A Reinforcement Learning Agent for Checkers based on MCTS and TD(λ)

Alexander Pluska

June 15, 2022

Abstract

The goal of this project is to implement a reinforcement learning agent for checkers. The used algorithm is inspired by early versions of AlphaGo [2], although heavy modifications had to be undertaken to accommodate for lack of computational resources. In particular the policy and value network are merged into one and during Monte Carlo Tree Search leaf nodes are evaluated only via value network and not via roll-out.

1 Introduction

Having followed the development of AlphaZero in awe, I thought it would be interesting to mimic some of their ideas for my own reinforcement learning agent. However with a naive single threaded Python implementation the agent needs to spend at least .5s at each position just to get to a reasonable depth during tree search, i.e. one that at least recognizes simple wins as such. This means that each simulated game needs at least 1 minute time. To achieve convergence AlphaZero required many hundreds of thousands simulations [4]. This could be achieved though massive parallelization, incredible computing resources and certainly also a much more efficient implementation. Even though checkers of course is significantly simpler than Go his kind of learning would require many years on my machine.

The first adaptation is instead of two networks, a distinct policy and a valuation network, to use only one valuation network, which would completely determine the policy function via tree search and negamax.

The second adaptation is to use a much more shallow network, with just two convolutional layers, specifically adapted to checkers, followed by a fully connected layer.

In early versions of AlphaGo the agent was bootstrapped by supervised learning on expert moves [2]. However for checkers a database of such expert moves is not readily available and creating it is beyond the scope of the project, so instead the value network is bootstrapped on random positions with material count as a target.

To ensure termination the game is augmented with an additional rule: The game ends immediately in a draw when a position is repeated. Clearly if there exists a winning strategy from a position then there exists one that avoids repetitions, so an optimal policy for the augmented game is also optimal for usual checkers.

Finally with only these adaptations a problem was encountered where the value network would reach an equilibrium evaluating each position as a draw. During the training positions with many kings would be reached from which the agent was unable to convert a win even with overwhelming advantage. Therefore self play is not immediately performed from the stating position but instead from random positions with reduced material, i.e. endgames.

In a funny way this is similar how chess is often first taught to children: At first the values of the different pieces are explained and the children just play to greedily maximize their material count. Then endgames are taught, in particular how to checkmate with reduced material. Finally more complex concepts are learned, of course also through other mechanisms, but also through play.

Additionally I implemented a GUI that allows the user to watch games played by the agent, e.g. during self-play, and also watch games against other agents during evaluation.

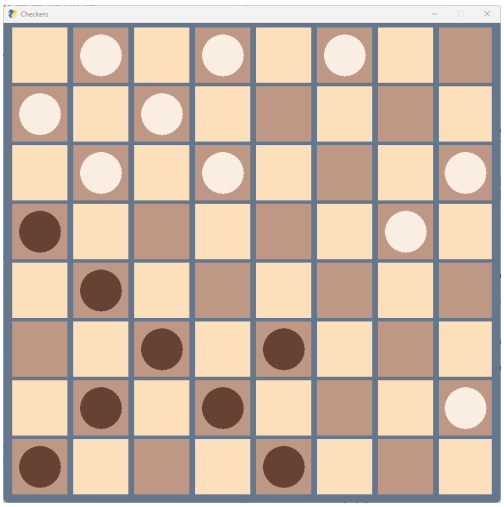

Figure 1: A simple gui to observe games

2 Learning

Before explaining learning let us first discuss in depth how the policy function works. At the foundation lays a CNN with three layers that reflect the geometry of the board. As an input the CNN takes a board position encoded

similarly to the encoding used in AlphaZero [3], i.e. it takes a 4x32 tensor where the first row one-hot encodes the position of dark men, the second of dark kings etc. Then a first convolutional layer is applied with a 12 element kernel and stride of 4, i.e. containing the information of each three consecutive rows, extracting 8 features. The 6 layers are then combined before applying a fully connected linear layer. At each stage ReLU is used as an activation. Finally a Sigmoid function is applied to obtain an evaluation between 0 and 1. During learning adam is used as an optimizer.

