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

Alexander Pluska

November 19, 2024

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 [1], although heavy modifications had to be undertaken to accommodate for lack of computational resources. We demonstrate that the agent learns, but is still far from being competitive with a checkers engine.

1 Introduction

Having followed the development of AlphaZero in awe, I thought it would be interesting to mimic some of their ideas and make my own reinforcement learning agent for a board game. To achieve convergence AlphaZero required many hundreds of thousands simulations [3]. Deepmind's incredible success with AlphaZero relied on massive parallelization, incredible computing resources and certainly also a much more efficient implementation than is feasible for me. Therefore I decided to make a number of adjustments:

- Instead of Chess or Go I decided to tackle Checkers, a game that is still complex enough to be interesting but simple enough to be implemented in a reasonable time frame.
- Instead of two networks, a distinct policy and a valuation network, I decided to use only one valuation network, which would completely determine the policy function via tree search and negamax.

In early versions of AlphaGo the agent was bootstrapped by supervised learning on expert moves [1]. 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. Otherwise the rules are those of American checkers, i.e. Kings can move and capture diagonally forward and backward, but only one step at a time.

2 The Agent

2.1 Evaluation Function

At the core of the agent is a evaluation function that is used to evaluate positions. The evaluation function is a neural network that takes a board position as input and outputs a value between 0 and 1 that is interpreted as the probability of the current player winning from the position. In our case we use a relatively small transformer [4] with 16 layers, 4 heads and an embedding dimension of 128. The outputs of the transformer are passed through a linear layer, averaged and then passed through a sigmoid function to obtain the final evaluation. The transformer contains an embedding for each pair of square and piece, i.e. 32 embeddings for dark men, 32 for dark kings, 32 for light men, 32 for light kings and 32 for empty squares. During evaluation, a position is first converted into 32 embeddings, one for each square, and then passed through the transformer.

2.2 Monte-Carlo Tree Search

Given the evaluation function, the agents policy is determined by a Monte-Carlo-like tree search. The search tree is initialized with the current position as the single root node with 0 expansions and then repeatedly expanded 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 update the evaluation of its parents recursively via negamax, i.e.
$$ev(\text{node}) = \max\{1 - ev(\text{child}) \mid \text{child} \in \text{children}\}.$$

- 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 between exploration and exploitation.

Expansions are performed until a time limit is reached or a forced win is found. The move chosen is the one associated with the child with minimal evaluation, i.e. minimal predicted value for the opponent.

2.3 Learning

Learning is performed in two stages:

First the transformer is initialized to approximate the material value of positions. This is done by supervised learning on 10 million randomly generated positions.

Finally the agent learns through self-play. During each stage the agent plays itself from the initial position + 4 random moves. After 128 games the agent is trained on the resulting positions. The target value for the positions is determined by TD(λ) with $\lambda = 0.9$. I.e. if the game consists of positions $p_0, p_1, \ldots, p_n$ and ended in a win for the current player, then the target value $v(p_i)$ for p_i is given inductively by

$$v(p_i) = \begin{cases} 1 & \text{if } i = n \\ \lambda \cdot v(p_{i+1}) + (1 - \lambda) \cdot v(p_i) & \text{otherwise} \end{cases}$$

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. These were the contestants:

- A baseline agent playing random moves (r)
- The agent after only the supervised learning (0)
- The agents after $1 - 7$ stages of self-play $(1 - 7)$

The results of the tournament can be seen below. The numbers indicate score out of 8 games (win = 1, draw = 0, loss = -1).

	r	0	1	2	3	4	5	6	7	total
r	x	-8	-2	-8	-5	-7	-4	-6	-7	-47
0	8	x	6	5	1	6	5	6	2	39
1	2	-6	x	-2	-7	1	-4	-1	-6	-23
2	8	-5	2	x	-8	2	-2	2	-5	-6
3	5	-1	7	8	x	7	4	7	2	39
4	7	-6	-1	-2	-7	x	-3	0	-4	-16
5	4	-5	4	2	-4	3	x	5	-1	8
6	6	-6	1	-2	-7	0	-5	x	-7	-20
7	7	3	-2	6	5	-2	4	7	x	25

4 Conclusion

Looking at the tournament results there is a number of takeaways:

- There is definitely some kind of forgetting happening, as seen in stages 1 and 4. To alleviate this one could increase the number of games played during self-play or decrease the learning rate.
- The worst performing agent (besides random) is the one directly following the supervised learning on material estimation. This indicates that the policy obtained from TD(λ) meaningfully differs from the one obtained from material estimation.
- The provided training regime was insufficient to outperform the baseline trained on material estimation. However, in stage 3 the agent was able learn a different policy that performer equally to the baseline, which is interesting.
- The biggest bottleneck for training was self-play, each stage taking about 2 hours for just 128 games. This could be alleviated by using a more efficient implementation and more computational resources.
- At least each stage was able to consistently beat a random player, which indicates that the implementation is correct and the agent is learning something.

References

[1] 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.

[2] 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.

[3] 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.

[4] Raghuraman Viswanathan. Attention is all you need. 2017.