Computer Science Tripos, Part II, Project Proposal

David Yu-Tung Hui, Trinity College

October 13, 2017

Deep Reinforcement Learning For Chess

Combining Rule-Based Search Algorithms with Deep Reinforcement Learning in the Context of Chess

Originator: David Yu-Tung Hui
Project Supervisor: Dr. Sean Holden
Project Overseers: Dr. Markus Kuhn, Pr. Peter Sewell

Introduction

John McCarthy, a founding father of Artificial Intelligence, considered Chess to be its *Drosophila Melanogaster*. [7] This was because Chess is a game where all the rules are known, yet is large enough not to be trivially solved. Training an artificially intelligent agent on Chess enables analysis of its effectiveness over a wide variety of strategies and situations. Many different chess-playing algorithms are begin constantly developed, even today, making it ubiquitous to understanding intelligent systems, just as the common fruitfly is to the biologist.

During his time in Bletchley Park, Alan Turing would often play chess, reasoning that his thought processes could be represented by a Turing Machine. [8] He, along with John von Neumann and Claude Shannon, would independently derive the *Minimax algorithm* to find the best possible next move. [5] [8] Used by nearly all chess engines, including the world-champion-beating Deep Blue, minimax analyses *game trees*, a subset of all possible game futures generated from *searching algorithms* [3].

As the game can produce very deep search trees, minimax is often limited by a specified depth. [12] A shortcoming of this approach is its reliance on an evaluation function used to determine the likelihood of victory given a state. [5] Many Chess engines rely on hand-crafted evaluation functions which may take years and sometimes a decade to develop. [4] Recent developments in Artificial Intelligence have shown that *Deep Learning* can be used to approximate such functions given enough labelled datapoints. [2] To reduce the size of datasets used, *Reinforcement Learning* algorithms enable a chess agent to learn by playing against itself. [5]

Inspired by behavioural psychology, [14] Reinforcement Learning is a promising methodology which has trained agents such as TD-Gammon or Deep-Q-Networks, which can outperform humans in Backgammon and Atari games. [15] [10] Both these architectures rely on one neural network. In contrast to this, AlphaGo, the engine which beat the World Champion in Go, is a hybrid architecture which combines reinforcement learning with minimax and searching algorithms. [13]

Such hybrid architectures harness the efficiency of classical searching algorithms with the large-scale parallel processing inherent within Deep Learning. This also means that the weaknesses of both strategies are apparent, such as the tendency of Deep Learning to overfit, or for search algorithms to generate large search trees. To analyse the effectiveness of this hybrid architecture, I will reimplement *Giraffe*, a

predecessor of AlphaGo. Developed by an AlphaGo Scientist, Matthew Lai, Giraffe was trained on a CPU in Chess, achieving a strength comparable to that of a human grandmaster. [5]

Upon reimplementing Giraffe to be executed on a GPU, a successful agent would beat an opponent playing random moves within 40 rounds, to 95 % significance. I will also estimate the agent's strength with ELO, and evaluate finer aspects of its performance in the Strategic Test Suite. This suite is made up of board positions chess engines typically find difficult to play against, making it a more fine grained metric. [6]

If there are no problems in getting this far, an optional extension to this project would be to investigate the effects of changing the searching algorithm which generates the game tree. Giraffe is constructed around probabilistic search, whereas AlphaGo is based on Monte-Carlo Tree Search. [5] [13] Changing the searching algorithm and creating a successful agent similar to that of AlphaGo would highlight the flexibility of this architecture.

Overall, combining rule based algorithms with Deep Learning would unify the strengths of rule-based analysis with statistical data-based inference, perhaps providing a foundation of the direction of research to come. It is possible that this new architecture may yield a new strategy for playing Chess, enhancing the knowledge of the game as well as providing a new opponent for humans to train against.

Structure

Work for the dissertation project can be categorised into five main bodies of work.

