Experience Replay and Target Network can be employed. Here the actor is a stochastic policy. Another policy is deterministic policy network, where $a = \mu(s;\theta)$. Given a state, the action is deterministic. The policy objective function is defined as

$$J(\theta) = \mathbb{E}_{r\sim P(r|\theta)} Q(s,a) = \mathbb{E}_{r\sim P(r|\theta)} Q(s,\mu(s;\theta)),$$

and the policy gradient is $\nabla_\theta J(\theta) = \mathbb{E}_{r\sim P(r|\theta}[\nabla_a Q(s,a) \times \nabla_\theta \mu(s;\theta)]$. We will introduce a deep reinforcement learning algorithm in the following sections, which coins Deep Deterministic Policy Gradients (DDPG)[5,6].

6.1 Policy Network

The PG method directly learns a policy function (instead of a Q-function) and are typically on-policy, meaning you need to learn from trajectories generated by the current policy. One way to learn an approximation policy is by directly maximizing expected reward using gradient methods, hence "policy gradient" (the gradient of the expected reward w.r.t. the parameters of the policy can be obtained via the policy gradient theorem for stochastic policies, or the deterministic policy gradient theorem[5] for deterministic policies).

During training, the network can be used to sample an action according to the predicted probability distribution. When evaluated, the action in a state is deterministic, the one with the highest probability will be selected.

6.2 Stochastic policy gradient

6.3 Deterministic policy gradient

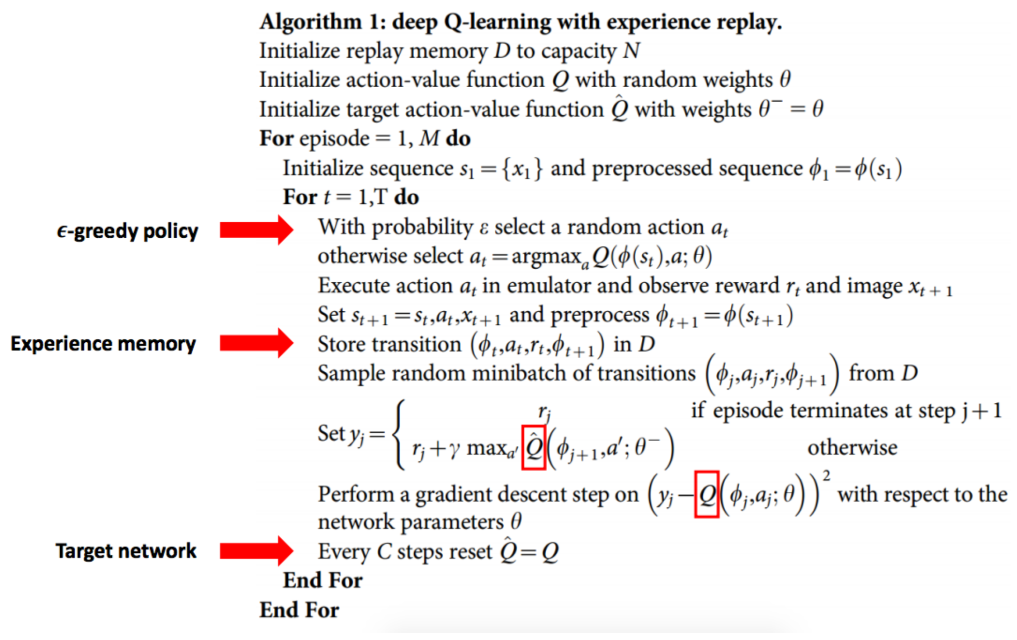
The deterministic policy gradient is the expected gradient of the action-value function[5].

6.4 Experience Replay

Experience replay keeps a specific number of visited episodes $\{s, a, r, s'\}$, and updates the network in batch with samples from the replay memory D. The model is trained to minimize the average temporal difference errors

$$L(s,a;\theta) = \mathbb{E}_{(s,a,r,s')\sim D}[r(s,a,s) + \gamma \max_{a'} Q(a',s') - Q(s,a)]^2,$$

making the predicted action-value $Q(s,a)$ close to the target value $r(s,a,s') + \gamma \max_{a'} Q(a',s')$. Both the experience replay and the target network are two effective approaches to improve the stability of the training.