Having this net a position is evaluated via an Monte-Carlo-like tree search as follows: First create a node with the initial position and set its number of expansions to 0. Repeatedly expand this node as follows:

- If a node with 0 expansions is to be expanded, add a child to the node for each legal move in its position and expand each of them until no more captures are possible. Then evaluate the leaves with the neural net and compute the implied evaluation of the position via negamax, i.e.
$$ev(\text{node}) = \max\{1 - ev(\text{child}) \mid \text{child} \in \text{children}\}$$
where the evaluation at each node is from the perspective of the party that has the move.

- If a node n already has expansions, expand the child c for which the value of
$$1 - ev(c) + \sqrt{2 \cdot \frac{\log(\text{expansions}(n))}{\text{expansions}(c)}}$$
is maximal, ensuring a trade-off betweeen exploration and exploitation. Finally update the value of the original node accordingly.

Perform expansions until some time limit is reached. Then choose the move associated with the child with minimal evaluation, i.e. minimal predicted value for the opponent.

Learning is performed in three stages:

First the net is initialized to approximate the material value of positions. This is done by supervised learning on 100000 randomly generated positions.

In a next stage 100 Games of self-play and learning are performed from endgame positions. The learning is done on batches of 10 played games using TD(0.9), i.e. after each game a list the positions $p_0, \ldots, p_n$ from the game is used as input data for the training and the targets t_i are computed from the evaluations e_i that were computed during the game by setting t_n as the result of the game and iteratively

$$t_{i-1} = 0.9t_i + 0.1e_{i-1}$$

The learning on batches was done to avoid over-fitting, but was probably insufficient.

Finally the agent learns through self-play from the original position after 2 random moves. Again learning was done in batches of 10. The total number of games played was only 300. Due to unknown issues, perhaps related to garbage collection, the agent would sometimes hang up during evaluation, making longer training infeasible.

Self-play is performed on the cpu while learning is performed on a gpu.

3 Evaluation

For evaluation agents from multiple stages of learning were pitted against each other in a round-robin tournament. There were 7 contestants

- The agent after only the supervised learning (bootstrapped)
- The agent after endgame training (endgame)
- The agents after 100, 200 and 300 games (100games, 200games, 300games)
- A baseline agent playing random moves (random)
- A checkers engine (easy) programmed by Ed Gilbert [1], the dll of which I accessed from python via ctypes. This is a weak engine which can be beaten even by beginners with some effort.

The results of the tournament can be seen below. As one can see by far the strongest version of the engine was the bootstrapped version, which just predicts material.

Results	bootstrapped	endgame	100games	200games	300games	random	easy
bootstrapped	x	1.0	1.0	1.0	1.0	1.0	0.5
endgame	0.0	x	0.0	0.0	0.5	0.0	0.0
100games	0.0	1.0	x	0.5	0.5	1.0	0.0
200games	0.0	1.0	0.5	x	1.0	0.5	0.0
300games	0.0	0.5	0.0	1.0	x	1.0	0.0
random	0.0	0.5	0.0	0.0	0.0	x	0.0
easy	0.5	1.0	1.0	1.0	1.0	1.0	x

4 Takeaway

The poor performance of the agent can be explained by different factors, however the most convincing is in my opinion that 300 games simply dont constitute enough data to train neural network with 500+ weights. While the initial agent from bootstrapping was promising, even drawing the easy engine, it had much more data to work on (100000 positions). In particular since at each stage only 10 games $\sim$ 1000 positions were considered the choice was really between miniscule learning progress or over-fitting, and what ended up happening was probably a mix of both, with a bias to the latter. While python was great for prototyping, it reinforced the bottleneck of not being able to simulate a sufficient number of games. For a future implementation I would choose a much more simple evaluation function, e.g. a single linear layer, that is less prone to over-fitting and focus on providing an optimized and possibly parallelized simulation, such that sufficient data can be generated.

References

[1] Checkerboard. https://github.com/eygilbert/CheckerBoard. Accessed: 2022-06-15.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.