Exploring Self-play in Reinforcement Learning

Jesse Short
Arizona State University
Team 9
jcshort4@asu.edu

Zachary Giorno
Arizona State University
Team 9
zgiorno@asu.edu

Shubang Mukund
Arizona State University
Team 9
smukund9@asu.edu

Kaushal Rathi
Arizona State University
Team 9
krathi3@asu.edu

I. Abstract

Self play has recently developed lot of traction in the field of reinforcement learning. The term self play refers to a type of multi-agent learning (MAL) that uses replicas of itself to test an algorithm's compatibility in multiple stochastic and deterministic environments. In multi-agent reinforcement learning experiments, researchers try to optimize the performance of a learning agent on a given task, in cooperation or competition with one or more agents. These agents learn by trial-and-error, and researchers may choose to have the learning algorithm play the role of two or more of the different agents. It increases the amount of experience that can be used to improve the policy, by a factor of two or more, since the viewpoints of each of the different agents can be used for learning. Earlier human experts over the course of several decades used to manually create evaluation functions, sophisticated search techniques and domain specific adaptive algorithms to develop programs. Since the field of computer chess emerged in the 1970s and until AlphaZero's inception, the basic architecture of almost all chess engines has been virtually the same. On every move, a depth-first search (DFS) of the game-tree, up until a specified depth, is performed: that is, every single legal sequence of a chosen number of moves from the current position is computed. Since no legal moves are left out, this class of search algorithms is known as brute-force search. However, since then, the success of renowned algorithms such as alpha zero (Deepmind)[1], self play has increasingly gained attention for more extensive research. Hence in this project we chose alpha zero as the basis research and then explored the functioning of self play on different levels by performing exploratory tests on various deterministic environment games.

II. Introduction

The alpha zero program utilizes a combined policy and evaluator network. This means that there are 2 principal predictions that are made which are policy vector and value representation. This is similar to actor critic methodology except both have been combined into a single residual network. In addition to this a snapshot of the board state is also captured with the policy vector.

At first, a convolutional neural network the class of neural network that performs particularly well in visual classification tasks is initialised to return 2 randomly generated outputs: v(s), which will later estimate the win probability of the player from position s, and p(s), a vector which will later estimate how promising each of the legal moves from position s is. The network will later be trained to take in position s and predict v(s) and p(s) from it. Call this initialised neural network nnet0. The game-playing agent design that AlphaZero uses is based on a version of Monte Carlo Tree Search [1] which, instead of exploring every possible branch as deeply as the hardware allows like DFS does, explores a small number of branches for next 'n' time steps.

The basic premise behind MCTS is that it does not require exploring as many branches as DFS, and the extra computational power can be redirected elsewhere. In AlphaZero's case, the exact version of MCTS that it uses is as follows:

- On every move s, a number of simulations are run. Think of simulations as mini-games the agent plays against itself to see which moves result in the most favourable outcomes.
- Each simulation entails executing the following algorithm until the first position that has not been visited by any of the previous simulations is encountered.
- Out of all the legal moves in the current position, select the most productive one. The most productive move is evaluated as the one that strikes the perfect balance between exploration and exploitation: that is, a move that has a good enough track record to be worth exploring (exploitation), but on the other hand has not already been explored too much to yield useful information (exploration). More rigorously, the most productive move 'a' is the one that maximises the upper confidence bound.
- Play the selected move and run the algorithm again until either the end of the game is reached - in which case the evaluation of move 'a' for the simulation is recorded as 1 if the player wins the game and -1 if it does not, and the exploitation score (or Q value) Q(s,a) is recomputed or, as mentioned above, a new position 'r' that has not been visited by any of the previous simulations is encountered after the n time steps simulation. In that case, record the evaluation of r for the simulation as v(r) and continue these steps over until the end of the game [1].
- Once the pre-selected number of simulations (AlphaZero uses 1600) have been completed, we now have a distribution, called policy, of how many times each of the legal moves in position s has been visited. Recall that only the most promising moves were visited in every simulation,

so the policy is a direct reflection of how promising the agent has deemed each of the moves to be. Therefore, we can use it to select our next move in the game.

