Deep Reinforcement Learning For Chess

David Yu-Tung Hui, Trinity College, dyth2@cam.ac.uk

February 2, 2018

Director of Studies: Dr. Sean Holden
Project Supervisor: Dr. Sean Holden
Project Overseers: Dr. Markus Kuhn, Pr. Peter Sewell

The Chess engine relies on three components: a search algorithm, an evaluation function and a learning algorithm. The evaluation function scores a chessboard with a deep network. These scores can be used in a search algorithm to find optimal moves and strategies. Reinforcement Learning is then used to train and optimise these two components. As of now, all of them have been implemented and are functional.

A successful project would be able to beat an agent playing random moves 95% of the time, within 80 plys. As current progress is around 60%, the goal can be met through incremental gains of each component. Extension goals have not yet been met because of their dependency to the success criterion.

Other deliverables were a proof of concept pole-balancing agent, a random agent which the chess engine plays against and scripts which evaluate the performance of the agent. All of these have been successfully implemented, apart from the pole-balancing agent. This is because it is a continuous-time problem, whereas Chess is discrete-time. As pole-balancing would require a different architecture to that of a Chess agent, this agent was not implemented.

A starting point was modifying the Giraffe Chess engine, from which the above engine architecture was derived. Written in C++, a Python backend was incorporated. This enabled Giraffe to use the fast processing power of a GPU in Deep Learning. The network architecture and training algorithms were written in the PyTorch deep learning library. To speed up the development process as well as evaluation, a Python entry point was adapted from Sunfish, another Chess engine. This required the network architecture and chessboard processing to be adapted from Giraffe. A Python entry point is a strong interface between the main function and the deep network, as well as the ability to quickly test and debug algorithms using the Python interpreter.

The MiniMax searching algorithm has been implemented and is used in training – done by self-play – and testing against the random agent. Efficiency can be improved through implementing heuristic variations such as alpha-beta pruning or MTD-Bi which reduce the size of the search tree. Many different feedforward network architectures have been implemented for use in the evaluation function. In order to find the optimal architecture – which balances size and learning rate, an adaptive network algorithm such as cascade correlation can be used. Finally, three Reinforcement Learning algorithms have been implemented: TD-Learning, TD-Leaf and TreeStrap, with TreeStrap the most effective.

TreeStrap, combined with MTD-Bi and better network architecture would greatly improve the power of the agent such that the success criterion can be met.