Artificial Intelligence Fundamentals: project report
Ninniri Matteo (student ID: 543873)

1 Summary

The project consisted in implementing the AlphaZero algorithm by Silver et al. [1]. To keep this report short, the (fairly complex) inner working of the algorithm is omitted. This report focuses instead on the implementation's architecture, how it differs from the paper's, and the results obtained. This project aims to make the implementation of new games just a matter of finding a good OpenAI gym's board implementation and overriding a couple of methods to access its data.

2 Differences with the original work

- This implementation is single-thread and single GPU.
- Some hyperparameters have been modified (see the Experiments section for further details).
- The number of ResNet layers in the networks is much smaller. The input representation is simplified and only features the current board's stones (although the user is free to personalize it).
- This implementation trains the neural network after it gathers *dataset_sz_threshold* samples through self-play and feeds the entire pool up to that point with batches of size *batch_sz*. The pool is then emptied. The two values are hyperparameters.
- The policy head's loss is replaced by the cross entropy loss, because of the better empirical results it produced. The original Negative Log-Likelihood is still kept commented inside the code.
- Like with AlphaGo Zero, but not AlphaZero, this implementation *can* employ data augmentation.

3 Implementation's architecture

This section will list the most relevant .py files and their iterations. The remaining files are either utilities, training scripts, or gameplay scripts.

- Node.py: implements the Node class which represents a search tree node (hence a board state), complete with their Q, N, and P values (W is not necessary). Implementing a new game basically means inheriting this class and overriding a few of its methods which access some of the gym board's information. It also contains two child classes, specializing in the Tic-Tac-Toe and Connect 4 games. They employ two OpenAI Gym's open source projects heavily modified by me. [1] [2]
- Board.py: it works as a wrapper of the search tree's root in order to make it easier to keep track of it while exploring its children. It also contains the high-level method for performing an action. Implementing the board for a new class is as easy as specifying the Node child class employed.
- Net.py: implements the neural network as described by the AlphaGo Zero paper.
- DataAugmenter.py: implements the data augmenter for Tic-Tac-Toe and Connect 4.

[1] https://github.com/davidcotton/gym-connect4
[2] https://github.com/alfiebeard/tictactoe-gym

- MCTS.py: it implements the methods which, given a board and an agent's neural network, perform: a root-leaf expansion, a full move search from the root (and returns the π vector, as well as the action's dataset sample and the action itself) and finally a method which lets the agent play a game by itself, returning the game's dataset as a byproduct.
- AlphaZeroTrainer.py: contains the fit() method which, given an agent, lets it play a number of games and uses the resulting data to train its network.
- Game.py: contains the utilities to let either humans or trained agents, or a combination of both, play a game given a Board instance.

4 Experiments

The experiments have been carried out on Kaggle's cloud computing service featuring an Nvidia P100.

4.1 Tic-Tac-Toe

The network featured 2 ResNet layers with 16 channels each. The agent has been trained by playing 500 games. Each move used 100 root-leaf expansions. The network was trained once the data pool reached a size of 128 samples and the batch size was 16. The optimizer was Adam, with a learning rate of 0.001. The MCTS' ϵ and α values were respectively 0.1 and 0.03.

4.1.1 Results

After the training phase, the agent was able to draw every single game, which is the best outcome when both players play perfectly. The agent correctly learns that the best strategy is to give priority to the center slot and eventually to the corners. As the second player, it does not fall for "tricks" such as the "double corner attack".

4.2 Connect 4

The agent was trained on 2500 games, split among multiple training days, starting each time from the previously produced checkpoint. It took approximately 60 hours of wall clock time. For the first 1750 games, it was trained using 150 root-leaf expansions and 300 for the remainder. The network employed 8 ResNet layers with 128 channels each. The MCTS' ϵ and α values were 0.1 and 0.03 respectively. The optimizer was AdamW with a learning rate of 0.001, which decreased to 0.0001 after 1250 games, and a weight decay equal to 0.0001.

4.2.1 Results

The agent reached an acceptable level of play. Even before the end of the training, it was able to defeat constantly a human with a decent, but not perfect, knowledge of the game. Despite this, it was still not enough to win against a specialized solver which employed alpha/beta pruning [3]. However, further studies have shown that it was still capable to predict correctly 30 out of 41 moves of a perfect game produced by letting a solver play against itself (run MCTS_surgery_c4.py to perform the experiment). 5 out of the remaining 11 moves had the same value as the move selected by the solver, meaning that it effectively chose 35 optimal moves out of 41. The evolution of the agent was also studied by having the checkpoint produced after 1200 iterations (so halfway through the training) and the last checkpoint

[3] https://connect4.gamesolver.org/

play against each other (run PitConnect4Agents.py for the experiment) for 20 games each (10 as the first player and 10 as the second player). The last checkpoint won 70% of the games it played as player 1, drew 20%, and lost 10%. As player 2, it won all the games it played.

5 Conclusions

Unfortunately, the algorithm is known to be computationally demanding and, as a result, it was not possible to push it further. Despite this, it was shown to be able to learn very well the basics of Connect 4, which is a solved but not exactly easy game. The following are a series of considerations gathered during the realization of the project.

5.1 Exploration is the key

The most hard-learned lesson of this project is probably the fact that AlphaZero is not immune from overfitting and, even worse, it can easily overfit on wrong ideas. This happens in particular if, during training, the MCTS does not explore what its network believes to be a bad move and wins a game despite playing suboptimal moves. The algorithm will reward such blunders, turning them into a bad habit that is hard to correct. The only way to avoid this misbehavior is to allow it to explore what it believes to be bad moves, which is what the Dirichlet noise and the annealing do.

5.2 The number of expansions during training matter

Preliminary results on Tic-Tac-Toe have shown that using only 10 root-leaf expansions for each search during training produced higher training losses. To prove how the number of expansions per move matters, a new agent was trained on the game of Tic-Tac-Toe with the same setup as section 4.1.1, but using 10 expansions per move, and it was tested against the main agent on 100 games (50 as player one and the remaining as player 2). To ensure a fair comparison, the original agent performed 10 expansions per move during the games. The main agent managed to win 5 games, despite the fact that Tic-Tac-Toe is a very easy game to draw, and never lost once, showing how better samples do indeed produce better agents.

It might be interesting, for a future study, to see how much the sample quality matters during training and whether starting the training with a higher number of expansions, giving the agent a good starting point, and slowly reducing the number of searches to accelerate the training time, might actually produce good agents with a fraction of the training time.

6 Appendix

The algorithm is essentially what could be described as a "Neural Network-assisted Monte Carlo Tree Search". The MCTS has been a part of the course in Artificial Intelligence Fundamentals.

References

[1] Silver D., Hubert T., Schrittwieser J., Antonoglou I., Lai M., Guez A., Lanctot M., Sifre L., Kumaran D., Graepel T., Lillicrap T. P., Simonyan T., and HassabisD. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.