A newer derivative of AlphaZero, called MuZero, is not limited to only board games such as chess, but can also learn to play a range of simple video games from the Atari collection. Both AlphaZero and MuZero were designed by DeepMind, a subsidiary of Alphabet Inc., the parent company of Google.

Keeping this in mind we wanted to reproduce and explore this methodology in lot more detail. Hence in this project we began with training a lite version of alpha zero followed by exploring the policy vector and canonical board states in a more simplified game domain and then finally we explored the core of policy evaluation and extraction functions. These exploration tests helped us link all these bits of knowledge to understand the overall working of self play as a whole.

III. Methods

Our first approach to our project was to explore self play in chess. We started by researching DeepMind's *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* research paper on *AlphaZero* [1]. We found a github project that sought to reproduce *AlphaZero's* results just as we wanted to do and we decided to use it as a source for our project [2]. This source helped us reproduce the "self-play" technique used to train alpha-zero. The results from this experiment are found in the Implementation, Simulation and Results section.

Through our research, we realized we did not have the sufficient base knowledge in this field to completely reproduce the *AlphaZero* results. We therefore decided to pivot our study of self-play in chess to more simple games. Our conclusive goal of this project was to gain experience learning about self-play by training an agent in several different game examples. Each of these examples taught us an important aspect about self-play that has given us a deeper general understanding of reinforcement learning.

After our chess implementation, we began experimenting with reproducing a reinforcement learning algorithm that trains a popular game known as *Geometry Dash*. In *Geometry Dash*, an agent attempts to jump through an obstacle course at a constant speed. Our goal in studying this game was to give us a better understanding of how self-play works from the perspective of the MCTS simulation part mentioned in the paper but on a high level.

Further, we explored another self-play example in the form of TicTacToe. This example helped us gain understanding of how states and Q-values are stored in self play and how a policy extraction is commenced from that data.

Finally, we wanted to learn more about how the Q-values are generated/updated before policy extraction is conducted. To accomplish this, we experimented with training a simplified version of Pacman to help us gain this understanding and difference between Q-Learning and approximate Q-Learning.

Putting together all of the pieces of experience from these various games has helped us gain valuable knowledge in the field of reinforcement learning and the self-play technique.

IV. Implementation, simulation and results

Alpha zero chess :

```
game   1 time=970.5s halfmoves= 97 Winner.white
game   2 time= 26.9s halfmoves=112 Winner.black
game   3 time= 20.2s halfmoves= 84 Winner.black
game   4 time= 22.9s halfmoves= 96 Winner.black
game   5 time= 31.1s halfmoves=130 Winner.black
game   6 time= 44.6s halfmoves=181 Winner.white
game   7 time= 21.6s halfmoves= 90 Winner.black
game   8 time= 22.1s halfmoves= 89 Winner.white
game   9 time= 11.2s halfmoves= 48 Winner.black
game  10 time=  9.8s halfmoves= 42 Winner.black
```

Fig. 1. Snapshot of training chess model using self-play

In our chess experiment, we ran an algorithm that allows the agent to play against itself. For this experiment, we used an *AlphaZero* recreation that we found on github [2] as a basis. In this self-play training algorithm, the agent is duplicated into two. One agent plays as the white pieces while the other plays as the black pieces. When the algorithm detects the model has improved, the current model is then updated with the improved model for future episodes. The resulting output from this experiment is shown in Fig.1. The output shows how long each episode took, how long the game lasted in terms of half-moves (or turns) and which agent attained the win for a given number of training episodes. It should be noted that the first game as seen in Fig.1 did not actually take 970.5 seconds to run. This excessive time is the result of the program beginning the run timer while it is still setting up interactions with the GPU.

```
game   3 time= 43.8s halfmoves= 77 Winner.white by resign


. . r . k . . .
. p . . . p . .
p P . . p . p .
. b . p . . R .
. R . . . P . .
. . N . . . P p
. . . B . . . P
. . . . K . . .


The score is:  0.3464868831743593
```

