Computer Science Tripos, Part II, Project Proposal

David Yu-Tung Hui, Trinity College

October 20, 2017

Deep Reinforcement Learning For Chess

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

Introduction

Recent advances in Artificial Intelligence combine the strengths of two subfields to produce powerful hybrid agents. Such agent architectures include AlphaGo, the first computer program to play Go with superhuman ability. [6] By harnessing Deep Reinforcement Learning with classical search algorithms, hybrid architectures possess the advantages, as well as weaknesses, of both fields.

To analyse such architectures, I will reimplement *Giraffe*, a predecessor of AlphaGo. Developed by Matthew Lai, Giraffe was trained on a CPU in Chess, achieving a strength of an international master. [2] My reimplementation would run on a GPU, making it one of the first chess engines to do so. With the increasing cost-effectiveness of GPUs, speedups in Deep Learning are attained and are becoming more popular. However, they are rarely used in Chess because most agents rely on hand-crafted search and evaluation algorithms with no machine learning. Optional further extensions would modify the searching algorithm and conduct empirical user studies.

Combining rule based algorithms with Deep Learning would unify many years of orthogonal research, and perhaps provide a foundation of the direction to come. It is possible that this new architecture may yield a new strategy within the game of Chess, enhancing the knowledge of the game as well as providing a new opponent for humans to train against.

Substance and Structure

The finished codebase will consist of an agent capable of playing against chess engines and humans using the GNUChess / XBoard Communication Protocol. Deep Learning networks run on top of an internal board representation, adapted from Giraffe. This contains rules and constraints on the legality of moves. Optional project extensions add another searching algorithm for comparison. In addition to the code, analysis of the above techniques will be summarised in the Dissertation.

Giraffe is based around *Probability-Limited Search*. From the current board position, a *game tree* is created by simulating all possible games until the probability of principal variation dips below a threshold value. Principal variation occurs when the nature of the game changes, for instance between midgame and endgame, or in checkmate, when a player wins. *Minimax* runs on the tree to find the optimal move. This is so named because the best move maximises the minimum values your opponent can force.

Built around this are two *deep neural networks*, the quality of which are the most important part of the project. One neural network evaluates the value of a board and is known as the *value network*. The other estimates the probability of principle variation, generating the search tree. As Chess players have timed moves, these will have to be coded efficiently to search as many nodes as possible.

There are two training phases. First, (board, distance-to-checkmate) pairs pre-train the value network by *supervised learning*, where distance-to-checkmate is an initial approximation of value. During the second phase, the agent plays against itself in *Reinforcement Learning*, specifically in *Temporal difference learning*. A classic Reinforcement Learning problem is the *exploitation vs exploration* tradeoff whereupon the agent balances exploiting the optimal strategy from numerous exploratory strategies. Giraffe explores by randomly generating a move during training or by adding or removing pieces to the board. *Memoisation* keeps track of which boards have been seen. [2]

The optional extension implements a search algorithm similar to AlphaGo. It uses *Monte Carlo Tree Search* instead of Probability-Limited Search in Giraffe. Here, the search tree is limited by a *policy network*, which generates three promising next moves from a board, greatly reducing the *branching factor*, or breadth of the search tree. [6]

Overall, the project can be categorised into five stages. During *preparation*, I will investigate papers and courses concerning Chess and Reinforcement Learning. [2] [6] [7] [1] This would enable me to structure the project and select programming languages, libraries, and protocols.

During *implementation* I shall back up my progress by uploading code to Github and trained network weights to Google Drive. Every week, I shall discuss progress with my supervisor during a 30 minute supervision. This, alongside *validation*, would follow a spiral or evolutionary model of code development.

If there is time, an optional *extension* would be implemented, adding more iterations to the engineering process. Validation of the extension should not affect the assessment of the project.

Finally, I will present my findings in a *dissertation*. As I plan to simultaneously document progress, this phase would entail condensing and finalising all materials.