During *preparation*, I will investigate background material by studying key papers and courses around Chess and Reinforcement Learning. [5] [13] [15] [1] This would enable me to structure the rest of the project and decide which languages, libraries, board representations and strategies to implement. This phase should conclude with the installation of all prerequisite software.

Implementation of the core work would then commence. As Giraffe is open-source but optimised for a CPU [5], I aim to reuse as much existing code as possible and adapt it for a GPU. Currently, I am planning to use Python and PyTorch for deep learning, a library which utilises the GPU to dynamically compute weights. [11] During development, I shall back up my code using .git and upload my code and trained network weights to GitHub and Google Drive respectively. I shall have a 30 minute supervision with my supervisor to discuss progress.

Upon completion of implementation, I will *validate* the agent by comparing its strength against different opponents. As there are many strategies for training an agent, I anticipate that software development would follow a spiral or evolutionary model of code development.

If there is time, an optional *extension* can be implemented, adding one more spiral to the engineering process. I plan to modify Giraffe and make it similar to that of AlphaGo. In addition, I would conduct user studies into how the agent plays. The validation criterion of the new agent will be identical to that described above, but should not affect how the project is assessed.

Finally, I will summarise and document the project and its outcome in a *dissertation*. Although I plan to document and write up my progress as I go, the final few weeks of the year will be used to condense and finalise the description of work undertaken.

Starting Point

Languages, Packages and Pre-Existing Code

Giraffe The original implementation of Giraffe is open sourced online and is free to download. A good starting point would be to compile and train the Giraffe agent, verifying that it is in working condition by generating the weights of a neural network. If my GPU implementation fails to change quickly, it may be possible to use these weights to bootstrap training.

Gaviota An open source dataset comprising of 5 million (board, distance-to-checkmate) pairs. The distance-to-checkmate is a metric of the value of the board. I may choose not to use this dataset because it can also be generated by random walks from the root node. [9] [5]

Python, PyTorch Used to construct and train the deep networks. The PyTorch package and Python language provide a nice balance between customisation and ease of use. As the main complexity will be successfully training the agent, the Python language is simple and brief enough such that it can be quickly modified, enabling faster experimentation.

C++, CUDA Used to construct the backend containing the board representation and the rules of chess. This will be a modification of the Giraffe source code. In addition, C++ can be used to create custom Deep Learning functions within PyTorch, such as custom layers, activation, loss or reward functions. To use the GPU, it is possible that CUDA may be used.

HTML, Javascript, Flask, Bottle Used to create a GUI in the optional extension such that a human can easily play against the agent. HTML and Javascript are used to create a webapp, enabling the game to be played remotely. I would use Flask and Bottle to enable the agent to react with actions done from a browser.

In order for the engine to play against other chess engines, middleware will be designed such that the engines can communicate with each other.

Tripos Courses

Artificial Intelligence, Part IB The course introduces the concept of an *agent*, as well as the two principle components used in this dissertation: *Searching Algorithms* and the *Backpropagation Algorithm*. Introduced in the concept of *Supervised Learning*, the backpropagation algorithm is central to deep learning, albeit in a slightly modified form. Knowing how Backpropagation works will be crucial to understanding how Reinforcement Learning algorithms modify supervised learning to train agents.

Machine Learning and Baysean Inference, Part II Expanding on Artificial Intelligence, this course provides an introduction to *Reinforcement Learning* and *Temporal Difference Learning*, two central algorithms to the project. As this is a Lent course, and these are new content for this year, I will have to read other literature as well as last year's lecture notes to understand the algorithms.

These two courses would provide a solid foundation for me to read the original TD-Gammon, KnightCap, AlphaGo and Giraffe papers to gain an insight into a successful algorithm.

Algorithms, Part IA In order to analyse the complexity of probabilistic search and Monte-Carlo tree search, concepts taught in the Algorithms Course to analyse the runtime and asymptotic complexity of algorithms would be a solid foundation. This course also introduces search algorithms such as depth-first search, breadth-first search and iterative-deepening search. It would be worthwile to compare the runtime and effectiveness of these against probabilistic search and Monte-Carlo tree search.