Fig. 2. Snapshot of the end-of-game game board for an episode

Fig.2 shows the final game board arrangement for a particular episode of self-play training. The 8x8 chess board is represented in an ASCII format where the dots represent

spaces that are not currently occupied by any pieces. The lower-case letters represent the black pieces while the upper-case letters represent the white pieces. In this particular episode, it is shown that white won the game because black decided to resign. At this end point of the game, black is left in the game with only two key pieces (a bishop and a rook) while white remains with four key pieces (a bishop, a knight, and two rooks). It is interesting to note that black decided to resign despite not being in immediate danger of check or check-mate. It is likely the agent controlling the black pieces decided to resign based on knowing the future moves would not lead to winning the game. This behaviour is caused because the agent is trying to maximize the reward as much as possible even though it knows it will lose.

In this experiment, we have successfully implemented a self-play algorithm. Here, we learned the basics of how in self-play you can have an agent play against itself to gradually update its own model. It should be noted that the project repository we were learning from was strung together in many different sub-scripts and it was extremely difficult to decipher. As a result of this, we began to feel our learning from this repository became saturated and that it would be very difficult for us to further learn any more about algorithm used in this repository. We also realized just how complex of a reinforcement problem chess is and that if we wanted to get actual good results using chess as a basis, we would need a better base knowledge of reinforcement learning that we did not yet have. This is why we decided to move on to exploring the self-play technique in more simple games.

Geometry Dash :

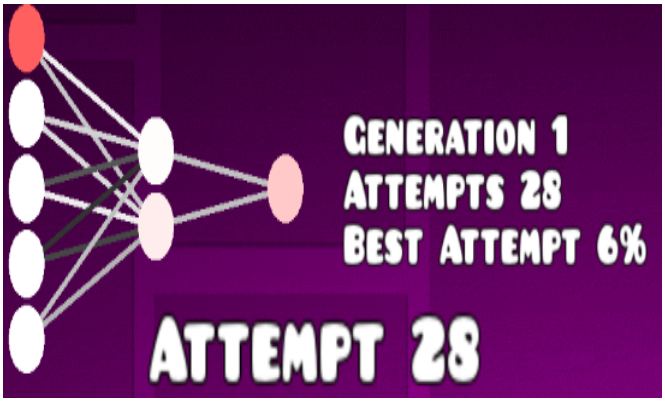

Fig. 3. Neural Network Figure for Training Geometry Dash

We explored geometry dash which is a game where a square has to go through and complete an obstacle course without dying. The main difference we discovered from geometry dash in comparison to chess for self play is that the agents data gets copied over to the new generation so that the new agents will have better timed jumps. The self play is player vs environment unlike chess where the self play is player vs player. As shown in Fig. 3, the inputs for the neural network are the distance to the obstacle, if the player is flying, player intersection with a jump orb, vertical obstacle detection, and the height of the nearest obstacle when flying. The only output in the game is whether to jump or not.

Fig. 4. Model learns as the green sprites die during the self play. Some green sprites start off jumping.

Each player has its own neural network and make up what's called a pair. The model is not pre-trained, at the start of the simulation, 10 green player sprites will start off jumping while 10 will not as shown in Fig.4. A value is obtained from the neural network known as the prediction which states whether the green player sprites should jump or not. If that value is greater than the jumping threshold (a number between 0 and 1) then the green player sprite will jump. After 1 generation, the best players data gets copied over to all green player sprites of the new generation with random modified values to further learn.

The simulation results obtained were 249 player deaths with 13 generations occurring. The agent was able to reach 100 percent. This can take anywhere between 5 and 20 minutes after several runs. The key process for what made the learning faster was iteratively replacing the best performing agent's data into the new agents.