ϵ-greedy policy
Experience memory
Target network

```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \boxed{\hat{Q}}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - \boxed{Q}(\phi_j, a_j; \theta)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Fig. 1: DQN with experience replay

6.5 Target Network

The non-linear neural network Q-value function is not stable. To resolve the problem, DeepMind team proposed to use a target network: cloning the actor and critic networks, and using them as the target networks to computing the target values. After frozen for a specific number of steps, the target networks are updated in very smooth pattern:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta',$$

where θ' is the weight vector of the target network and θ is the weight vector of the actor or critic network, and the update factor $\tau \ll 1$ ensures the change of the network is subtle and effectively improve the stability of the training. We have derived that

$$\mathbb{E}_{r\sim P(r|\theta)}[\nabla_a Q(s,a) \times \nabla_\theta \mu(s;\theta)],$$

therefore the update of the actor network depends on two components: $\nabla_a Q(s,a)$ and $\nabla_\theta \mu(s;\theta)$. The first term is computed from the critic network and the second one comes from the actor network itself, both require the predicted action from the actor network from the current state.

We have the weights of the actor and the critic network, then we can update the target networks

$$\begin{aligned} \text{critic's target network}: \quad & \theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}, \\ \text{actor's target network}: \quad & \theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}. \end{aligned}$$

7 Actor-Critic Method

7.1 A3C

Asynchronous Advantage Actor-Crtic (A3C, **(author?)** [7]) is different from DQN, it has a global network, and multiple subnetworks, each of them is trained independently by interacting with the environment. A3C performs better and learns more efficiently than DQN.

7.2 UNREAL

UNsupervised REinforcement and Auxiliary Learning (UNREAL, **(author?)** [8]) learned additional auxiliary tasks to improve the performance of the actor.

7.3 Ensemble in Reinforcement Learning

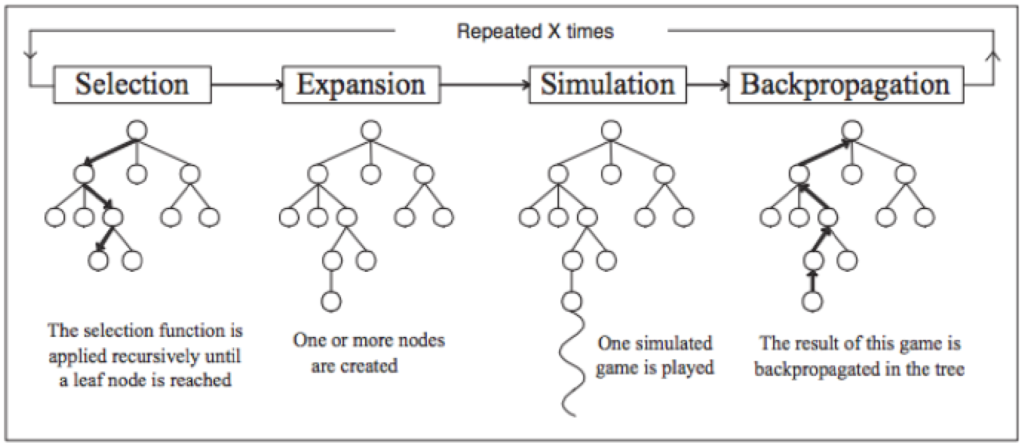
(author?) [9] proposed to strengthen reinforcement learning across environments with the assistances from ensemble learning techniques.

8 AlphaGo

AlphaGo[5] is a combination of Monte Carlo Tree Search (MCTS) and machine learning (supervised and reinforcement learning). MCTS is an alternative approach to brute force and heuristic search. It runs many game simulations starting at the current game state until the game is over. The simulations keeps trace some values, e.g. the number of time a new state is visited, the chance to win. The more simulations are executed, the more accurate a selection leading to a winning. It's proved that, as the number of simulations increases, MCTS leads to an optimal play.

The advantage for MCTS is no requirement in domain knowledges to play the game. But, it suffers from the exploration-exploitation tradeoff. To achieve a good result, it relies on large number of simulations. It seems both MCTS and hand-crafted rules are necessary.

Fig. 2: Architecture of AlphaGo[5]


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

8.1 SL Policy Network

