Panda's Data Generation Method

Sebastiano Rebonato-Scott, April 2025

Abstract

Panda generates data using self-play. The data used to train Panda's value network comes in three parts: a representation of the position, an evaluation of the position, and a score corresponding to the result of the game (1.0 for a white win, 0.5 for a draw, 0.0 for a black win). The unique aspect of Panda's approach comes in the way it assigns evaluations to positions. The typical approach is to assign to each position that occurred in the game a value which is equal to the evaluation returned by the search function on that node. Panda does assign these values to nodes initially, but then tries to improve upon the accuracy of these evaluations using the information which was later obtained by playing out the game. To do this, it backtracks through the search tree, re-evaluating positions which it deems were mis-evaluated.

A few disclaimers:

- this algorithm is essentially theoretical. Generating sufficiently large amounts of data to train a value network even once takes long enough. I don't have the compute to generate several sets of data, some using this algorithm and some not, and compare results. As such, there is no empirical evidence to suggest that training a value network on data which uses this algorithm will gain elo compared to training a value network on data which does not. The reason I use it is because I want to experiment with some unique ideas in my chess engine.
- the algorithm (like all chess search algorithms) is heuristic. The methods used to identify mis-evaluated nodes and to re-score them are intuitive but not theoretically sound.
- the purpose of this document is to collect/formalise my ideas of the algorithm, not to propose that other engines use this method.

Finding Mis-evaluations

The backtracking algorithm is based on the following assumption: if the value, v, assigned to a node is accurate and the move chosen at that response was optimal, then there should be no response which achieves a better value for the opponent than $-v$. In the case of a perfect search (i.e. a search which can evaluate the whole game tree), this should hold for any node. However, due to the very imperfect nature of the search used by chess engines, there is an asymmetry: positions are evaluated with a slight bias towards the side to move. This is because the moves of the side to move in the position are those searched at the highest depth, and therefore are less likely to be pruned (skipped) based on heuristics. As such, strong responses for our opponent on the next turn are more likely to be skipped than strong moves for us on this turn. If, despite this asymmetry, a node still satisfies this condition, it is defined as *definitely accurate*.

N.B. here, all nodes are scored from the perspective of the side to move.

Consider three successive nodes in the search tree: A, B and C, with assigned values v_A, v_B and v_C respectively.

We define A as:

- definitely mis-evaluated if $v_C \leq -v_B \leq v_A$. Here v_B is *definitely accurate* and v_A is not, so A has been mis-evaluated.
- maybe mis-evaluated if $-v_B \leq v_C < v_A$. Here neither v_A nor v_B is *definitely accurate*, so we should use v_C to determine which to trust.

Note that we do not consider nodes where $-v_B < v_A \leq v_C$ to be possibly mis-evaluated as here the fact that $-v_B < v_A$ can be attributed to the asymmetry of our imperfect search (described above).

Here are some examples:

1. We assign the value of A as +1. Our opponent assigns the value of −1.5 of the position after our move and plays a response. Now that these two moves have been made on the board, we assign the value −2 to C.
2. We assign the value +1 to A. Our opponent assigns 0.0 to the position after our move and plays a response. We now assign +0.1 to C.

Consider first example 1: it seems very likely that either our evaluation of A was inaccurate, or the move which we chose at A was suboptimal. In example 2, even though neither v_A nor v_B is *definitely accurate*, v_C is much closer to v_B than v_A. So intuitively it seems that v_A is wrong.

In cases such as example 2, Panda uses a probabilistic approach. We define $S(x)$ as the expected score from the game given the evaluation of the position x. In the case of example 2, we generate a random number from a uniform distribution between $-v_B$ and v_A. If this number is greater than v_C then the node is deemed mis-evaluated.

Re-scoring mis-evaluations

An important assumption made when determining that the node was mis-evaluated was the assumption that the move chosen at that node was optimal. We should first test this. We do this by running a search with all moves except the move chosen at A considered, and returning the best move, m, from this search. We now make the move m on the board and compute the value, u, of the resulting position. Here, if $-u$ > $-v_B$, then we have reason to believe that our alternative move was in fact better, so we should assign $-u$ to node A. Otherwise, we have no reason to believe that our chosen move was suboptimal, so so should assign $-v_B$ to A.