TicTacToe :

So far into the project we had learned about self play on a high level and learned different configurations it could be used in such agent vs agent or agents vs environment. The experiment with Geometry Dash had exposed us to live training of the model. However we wanted to focus on the part in the paper of alpha zero where they explained about how simulation is ran for each strong until next 'n' steps and then backtracked from the highest rewarding state to the initial strong move. Hence to observe this we decided to move to a much simpler domain with finite states and actions which is TicTacToe game. We started with first training the model as seen in Fig.5 using self play. The repository that we used actually logs all the data generated during self play training. Hence once the model had been trained successfully (up until the point where a new model does not show any performance

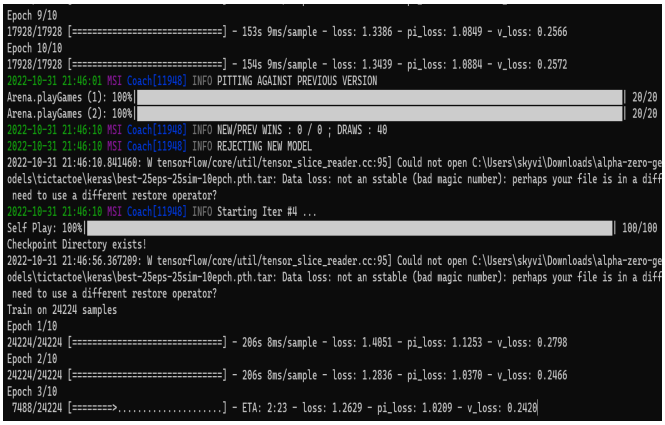

Fig. 5. Snapshot of training TicTacToe model using self play

improvement) we explored this data that was logged. We found that the model actually uses all this data for policy extraction. The data is referred as target example and format of the data is shown in Fig.6.

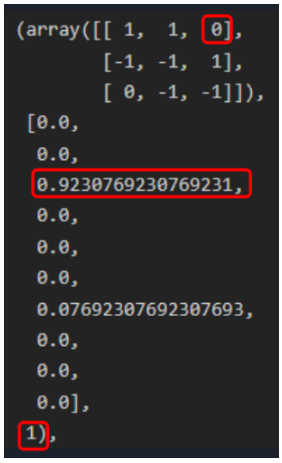

Fig. 6. Target example data generated from self play learning.

The target example has 3 key pieces of information. First the canonical state of the board, policy vector and the player whose most likely to win. The red box in the policy vector shows the highest probability of the next move that should be played which is 0.923 and corresponding to that the red box in canonical board shows where the move is to be played and lastly which player is most likely to win is marked in the bottom red box. This gave us insights related to how the simulation played out in alpha zero chess. They would create target examples similar to above and then play all the strong actions (high probability actions) from the policy vector and repeat this process for next 'n' steps and then backtrack from terminal states to these initial strong moves and play the move which gives highest value representation. Additionally, since TicTacToe is limited in state space this same piece of information is used for policy extraction during the testing phase. We also played against the model to check if the policy generated was accurate and it was.

Pacman: In TicTacToe we understood how the simulation worked and we realized that the policy vector has Q-values mentioned which correspond to which action needs to played. But then we wanted to focus on learning more about how these Q-values are generated and essentially how a policy vector is generated for extraction. We found another repository which allowed us to explore Q-learning and approximate Q-learning on a detailed level in the domain of Pacman. The most interesting part of this repository is that you need to write code snippets across multiple files for the Q-learning to work. Hence we wrote the code at different instances to complete the repository and then ran our tests and simulations. Another advantage is that this gave us lot of flexibility for testing. For instances we could create custom maps and switch maps to see the functioning and testing of the Q-learning.

Initially, we started with training the pacman in small size map/layout/grid. In this grid there are 2 food palettes,1 ghost and pacman and the goal of the pacman is to eat the 2 food palettes to finish and win the game. If the ghost killed the pacman first then the game would be lost. Hence we trained the pacman for 2000 episodes. During the training it was observed that the pacman was training using self play by playing against the environment. After training, during testing phase the pacman was able to win the game every time. It could correctly avoid the ghost and eat the food palettes every time. This can be seen below in Fig.7.

Fig. 7. Pacman after training on small size layout emerging victorious.

After this we switched the layout with a bigger medium sized layout and tried to reproduce the same result. This layout consisted of 4 food palettes,1 ghost and 1 small vertical dividing wall in the middle. Our expectations that the pacman

would be able to win actually did not come to fruition. We learned that this is because of the fact that Q-learning is not able to generalize well across multiple complex variables such as the randomness of the ghost actions, multiple food palettes and choosing the priorities among them and lastly the state space being so large in its entirety. This has been depicted in Fig.8.