An SL policy network $p_\sigma(a|s)$ tries to predict experts' moves. The input of SL policy network is a simple representation of the board position, and the output is an action (a 19×19 Go board, i.e. a binary vector of 361 dimensions[10]). It's a classical classification problem, and trained using sampled position-move pairs (s,a) from human played games. The surrogate loss function is a cross entropy (CE) function

$$L(s,a;\sigma) = -\sum_a I(s,a) \log p_\sigma(a|s),$$

where $I(s,a) \in \{0,1\}$ is an indicator, shows whether a human player takes action a in state s. Using stochastic gradient descent approach to minimize CE loss is equivalent to maximize the likelihood of an expert move a in state s,

$$\Delta\sigma = \nabla_\sigma \log p_\sigma(a|s).$$

The SL policy network $p_\sigma(a|s)$ has 13 layer, including 12 convolutional layers and one soft- max output layer, which indicates the probability distribution over all legal moves. Also, a fast policy network $p_\pi(a|s)$

is modeled as a linear softmax of a smaller set of board features. It's the rollout policy network for rapidly sampling actions during rollouts. Both policy networks are move-predictor, mimicking human players' moves. But there is a large uncharted territory that a human may fail to explore. To explore fresh strategies and obtain an enhanced policy network, the SL policy network $p_\sigma(a|s)$ is improved further based on the policy gradient reinforcement learning algorithm. The reinforcement learning algorithm is trained using the self-played games between the current RL policy network and a randomly sampled version in previous iteration.

8.2 RL Policy Network

A RL policy network $p_\rho(a|s)$ with the same structure as the SL policy network $p_\rho(a|s)$ is built, and initialized using $p_\rho(a|s)$'s weights, i.e. $\rho = \sigma$. A reward function is defined for RL network as follows

$$r_t = r(s_t) = \begin{cases} 0, & t < T \text{ for both,} \\ +1, & t = T \text{ for winner,} \\ -1, & t = T \text{ for loser.} \end{cases}$$

The outcome $z_t = r(s_T) = \pm 1$ is the terminal reward of the player in current state s_t (note: the reward r_t is difference from the outcome z_t). The weight vector ρ is updated with the stochastic gradient descent method to maximize the expected outcome,

$$\Delta\rho = \frac{\alpha}{n} \sum_{i=1}^{n} \sum_{t=1}^{T} \nabla_\rho \log p_\rho(a_t^i | s_t^i) \times z_t^i,$$

where n is the size of mini-batch, and a game will be played until the end for each batch.

8.3 Value Network

To evaluate a position, we estimate a value function $v^p(s)$ that predicts the reward from position s of the game following policy p,

$$v^p(s) = \mathbb{E}[z_t | s_t = s, a_{t\ldots T} \sim p].$$

If a perfect policy is known $p^*(a|s)$, we would like to know the optimal value function $v^*(s)$ under the perfect policy. In practice, we estimate the value function $v^{p_\rho}(s)$ with the stronger RL policy network $p_\rho(a|s)$. Further, a value network $v_\theta(s)$ with weight vector θ is built to approximate the value function $v^{p_\rho}(s)$, i.e.

$$v_\theta(s) \approx v^{p_\rho}(s) \approx v^*(s).$$

The output of the value network is a single value rather than a probability distribution.

The successive positions in a Go game are strongly correlated, differing by just one stone and sharing a same target reward. To avoid overfitting for using a complete game to train the value network. AlphaGo samples 30 millions distinct positions, each sampled from a separated self-played game. The algorithm generates a game with the following phrases:

