The Implementation of Artificial Intelligence and Machine Learning in a Computerized Chess Program

by James “The Godfather” Mannion
Computer Systems, 2008-2009
Period 3

Abstract

Computers have developed to the point where searching through a large set of data to find an optimum value can be done in a matter of seconds. However, there are still many domains (primarily in the realm of game theory) that are too complex to search through with brute force in a reasonable amount of time, and so heuristic searches have been developed to reduce the run time of such searches. That being said, some domains require very complex heuristics in order to be effective. The purpose of this study was to see if a computer could improve its heuristic as it runs more searches. The domain used was the game of chess, which has a very high complexity. The heuristic, or evaluation function, of a chess program needs to be able to accurately quantify the strength of a player's position for any instance of the board. Creating such an evaluation function would be very difficult because there are so many factors that go into determining the strength of a position: the relative value of pieces, the importance of controlling the center, the ability to attack the enemy's stronger pieces – something that chess masters spend entire lives trying to figure out. This study looked to see if it was possible for a computer program to "learn" an effective evaluation function by playing many games, analyzing the results, and modifying its evaluation function accordingly.

Introduction

Heuristic searches (such as the A*) in general can be applied to practically any domain that somebody would want to search through. The actual heuristic functions used, however, are extremely domain-specific. Some problems require very simple heuristics (or estimations that allow a program to effectively and efficiently reach an optimum solution), such as a problem whose domain involves points on a flat surface that are connected by paths, in which the shortest path from point A to point B is desired. Such a problem's heuristic would simply be the distance of a hypothetical straight path from the current point on the path to point B. By adding this estimation to the total distance traveled along a path thus far and comparing it to the total distances and estimated remaining distances of other possible paths, a search program can find the optimum solution to this problem quite easily. This is an example of a domain with a very simple heuristic to calculate. However, many problems that researchers are interested in nowadays have much more complex domains, and therefore much more complex heuristics. The age-old game of chess is one such problem.

In 1997, IBM created a chess program called "Deep Blue" that beat the world champion chess player at the time. Deep Blue used a "brute-force" approach to chess. It would look at every possible move, and then every possible move that could result after each of those original moves, and each third possible move, and so on and so on until every possible move and every possible game had been looked at. And it would calculate all of this before it even made the first move. Each time it would search all the possibilities and make the move that would give the highest chance of checkmate down the road. Even a chess master was no match for Deep Blue. Most chess masters can look about 10-12 moves into the future, but no human has the brain power to play out entire games in their head before making each move. However, IBM needed a state of the art supercomputer to run this program at a reasonable speed. Running it on anything less than a supercomputer would take days or even weeks for the game to finish. On average, there are about 30 moves that a player can make at any given time, and the average game lasts around 50 turns (each turn containing one white and one black move). This means that on the first turn, Deep Blue's brute-force algorithm would have to search through 30^{100} moves, or about 5.15×10^{147} moves. On the second turn it would have to search through about 30^{98} moves, the third 30^{96} moves, and so on and so forth. It is quite clear that using such a method on a machine other than a supercomputer would simply be impractical.

It is possible to write a computer chess program that is still effective without using the brute-force method. By looking only, say, two or three moves into the future rather than 50 moves or more, a program can still make educated decisions about good moves to make if it has some way of estimating how strong a projected position would be. This is where a heuristic, or evaluation function as it is more commonly called in the context of board games like chess, comes in handy. By looking a few moves into the future, applying the evaluation function to each possible board, and choosing a move based on which projected board has the highest strength of position, then a computer can still be an effective chess player, while at the same time dramatically cutting down the number of required calculations.

