COMP 4200: Artificial Intelligence Project Proposal

By Auris Kveraga, Michael Mitkov

Chess Engine Training Using Support Vector Machines

Problem Statement:

Chess has long been considered a measure of human intelligence, and as a result has been an intellectual battlefield between man and machine since the inception of the AI field. As a game with relatively simple rules and discrete structure but vast domain of potential games, Chess has served as a fascinating barometer for the strength of AI, and a training ground for various approaches to the problem. Many notable minds in the field such as Alan Turing, John von Neumann and Claude Shannon have formulated their own chess engines, but it wasn't until 1997; when IBM's Deep Blue famously defeated World Chess Champion Garry Kasparov, that an AI overcame the best human player. While Deep Blue's win was somewhat controversial, with some (notably including Kasparov himself) doubting the validity of the result, the two decades since have erased any doubt as to who stands atop the heap; if current world champion Magnus Carlsen won a game out of one hundred against any of the top chess engines it would be a massive upset. While today Deep Blue would be considered a weak engine, its influence is clearly seen in most of its descendants (1). Most modern chess engines rely on handwritten evaluation functions, modified and tuned with carefully selected weights. These functions and weights are designed by a cadre of professional chess players and programmers, implemented in conjunction with alpha-beta search, and are contingent on domain-specific heuristics and knowledge (2). The preeminent example of this Deep Blue inspired architecture is Stockfish (3). Lately however a handful of newer engines have circumvented the need to handwrite a large host of evaluation functions and weights, intentionally eschewing any domain-specific knowledge in

favor of using self-play reinforcement learning to learn how to evaluate each position. Google's AlphaZero is the most famous of these newer engines, and is implemented using a deep neural network which takes the board position *s* as its input, and outputs a vector containing move probabilities **p** with the property such that $p_a = \Pr(a|s)$ for each action a, and a scalar value which represents the expected outcome of the game from the position *s*. This allows AlphaZero to eschew any handwritten evaluation functions or domain-specific knowledge, as it optimizes the parameters of its neural network, θ, through reinforcement learning and is able to evaluate its potential actions through predicted probabilities. AlphaZero replaces the traditional alpha-beta search with the Monte Carlo tree search algorithm, where each search is comprised of a series of simulated games, and search depth is prioritized over breadth as opposed to traditional search engines. AlphaZero evaluates several orders of magnitude fewer positions per second (60,000 vs 60 million for Stockfish), but the quality of each move/potential position considered is significantly higher than its competitors; a more human approach to the problem (2). After much research of traditional and newer chess engines, we have observed that both are computationally heavy albeit in somewhat different ways, with traditional engines such as Stockfish relying on incredibly large searches using complex evaluation functions, while reinforcement learning engines such as AlphaZero rely on massive amounts of time and resources spent training their neural networks. We decided that a combination of these approaches could be implemented, one that uses domain-specific evaluation functions, but simultaneously utilizes a learning component to decide the weights and probability of success for each potential action. Our approach will avoid the computationally expensive Neural Network Architecture, employing instead support vector machines (SVM).

<u>Problem Analysis:</u>

We will create an agent with a similar architecture to popular chess engines, using a domain-specific evaluation engine (similar to most conventional engines, e.g. Stockfish) with an added learning component; utilizing an SVM and reinforcement learning through self-play to improve evaluation functions and calculate associated weights. The environment of the agent is the chess board, represented as an encoded 8x8x12 matrix, referred to colloquially as "bitboards" (4). The state of the pieces on the board is represented on the matrix by a 1 if a specific piece is present, or a 0 if it is not; given there are 6 piece types for both black and white, this corresponds to 12 different 8x8 board representations. This is a fully observable, multi-agent environment that is both deterministic and episodic in nature. Because the game pieces are not in continuous movement, state only changes after a single action is performed by the agent after every move, making the environment discrete. The potential states also belong to a vast but finite set; the estimated state space complexity of a chess game is 10^{43}. The entire game of chess is estimated to have 10^{123} game paths (5). We will devise our architecture similar to that of one using a Convolutional Neural Network (CNN), where the decision making process for each chess move is treated as pattern recognition, replacing the traditional CNN with an SVM implementation, which is a machine learning algorithm that is popular for pattern recognition. The state transition function for the agent will accept the movement of a certain piece, bound by the rules of the game (e.g. bishop moves on the diagonal), as an action, and return a new bit board (8x8x12 matrix), which is subsequently shifted by a factor of 1/32, as 2 positions, or pixels, are changed after each action (6). State transitions are deterministic. State-space time and real time are not linked, and transition speed is limited only by the speed of the program (4).The evaluation functions of the agent will be based on traditional chess engines, such as Stockfish, Komodo, and Leela. These evaluation functions are grounded in traditional chess principles, evaluating