- Sampling a time step $u \sim U(1, 450)$, and all actions a_t before u, i.e. $a_t \sim p_\sigma(\cdot|s_t)$ (where the initial state come from?), where $t < u$
- Sampling an available action $a_u \sim U(1, 361)$, repeatedly until a_u is legal
- Sampling a sequence of actions $\{a_{u+1}, \cdots, a_T\}$ until the end of the game, $a_t \sim p_\rho(\cdot|s_t)$, where $u + 1 \leq t \leq T$
- Obtain the game outcome $z_{u+1} = r(s_T)$ at the end of the game
- A single date entry (s_{u+1}, z_{u+1}) is collected from the above self-played game, and it's an unbiased sample of the value function $v^{p_\rho}(s_{u+1}) = \mathbb{E}[z_{u+1}|s_{u+1}, a_{u+1}, \cdots, T \sim p_\rho]$

Suppose there are m state-outcome pairs $\{s_i, z_i\}_{i=1}^m$ in the training set. The stochastic gradient descent method is used to minimize a mean-squared error (MSE) between the predicted value and the target outcome

$$\Delta\theta = \frac{\alpha}{m} \sum_{i=1}^{m} [(z^i - v_\theta(s^i)) \nabla_\theta v_\theta(s^i)].$$

8.4 Searching with Policy and Value Networks

AlphaGo combines the policy networks and the value networks in an MCTS algorithm that selects actions by lookahead search. Before we dive into the AlphaGo MCTS algorithm, we have a brief introduction of the general MCTS algorithm, which may be helpful to understand the following sections.

8.5 MCTS

Before **(author?)** [11] invented MCTS, the most popular tree search algorithm is Minimax alpha-beta pruning. It works very well when the branching factor is small and we have a good heuristic function. If the branching factor is large, it will be intractable to use the Minimax. Especially in most complex game, we do not have a good heuristic to search the game tree. MCTS is created for these issues, and its biggest advantage is no reliance on the prior knowledge of the game. There are four stages in MCTS: selection, expansion, simulation and back propagation, as indicated in Fig. 3.

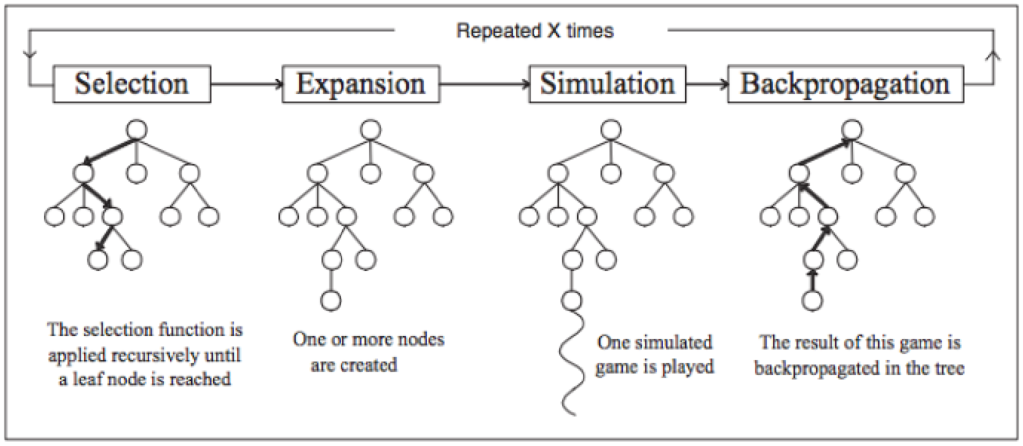

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Fig. 3: Outline of MCTS[11]

- Selection: it is an in-tree search phrase, traverses the game tree from the root node. When a node has children, it requests to select an optimal one based on a tree policy or the stored statistics in each node.
- Expansion: when it reaches a leaf node, the search algorithm may step into the node and expand it. It will branch the node and produce one or more children and select one to visit. (A leaf node contains at least one child that has never been visited; a terminal node is the end of the entire game.)
- Simulation: when a child of a leaf node is selected, it's a new root node, and the search will randomly select legal actions until reaches a terminal node.
- Back-propagation: the game result at the terminal node is back-propagated along the searching path until the root node, and nodes' statistics (e.g. number of visits, win rate) will be updated accordingly.

The whole procedure will be repeated multiple times. As the number of simulations increases, the statistics about the game will become more and more accurate.

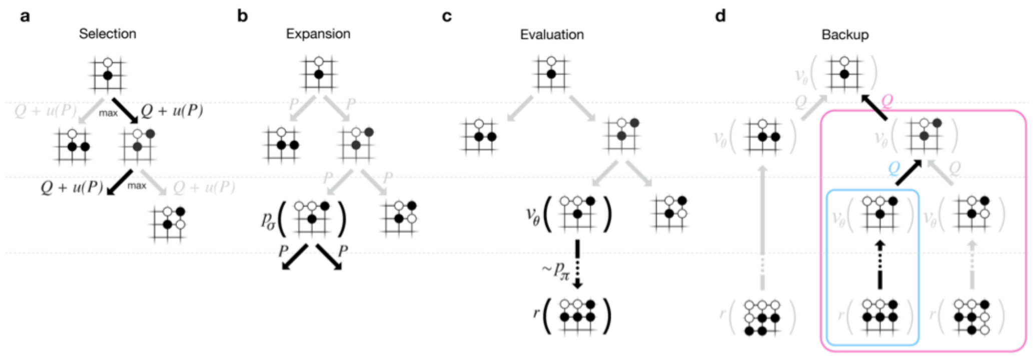
8.5.1 MCTS in AlphaGo

Each node s in the search tree contains edges (s,a) for all legal actions $a \in \mathcal{A}(s)$. Each edge keeps track of the statistics: the action-value $Q(s,a)$, visit count $N(s,a)$, and prior probability $P(s,a)$, which is estimated using the SL policy network, i.e. $P(s,a) = p_\rho(a|s)$. More specifically, the visited count contains two components, one for the leaf evaluation, another one for the rollout rewards. Therefore, it contains $W_v(s,a)$ and $W_r(s,a)$