Programming in C / C++, Part IB Many chess engines and position evaluators are implemented in C or C++, such as Giraffe. This course introduces operators in both languages and describes their function and utility. Low level knowledge of its functionality will be useful to isolate parts of the Giraffe codebase I need. In addition, should I wish to design custom Deep Learning functions, this would be based in C++.

Computer Design, Part IB Due to the increasing cost-effectiveness of GPUs, a recent trend within Deep Learning is to implement them by parallel processing within GPUs. This course introduces the architecture of a GPU and how it can be leveraged using *CUDA*. Although the majority of the implementation will be abstracted away by a deep learning library, knowledge of CUDA will help with creating and debugging custom Deep Learning functions.

Concurrent and Distributed Systems, Part IB Many agents can be trained simultaneously by playing each other. As Chess is a turn by turn game, concurrency errors may result in a failed move, a simultaneous move or an erroneous move. This course introduces strategies, design patterns and algorithms for preventing such errors, and would thereby be critical when training the agent against itself.

Human-Computer Interaction, Part II To obtain feedback about the chess engines, an extension experiment is to conduct user feedback about the chess engine. Should I wish to conduct this experiment, Human-Computer Interaction describes how to design robust user-studies and tests. As this is also a Lent Course, I will read last year's lecture notes if I choose to implement the extension work.

In addition to these, **Software Engineering, Part IB** and **Mathematical Methods for Computer Science, Part IB** will have introduced the discipline of software engineering and mathematical notation which would be useful in understanding and implementing the project.

Substance

The finished codebase will consist of a trained agent, capable of playing against chess engines and humans using the syntax of GNUChess on the command line. The agent runs on top of the internal board representation, which contains a set of rules and constraints limiting the agent to legal moves. This can be achieved by adapting the open-source Giraffe codebase to be optimised on a GPU. Optional project extensions add a GUI to increase interaction, as well as the code of another agent similar to AlphaGo.

At the centre of the agent is *Probability-Limited Search*. This search algorithm creates a *game tree* from the current state. All possible games are simulated from the current board position and terminated when the probability of principal variation dips below a threshold value. Principal variation is a change in the nature of the game, such as a transition between midgame and endgame, where only 3-4 pieces remain, or in checkmate, when one player captures the opponent's king and wins. *Minimax* is run on top of this game tree to find the next best move. The best move maximises the minimum values your opponent can force from their subsequent move after yours. The primary problem with this algorithm is the lack of knowledge on whether there exists a better move outside the tree.

Built around this are two *deep neural networks*, the quality of which are the most important part of the project. These will have to be coded efficiently such that training and computation can run as quickly as possible. As Chess is a game with limited thinking time, a faster network could analyse more nodes within the same unit time, making the search more comprehensive. One neural networks approximates the evaluation function of the board and is known as the *value network*. The other estimates the probability of principle variation, generating the search tree.

The whole agent itself is trained in two phases. First, (board, distance-to-checkmate) pairs pre-train the network by *supervised learning*. The distance-to-checkmate, the number of moves until checkmate, is an initial approximation of the value of the board. Pre-training enables the network to obtain initial weights detailing which moves are better than others.

During the second phase of training, the agent plays against itself, and updates the weights of the network using *Temporal difference learning*, a technique within *Reinforcement Learning*. A classic Reinforcement Learning problem is the tradeoff between *exploitation vs exploration*. In order to find the optimal strategy, the agent must decide and exploit the best strategy out of numerous explored options. By randomly generating a move during training or by adding or removing pieces to the board, exploration is achieved through increasing the number of boards the agent trained on. *Memoisation* can be used to keep track of trained board positions to generate hitherto unseen board positions. [5]

The AlphaGo architecture has two fundamental differences. First, instead of Probability-Limited Search, *Monte Carlo Tree Search* is used instead. Instead of the branches of the tree being limited by probability, they are limited by a *policy network*. For each board, the policy network generates three possible next moves, greatly reducing the *branching factor*, or breadth of the search tree. [13]

In addition to this, a substantial portion of the Dissertation will also be on analysis, the details of which are discussed in the next section.