Starting Point

Tripos Courses

Artificial Intelligence, Part IB The course introduces the concept of an *agent* and an *environment*, as well as two principle project components: *Searching Algorithms* and *Backpropagation*. These concepts would be fundamental to understanding an overview of Giraffe and how the structure of the codebase would be laid out.

Machine Learning and Baysean Inference, Part II Expanding on Artificial Intelligence, this course introduces *Reinforcement Learning* and *Temporal Difference Learning*, techniques used in both Giraffe and AlphaGo. As these is a Lent course, I will read other literature alongside last year's lecture notes to understand the algorithms.

These two courses provide a solid foundation for understanding the original TD-Gammon, KnightCap, AlphaGo and Giraffe papers.

Algorithms, Part IA Taught concepts such as runtime and asymptotic complexity would be a solid foundation for comparing Giraffe and AlphaGo with each other, as well as comparing the search algorithms within each engine to commonly used search algorithms.

Programming in C / C++, Part IB Many chess engines and position evaluators are implemented in C or C++. By describing the low-level function and utility of operators this course would be useful when isolating parts of Giraffe that I need. In addition, C++ can be used to create custom Deep Learning functions within PyTorch, such as custom layers, activation, loss or reward functions.

During implementation, the following three courses will provide guidance. **Computer Design, Part IB** introduces the architecture of a GPU and how *CUDA* can be used in computation. **Concurrent and Distributed Systems, Part IB** introduces design patterns to prevent concurrency which may occur during a match, or when the agent trains against itself. Finally, **Human-Computer Interaction, Part II** describes how to design user-studies and tests, providing valuable advice in the extension experiment for empirical user feedback.

Additional Languages, Packages and Datasets

Giraffe A good starting point would be to download and train the open source Giraffe agent, verifying that it is in working condition. If my GPU trains too slowly, this can be used for bootstrapping.

XBoard / WinBoard Communication Protocol XBoard, or WinBoard on PC, is a GUI useful for debugging. It supports the XBoard Communication Protocol, the most popular protocol for chess engines to communicate.

CCRL, Gaviota The Computer Chess Rating Lists specialise in agent to agent training. Games can be downloaded from the website and used to train Giraffe. Each chessboard can then be annotated by Gaviota to produce (board, distance-to-checkmate) pairs used in training. [4] [2]

Python, PyTorch The PyTorch Package runs on top of Python, abstracting GPU processes to construct and train deep networks. As the main complexity of the project is successfully creating a deep network, this ecosystem has adequate flexibility to experiment and implement various network designs. [5]

Evaluation

The most popular metric is **ELO**, an internationally recognised strength metric. The most advanced chess engines have 3200 ELO whereas an amateur human has 1000. [2] This can be estimated using the OrdoELO or BayesELO test suites. At a more fine grained level, the **Strategic Test Suite** tests agents on how quickly and accurately they recognise optimal moves over 10000 positions. [3] In contrast to both, **empirical user studies** analyse style, aesthetics and humanlike qualities of an agent. However, none of the above explicitly measures whether the agent has been successfully trained. This is because an agent is more likely to perform better against strong opponents, rather than against any opponent, because the agent trains by playing itself. Therefore I shall use the following statistical test for evaluation.

Null Hypothesis : Within a chess game of 80 plys, there is no significant difference between a trained agent and one that plays randomly generated moves. A ply is the unit of a turn from a player.

Alternate Hypothesis : Within a chess game of 80 plys, a trained agent performs better than one that plays randomly generated moves.

Significance : 95 %

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. The same criterion will be used to evaluate the optional extension, though will not affect how the project's assessment.

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: 7th October – 20th October

- Submit project proposal by 20th October.
- Investigate, identify and install all relevant software into all development machines.
- Start documentation log for work done this week.

Milestone: Submit project proposal report.

Sprint 2: 21st October – 3rd November

- Start git repository and sync to GitHub for backing up software.
- 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 (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 and estimated 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.

Cambridge HPCS The Cambridge High Performance Computing Services cluster will be used as a backup if all of the above fails.

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

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

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

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

[6] 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.

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