the Monte-Carlo estimates of the total action-value, accumulated over $N_v(s,a)$ and $N_r(s,a)$ leaf evaluations and rollout rewards, respectively. The action-value $Q(s,a)$ is a weighted average of the two estimates.

The MCTS algorithm in AlphaGo is the same as the general MCTS, contains four stages: selection, expansion, evaluation, and backup, as indicated in Fig. 4.

Fig. 4: MCTS in AlphaGo



8.5.2 Selection

Each simulation starts from the root node of the search tree and finishes when it reaches a leaf node at depth L. At each time step $t < L$, an action is selected based on the statistics

$$\begin{cases} a_t &= \arg\max_a [Q(s_t,a) + u(s_t,a)], \\ u(s,a) &= c_{puct} P(s,a) \frac{\sqrt{\sum_b N_r(s,b)}}{1+N_r(s,a)}, \end{cases}$$

where the constant $c_{puct} > 0$ determines the level of exploration. The search strategy prefers the actions with high prior probability and low visit count. As the simulation goes on, it asymptotically prefers actions with high action-value.

8.5.3 Expansion

When the visit count $N_r(s,a)$ of an edge (s,a) exceeds a threshold, e.g. $N_r(s,a) > n_r$ (may be dynamically changed), the successor node s' is added to the search tree (even an edge (s,a) goes out of the node s has been visited for many times in the rollouts, it's still possible that the edge is not in the search tree). The statistics of the out-edges of the new state are initialized as

$$\begin{cases} N_v(s',a) &= N_r(s',a) = 0, \\ W_v(s',a) &= W_r(s',a) = 0, \\ Q(s',a) &= 0, \\ P(s',a) &= p_\sigma^\beta(a|s'), \end{cases}$$

where β is the softmax temperature of the SL policy network.

8.5.4 Evaluation

The leaf node s_L is evaluated by the value-network $v_\theta(s)$, unless it has been evaluated in previous simulations. The rollout phrase in each simulation starts at s_L until the end of the game. At each time-step $t \geq L$, an action is selected according to the rollout policy network, i.e. $a_t \sim p_\pi(\cdot|s_t)$. At the end of the game, the outcome z_t can be computed.

8.5.5 Backup

At the end of the simulation, the rollout statistics are updated in a backward pass through each in-tree time step $t \leq L$,

$$\begin{aligned} N_r(s_t,a_t) &\leftarrow N_r(s_t,a_t) + 1, \\ W_r(s_t,a_t) &\leftarrow W_r(s_t,a_t) + z_t. \end{aligned}$$

Another backward propagation is performed to update the value statistics, after the evaluation $v_\theta(s_L)$ of the leaf s_L,

$$\begin{aligned} N_v(s_t,a_t) &\leftarrow N_v(s_t,a_t) + 1, \\ W_v(s_t,a_t) &\leftarrow W_v(s_t,a_t) + v_\theta(s_L). \end{aligned}$$

The overall average action-value $Q(s,a)$ is updated as the weighted average of the Monte-Carlo estimates on rollout rewards and leaf evaluation, i.e.

$$Q(s,a) = (1-\lambda)\frac{W_v(s,a)}{N_v(s,a)} + \lambda\frac{W_r(s,a)}{N_r(s,a)},$$

where $\lambda \in [0,1]$ keeps a balance between the leaf evaluations and the rollout rewards in the simulations. It's a kind of exploration-exploitation tradeoff, where the leaf evaluation gives the signal of exploration (simulators repeatedly kicked the boundary of the search tree), and the rollout rewards of exploitation level (in-tree search limited in the charted territory).

To make the explanation clear, we introduce the initial several steps from the perspective of one simulator:

- Evaluation: when one player places the first stone, a root node s_1 is added to the search tree. Suppose one simulator starts to simulate from $s_L = s_1$, initializing the statistics $\{N_v(s_1, a) = N_r(s_1, a) = 0, W_v(s_1, a) = W_r(s_1, a) = 0, Q(s_1, a) = 0\}$ for all legal actions a of s_1. It evaluates the state-value $v\theta(s)$ of the leaf node (other available simulators may have already evaluated it). After the evaluation, it starts to rollout according to the rollout policy network $p_\pi(a|s)$ until the end of the game. The corresponding game results will be computed in the simulation.
- Backup: all statistics of the out-edges of the root node will be collected from available simulators and updated or computed.
- Expansion: suppose one legal action a_1 of s_1 has its rollout visit count larger than the threshold n_r, the new node s_2 will be added into the search tree, and initialized with $\{N_v(s_2, a) = N_r(s_2, a) = 0, W_v(s_2, a) = W_r(s_2, a) = 0, Q(s_2, a) = p_\sigma\beta(a|s)\}$.
- Selection: the simulator starts the simulation at the root node until reaches the leaf node s_L in the search tree. At each time step $t < L$, it selects the action according to $a_t = \arg\max_a[Q(s_t, a) + u(s_t, a)]$.
- Together with other simulators, the above steps are repeatedly executed, the related statistics are updated with the helps of the SL policy network $p_\sigma(a|s)$, the rollout policy network $p_\pi(a|s)$ and the value network $v_\theta(s)$. The player could be guided with the asymptotically accurate statistics to make another move.
- Starting from the new state after the suggested move, all simulations are repeated again until the end of the game.

The strong RL policy network $p_\rho(a|s)$ is not utilized explicitly in AlphaGo's MCTS, the reason lies at the fact that the SL policy network $p_\sigma(a|s)$ learned from expert moves is much more diverse in strategies and very suitable for the tree search in MCTS. Meanwhile, the state-value network $v_\theta(s)$ learned from the self-played games sampled using $p_\rho(a|s)$ has a better generalization and presents its implicit power as well.

Also, when the stochastic policy networks $p_\sigma(a|s)$ and $p_\pi(a|s)$ predict the probability distribution over actions. The illegal actions may harvest the highest probabilities. There- fore, it requires a renormalization to redistribute the probabilities of these illegal moves to the legal ones[12] to produce accurate predictions.

References

1. 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.

2. Hado P van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, pages 4287–4295, 2016.

3. Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 1998. *A Bradford Book*, 1998.

4. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

5. David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

6. Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

7. Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

8. Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

9. Vukosi Marivate and Michael Littman. An ensemble of linearly combined reinforcement-learning agents. In *Proceedings of the 17th AAAI Conference on Late-Breaking Developments in the Field of Artificial Intelligence*, pages 77–79. AAAI Press, 2013.

10. Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014.

11. Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: a new framework for game ai. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 216–217. AAAI Press, 2008.

12. Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*, 2014.