Training a value function in the game of Go

Bo Peng*

January 11, 2017

**** This is not a research paper, but some thoughts on a possibly efficient method to train a value function in the game of Go. The author welcome all critiques and discussions, and you can reach me at "bo at withablink dot com".***

In early 2016, Google Deepmind reshaped the study of computer Go by the celebrated Nature paper, where a MCTS + SL + RL pipeline was proven to be highly successful.

The training of the policy networks p_π and p_σ in the Nature paper are straightforward, while the training of the value network v_θ is more elusive, which requires a new RL policy network p_ρ and a huge number of self-play positions.

This is an intriguing situation, because mathematically the perfect policy network can be quickly and directly derived from the perfect value network (the best next move is the move that maximises the value for the player, if the value function is perfect), but the reverse direction is less direct, where one has to iterate the policy network and apply an end-game value function. In this sense, the value function is more essential.

Here we purpose a method to train a value function V directly, without the appearance of the policy network. And we aim to build a pure-value-network Go player.

1 Defining the value function

Let s be the board state. If s is an end-game position, a natural value function $\tilde{V}(s)$ exists, which is simply area scoring,

$$\tilde{V}(s) = \sum_{i \in B} \tilde{v}(s, i).$$

Here $B = \{1, \cdots, 361\}$ are the board intersections, and $\tilde{v}(s, i)$ is the final ownership function of the board intersection i,

$$\tilde{v}(s, i) = \begin{cases} 1 & i \text{ is owned by the player in the end-game position } s \\ -1 & i \text{ is owned by the opponent in the end-game position } s \\ 0 & \text{otherwise} \end{cases}.$$

This suggests an ownership function $v(s, i)$ for any board states s and board intersection i,

$$v(s, i) = \begin{cases} 1 & i \text{ is owned by the player in the end-game position assuming perfect play from state } s \\ -1 & i \text{ is owned by the opponent in the end-game position assuming perfect play from state } s \\ 0 & \text{otherwise} \end{cases}$$

where perfect play is assumed for both players.

And the value function $V(s)$ for any board states s is defined by,

$$V(s) = \sum_{i \in B} v(s, i).$$

This value function $V(s)$ has merits over the widely-used "winning probability" $W(s)$,

1. It can ensure a natural playing style (in the sense of maximising territory) even when the winning probability is 100%.
2. It can easily deal with different komis.
3. It is more similar to the intuition of human players, and this can be helpful in Go education and human studying.

One of the possible drawbacks is the winning probability is often reported to be a bit lower when $V(s)$ is used as the value function, because we can only approximate the perfect $V(s)$. However, we will show there are multiple ways to train $V(s)$, and better training can lead to better results.

2 Training the value function

METHOD 1 (fit end-game state): The ownership function $v(s, i)$ can be trained in a straightforward way. For a game G where the end-game state s_e is known, we have the approximation for any state $s \in G$,

$$v(s, i) \approx v(s_e, i).$$

So we can train $v(s,\, i)$ to be equal to $v(s_e,\, i)$, with a learning speed related to the number of moves passed (because $v(s,\, i)$ gets closer to $v(s_e,\, i)$ as we move closer to the end-game).

Here biases are introduced because both players are not perfect. However, because v is an finer function than V (which is already finer than W), the bias is better controlled than the case of W, and we can use all states in the game to train our network, instead of just picking 1 state in each game to avoid over-fitting.

METHOD 2 (fit moves): Giving a board state s where the player is to play, let i be the move the players chooses (in a human professional game, or, after MCTS in a self-play game). Let s_{+i} be the board state after the player played i. If we assume the player is perfect, i.e. the move i is the best possible move, then we have,

$$V(s_{+i}) \geqslant V(s_{+j}) \text{ for all possible moves } j.$$

Hence a method to train $V(s)$ is,

$$\text{Train } V(s_{+j}) \text{ to be equal to } V(s_{+i}), \text{ if } V(s_{+j}) > V(s_{+i}).$$

And a method to write this in terms of $v(s,\, k)$ is,

$$\text{Train } v(s_{+j},\, k) \text{ to be equal to } v(s_{+j},\, k) + \frac{V(s_{+i}) - V(s_{+j})}{361} \text{ for all } k \text{ , if } V(s_{+j}) > V(s_{+i}).$$

And there can also be some more aggressive methods, such as,

$$\text{Train } v(s_{+j},\, k) \text{ to be equal to } \min(v(s_{+j},\, k), v(s_{+i},\, k)) \text{ , for all } k \text{ and } j \neq i.$$

The above method is a bit too aggressive, and one can create many interesting methods in between, using the distribution of $v(s_{+j},\, k)$ and $v(s_{+i},\, k)$ (for example, just reducing some of the k such that $V(s_{+j})$ becomes same as $V(s_{+i})$). Moreover, we can also train $V(s_{+i})$ to be a higher value if there are j such that $V(s_{+j}) > V(s_{+i})$.

METHOD 3 (fit minimax property): The perfect value function shall have the minimax property in the obvious way. So we can train our $V(s)$ to satisfy the minimax property as well.

In fact, one can train it such that a shallow-level MCTS gives as close a result as a deeper-level MCTS. This can be regarded as bootstrapping. And the program is less likely to develop bad habits than applying RL on self-play results.

To summarise, we show it is possible to train a value function directly, without the appearance of the policy network. And one can combine the above 3 methods to give better results.

3 Training results

[TODO: I AM COLLECTING DATA TO DO SOME TESTS...]