It sounds simple enough, but once you actually try to create such an evaluation function, it becomes much, much harder. How do you effectively evaluate a position? Do you just look at the number of your pieces versus the number of their pieces? Do you look at whether or not your pieces control the center? Do you look at the possibility of sacrificing one of your own pieces in order to capture a more important piece? An evaluation function can be very complicated to formulate, especially in a game such as chess where there are so many strategical factors to take into account. Chess masters spend entire lifetimes figuring out the best way to evaluate which moves they should make, so at first creating such a function could seem very daunting, especially for someone who has not devoted years to learning the game of chess.

But if chess masters learn how to decide which moves to make by playing a lot of games and learning from their mistakes, why couldn't a computer do the same thing? In fact, computers, although they lack human intuition, can play full games of chess in a matter of seconds, so it is conceivable that they could learn how to effectively evaluate moves much faster than a human could. Therein lies the purpose of this study: to see whether or not a computer can be programmed to not only play chess effectively and efficiently, but also learn from it's mistakes, like humans do, and get better from game to game. The chess program I have developed uses Temporal Difference learning to analyze the games it plays and modify its evaluation function accordingly. This means that theoretically, the program could start out by making random moves, but then modify its evaluation function so that it progresses from simply choosing a random move to actually making an educated choice.

Background

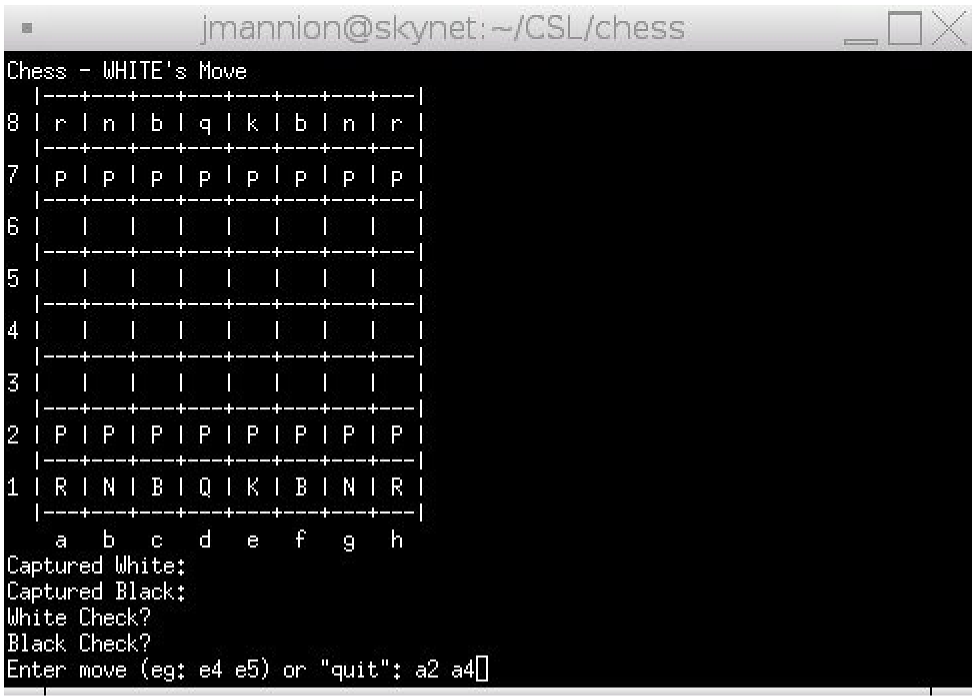
In 1950, Claude Shannon wrote a groundbreaking paper called "Programming a Computer for

Playing Chess." It dealt with various issues, such as how one might go about writing an artificial intelligence program that could play chess well, and what the evaluation of such a program might look like. It discussed the problems involved with brute-force chess algorithms and suggested an AI algorithm that a computer chess program could use. It suggested using a 2-ply algorithm (an algorithm that looks two turns into the future before applying the evaluation function), however if the program finds that a move could lead to a check or checkmate, then it would investigate that move and subsequent moves out to as many as 10 turns. At the end of the paper, Shannon speculated about the possibility of computer programs that can "learn," but noted that such developments were probably man years down the road.