Evaluation

The aim of this project is to determine the effectiveness of hybrid Deep Learning architectures using Chess. The agent should be robust enough such that it can produce enough data to measure the following:

Estimating ELO ELO is an international rating used to estimate the strength of a competitor. The most advanced chess engines have 3200 ELO whereas an amateur has 1000. [5] Two methods of estimating the agent's strength is with the OrdoELO or BayesELO test suites.

Strategic Test Suite A more fine grained metric of measuring the strength of an agent is by testing it on the Strategic Test Suite. Comprising of board positions that chess engines typically find difficult, agents are score by how accurately and how quickly they recognise the next optimal move. [6]

User Studies Optional user studies enable analysis of the style, aesthetics and the humanlike qualities of the games played.

In addition to the above, running the agent also enables comparison between modern-day commercially available architectures and the dedicated hardware of thirty years ago. In addition, it is possible to compare the difference in quality between running the agent with and without a GPU.

Although all of the above metrics are viable measures of the agent's quality, none explicitly state outright that the agent has been successfully trained.

Success Criterion

As the agent will be trained by playing against itself, it is more likely that it knows how to exploit situations when its opponent is playing well, rather than against any opponent. To ensure that the agent is strong in all situations, its performance will be evaluated against an opponent playing randomly generated moves.

Success The agent can beat an opponent playing randomly-generated moves within 40 rounds, to 95 % significance. More formally, 40 rounds is equivalent to 80 plys, where a ply is a turn taken by one of the players. These values were chosen because the mean length of a chess match is around 80 plys, and 5 % is the lowest commonly used significance threshold for p-values. The same criterion will be used to evaluate the agent with the AlphaGo architecture.

Plan of work

I have divided up the time between the start of the year and the submission date into 16 different sprints, named after the technique in agile code development. Buffer sprints are laid out around deadlines to allow margin for areas of code development which may take longer than anticipated.

Sprint 1: 7^{th} October – 20^{th} October

- Submit project proposal by 20^{th} October.
- Investigate, identify and install all relevant software into all development machines.
- Start git repository and sync to GitHub for backing up software.
- Start documentation log for work done this week.

Milestone: Submit project proposal report.

Sprint 2: 21st October – 3rd November

- Clone and install Giraffe open-source code, build on own machine and investigate how it works.
- Read relevant papers, including TD-Gammon, KnightCap, AlphaGo and Giraffe.

Milestone: Run and train a working model of Giraffe.

Sprint 3: 4th November – 17th November

- Identify and isolate the board representation and chess rule constrainer portion of Giraffe.
- Learn and develop how to train reinforcement learning agents in PyTorch, with the OpenAI Gym.
- Begin to reimplement Giraffe's deep networks with PyTorch.

Milestone: Train an agent to play OpenAI's Pendulum.

Sprint 4: 18th November – 1st December

- Finish implementing Giraffe's deep network.
- Pre-train the network with Gaviota's (board, distance-to-checkmate) pairs.
- Begin to train the agent.

Milestone: Pre-train the network with Gaviota.

Sprint 5: 3nd December – 15th December

- Implement the evaluator: the opponent who plays with randomly-generated moves.
- Successfully train the agent with Reinforcement Learning.
- Evaluate the performance of the agent.

Milestone: View the evaluation of the trained agent.

Sprint 6: 16th December – 29th December

- Continue to train Giraffe by playing it against itself.
- Evaluate Giraffe with the Strategic Test Suite
- Evaluate Giraffe by estimating its ELO

Milestone: Successfully train Giraffe to pass the success criterion.

Sprint 7: 30th December – 12th January

- Conduct experiments into how running Giraffe on CPU differs from GPU
- Analyse data from the Strategic Test Suite and ELO ratings
- *Extension*: Conduct User Studies into Giraffe.

Milestone: Successfully train Giraffe to pass the success criterion.