positions based primarily on piece material value, positioning (open files, pawn structure, king safety, passed pawns), and game stage (early/mid/end game). The weights of these functions will be learned using a trained SVM model. Evaluations are only performed on "quiet" positions, with no unresolved captures or checks, meaning that evaluation of unresolved positions is delayed until the engine performs a "quiescence search", which resolves the system to a "quiet" state. Stockfish uses a minimax search that evaluates each potential leaf using the quiescence search, and utilizes various pruning strategies in an attempt to prioritize search depth for promising variations. Historically, neural networks have been considered too slow for use with these pruning strategies (alpha-beta pruning), and thus engines have favored domain-specific search optimizations. We will use minimax search in conjunction with an SVM model. The SVM model will provide not only the weights of the evaluation functions, but the probability of success for each given action, which will significantly decrease the search space of the agent for each potential action. We will explore two methods of training the SVM model. The first is inspired by AlphaZero, where the agent will play a series of games against itself, otherwise known as reinforcement training. The second method we will explore is traditional training, where we will perform a train-test split on a large dataset of games played by players of a high rating (referred to as Elo). This data will be taken from https://www.ficsgames.org/. We will then compare these results by having the two play a series of games against each other. This will reveal which method yields better results in terms of strength. The performance of these agents will be measured against the others using several techniques: firstly the agents will play against each other, they will also play against Stockfish, and potentially other popular chess engines. Interfacing with other engines will be accomplished using the UCI interface and the python-chess module. These various competitions will provide an Elo estimate for the agents, which will

then be compared to determine which methods and techniques are superior, and to determine the viability of an SVM learning component.

Conclusion:

The ultimate goal of the project is not to compete with the current engines, but rather to explore new methods and AI concepts in creating a chess agent. We do not expect higher performance than any of the highest performing engines available, especially due to the historic difficulties of implementing a learning component with a CNN, our goal however, is to create several autonomous chess agents, which borrow from various established chess engine's architectures, and to explore the viability of a learning component implemented using the SVM model. Furthermore, the results should help us explore not only the viability of an SVM model, but the efficacy of the possible training techniques used to learn the model (traditional data oriented training, or reinforcement training).

Timetable (tentative):

- February 27th: Environment (bit board, pieces/values, state matrix encoding), development of domain-specific evaluation functions
- March 20th: Development of SVM, data pipeline/gathering, traditional training on data, integration of SVM as learning component
- April 10th: Implement Minimax search, and Alpha Beta pruning. Develop an evaluation environment using the UCI interface. Run agents through evaluation, modify learning component and evaluation functions as needed, create second (or more) agent(s) trained with reinforcement learning

References

1. M. Campbell, A.J. Hoane, F. Hsu, *Artificial Intelligence* **134**, 57 (2002)
2. D. Silver, T. Hubert, J. Schrittweiser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, A General Reinforcement Learning Algorithm That Masters Chess, Shogi and Go Through Self-Play, 2
3. T. Romstad, M. Costalba, J. Kiiski *et al.*, Stockfish: A strong open source chess engine. https://stockfishchess.org/. Retrieved February 2nd, 2020
4. B. Oshri, N. Khandwala, Predicting Moves in Chess using Convolutional Neural Networks
5. Problem and Search Spaces, http://www.cse.uaa.alaska.edu/~afkjm/cs405/handouts/search.pdf, Retrieved January 30th, 2020