About 50 years later in 1999, D.F. Beal and M.C. Smith published a paper called "Temporal difference learning for heuristic search and game playing." At this point, programs that could play chess had been well-developed, but research in machine learning was really just beginning. The paper investigated the use of Temporal Difference learning as a way of making a computer chess program modify and improve its evaluation function each time it plays. The researchers made their chess program learn for 20,000 games, and then ran their program against a program that had not learned its evaluation function. The learned program performed decisively better over 2000 games than the unlearned program, showing that it is indeed possible for computers to improve their evaluation functions.

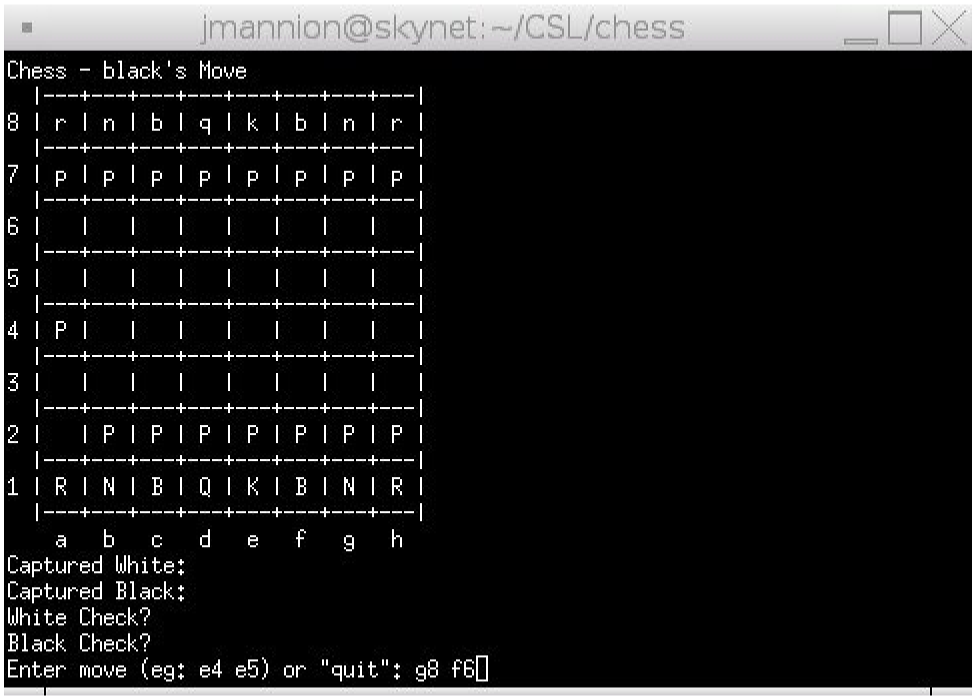
Development

Python was used to code this chess program. The first stage of the program simply involved a console-based chess game where two humans could input their moves into the command line, and the board would be re-printed with the new move, as shown in Figs 1-3:



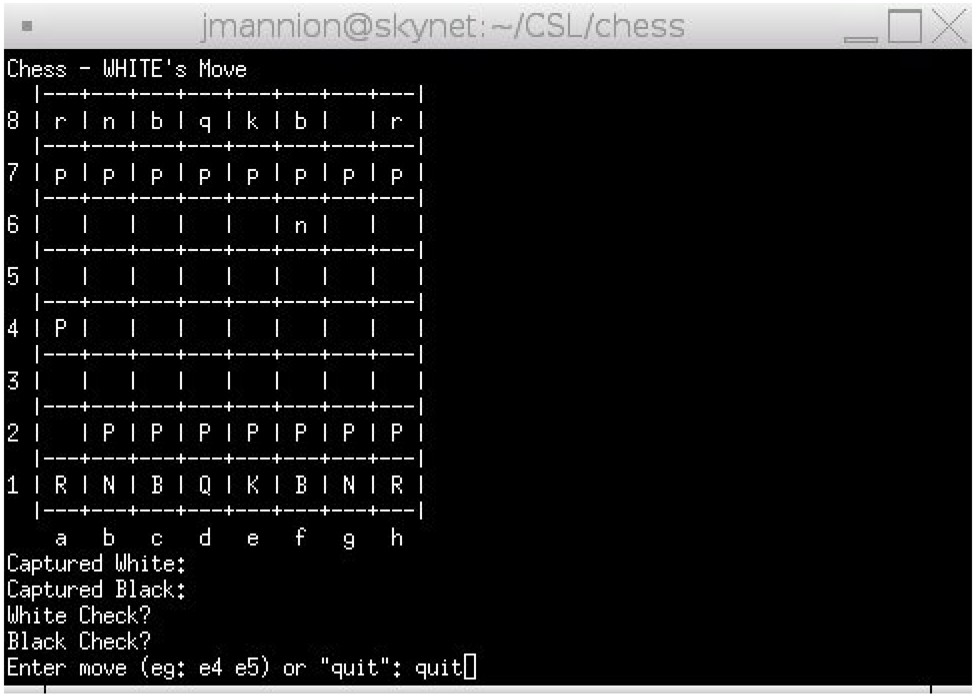
```
Chess - WHITE's Move
  |---+---+---+---+---+---+---+---|
8 | r | n | b | q | k | b | n | r |
  |---+---+---+---+---+---+---+---|
7 | p | p | p | p | p | p | p | p |
  |---+---+---+---+---+---+---+---|
6 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
5 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
4 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
3 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
2 | P | P | P | P | P | P | P | P |
  |---+---+---+---+---+---+---+---|
1 | R | N | B | Q | K | B | N | R |
  |---+---+---+---+---+---+---+---|
    a   b   c   d   e   f   g   h
Captured White:
Captured Black:
White Check?
Black Check?
Enter move (eg: e4 e5) or "quit": a2 a4
```

Fig 1 – The starting board, with white about to move a pawn



```
Chess - black's Move
  |---+---+---+---+---+---+---+---|
8 | r | n | b | q | k | b | n | r |
  |---+---+---+---+---+---+---+---|
7 | p | p | p | p | p | p | p | p |
  |---+---+---+---+---+---+---+---|
6 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
5 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
4 | P |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
3 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
2 |   | P | P | P | P | P | P | P |
  |---+---+---+---+---+---+---+---|
1 | R | N | B | Q | K | B | N | R |
  |---+---+---+---+---+---+---+---|
    a   b   c   d   e   f   g   h
Captured White:
Captured Black:
White Check?
Black Check?
Enter move (eg: e4 e5) or "quit": g8 f6
```

Fig 2 – White pawn moved, black about to move a knight



```
Chess - WHITE's Move
  |---+---+---+---+---+---+---+---|
8 | r | n | b | q | k | b |   | r |
  |---+---+---+---+---+---+---+---|
7 | p | p | p | p | p | p | p | p |
  |---+---+---+---+---+---+---+---|
6 |   |   |   |   |   | n |   |   |
  |---+---+---+---+---+---+---+---|
5 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
4 | P |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
3 |   |   |   |   |   |   |   |   |
  |---+---+---+---+---+---+---+---|
2 |   | P | P | P | P | P | P | P |
  |---+---+---+---+---+---+---+---|
1 | R | N | B | Q | K | B | N | R |
  |---+---+---+---+---+---+---+---|
    a   b   c   d   e   f   g   h
Captured White:
Captured Black:
White Check?
Black Check?
Enter move (eg: e4 e5) or "quit": quit
```

Fig 3 – Black knight moved, white about to quit

References

1. Shannon, Claude E. “Programming a Computer for Playing Chess.” 1950.

2. Beal, D.F. and Smith, M.C. “Temporal Difference Learning for Heuristic Search and Game Playing.” 1999