Sprint 8: 13th January – 26th January

- Produce a draft of the progress report for feedback from supervisor.
- *Extension*: Conduct User Studies into Giraffe.

Milestone: Obtain progress report feedback from supervisor.

Sprint 9: 27th January – 9th February

- Finish and submit the progress report.
- *Extension*: Implement Monte-Carlo Tree Search.
- *Extension*: Begin to implement and train policy network.

Milestone: Submit progress report by Fri 2 Feb 2018 (12 noon)

Sprint 10: 10th February – 23rd February

- Prepare slides and a five minute presentation detailing the project.
- *Extension*: Finish implementing and training the policy network.

Milestone: Give the Progress Report Presentation

Sprints 11 and 12: 24th February – 23rd March

- Buffer sprints, finish the optional extensions if started.

Milestone: Make final changes to the codebase

Sprint 13: 24th March – 6th April

- Begin writing the dissertation.
- Select important notes and documents from logbook to include in the dissertation.

Milestone: Write first draft of dissertation

Sprint 14: 7th April – 20th April

- Clean up, review and finalise the dissertation.

Milestone: Submit dissertation for feedback from supervisor.

Sprint 15: 21st April – 4th May

- Amend dissertation with feedback from supervisor.

Milestone: Submit dissertation if completed.

Sprint 16: 5th May – 18th May

- Buffer sprint in case dissertation is not adequately finished.

Milestone: Submit dissertation by deadline, Fri 18 May 2018.

Resource Declaration

Macbook Pro, late 2015 (personal machine) El Capitan, 16 GB RAM, 2.5 GHz Intel Core i7, 500 GB SSD: Used for code development, dissertation writing. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Desktop (personal machine) Ubuntu, 32 GB RAM, 3.6 GHz Intel Core i7, Nvidia GeForce GTX 1080, 256 GB SSD: Used for code development, agent training, running and evaluation. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

To prevent loss of data, I shall regularly sync my code with GitHub and save trained network weights to Google Drive. In the event of hardware failure, I transfer the code and data over to the following device.

Special hardware from the Computer Lab A desktop computer with an Nvidia GPU, preferably with 8 GB or more: If set up in the Intel Lab, it would be a good location for user-studies to be held. In addition, if my personal hardware fails, this would be used for agent training.

References

[1] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. *KnightCap: A chess program that learns by combining TD(lambda) with game-tree search.* arXiv:cs/9901002 [cs.LG], 1999.

[2] Christopher Bishop. *Pattern Recognition and Machine Learning.* Springer, 2009.

[3] Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. *Deep Blue chess algorithm.* `www.tcm.phy.cam.ac.uk/BIG/pob24/deep\_blue.pdf`, 2002.

[4] Stockfish Community. About stockfish. `https://stockfishchess.org/about/`. Accessed: 2017-10-13.

[5] Matthew Lai. *Giraffe: Using Deep Reinforcement Learning to Play Chess.* arXiv:1509.01549 [cs.AI], 2015.

[6] Thomas Mayer. *STS 1.0 revisited.* Computer Chess Club, 2010.

[7] John McCarthy. *Chess as the Drosophila of AI.* `http://jmc.stanford.edu/articles/drosophila/drosophila.pdf`, 1997.

[8] Donald Michie. *On Machine Intelligence.* Ellis Horwood Limited Publishers, 1986.

[9] michiguel. Gaviota tablebases. `https://github.com/michiguel/Gaviota-Tablebases`. Accessed: 2017-10-13.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. *Human-level control through deep reinforcement learning.* Nature, 2015.

[11] PyTorch. Pytorch. `http://pytorch.org/about/`. Accessed: 2017-10-13.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education Inc., 2010.

[13] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. *Mastering the game of Go with deep neural networks and tree search.* Nature, 2016.

[14] Richard Sutton and Andrew Barto. *Reinforcement Learning, An Introduction.* The MIT Press, 1998.

[15] Gerald Tesauro. *Temporal Difference Learning and TD-Gammon.* The MIT Press, 1998.