Fig. 8. Pacman during testing phase on medium size layout dying every time.

This result made us realized the need and advantages for approximate Q-learning. Hence we decided to train the model using approximate Q-learning and self-play and observed the results on the same medium sized layout. We observed that this time Pacman was able to win every time against the ghost and eat all the food palettes successfully. This has been shown in Fig.9.

On further analysis we found that the features that have been used in this game are custom built. Some of them are manhattan distance between ghost and pacman, manahttan distance between food palletes and pacman, whether a collision is imminent with the ghost, whether a ghost is one step away, whether food will be eaten etc. We learnt that using features actually simplifies the state space extensively (which was the disadvantage in Q-learning) and this is the main reason as to why in approximate Q-learning Pacman emerges victorious. For example, in Q lewarning, pacman in 4 corners of the grid are represented by 4 different states however in approximate Q-learning the same can be represented by just using one state.

Another important learning is if instead of these custom features, they are replaced by a deep neural network since it is the standard de facto approximator and arguably the best, we essentially have Deep Q-Learning.

Fig. 9. Pacman during testing phase on medium size layout dying every time.

V. Discussion and Conclusion

In the pacman domain we discovered the basics and importance of Q-learning and approximate Q-learning. We also understood how essentially a deep Q-learning mechanism can be created. Hence if a state and action is given to this deep neural network then it should output Q-value. A series of these actions and states would give series of Q-values as output and essentially forms the policy vector that was observed in TicTacToe. In case of TicTacToe, during training since the state space is small these Q-values are updated simply by reaching towards the end states (at the game termination) get the rewards accordingly and then backtrack from these terminal states all the way to the initial start state of the game and update all the Q-values repectively.

However in case of chess because the state space is so large ($\sim 80^{10}$ states) there is not enough computation power and memory and hence during the training these Q-values are updated using the concept of simulation that was stated in alpha zero and could be also observed in Geometry dash where the model was being trained live. In simulation strong actions are played in an iterative manner over next 'n' time steps and then value prediction is utilized to backtrack from the terminal states of the 'n'th time step (not the end of the game) to the initial state from where the first few strong actions were played and ultimately decide one action that needs to be played and this is decided based on which action has the highest value prediction and this process is repeated until the end of the game. Finally, the delayed reward from the ending of the game helps to update all the Q-values of all actions and states from terminal till the initial ones and this entire process is repeated until the model is converged.

In conclusion, this project/experience helped us learn the basics of reinforcement learning and put all the pieces together

on all different levels and ultimately helped us understand the functioning and logic behind much more complicated algorithm of self-play. The observed results from self-play also made us understand the importance of self-play in multi-agent domains since it gives the agent capability to learn its own strategies by exploring, instead of choosing supervised learning which hampers the exploration considerably.

Considering all these learnings/lessons learned and some mistakes we made such as under estimating the hardware requirements, in future, we would now be able to estimate requirements, explore and understand more complex algorithms with a bit more ease and essentially strive to do better research in this field.

VI. Statement of Contribution

Name:- Kaushal Rathi
Contributions:-

- Researched Self-play (reading papers, watched videos and read articles to understand self-play), and also discovered some GitHub repositories and repository for Pacman.
- Trained alpha-zero chess algorithm once.
- Trained and generated the results of TicTacToe that have been mentioned in the report.
- Wrote the code for Pacman and implemented/trained the model and eventually generated all results that have been mentioned in the report.
- Contributed to all progress check Power point presentations.
- Wrote the 'IMPLEMENTATION, SIMULATION AND RESULTS' sections for TicTacToe and Pacman in the report. Also wrote the 'DISCUSSION AND CONCLUSIONS' section of the report.
- Also checked and revised the report for any grammatical/semantics and conceptual errors.

Signatures for peer approval:- Zachary Giorno, Shubang Mukund, Jesse Short

Name:- Jesse Short
Contributions:-

- Researched DeepMind's Self-play technique for AlphaZero.
- Researched various chess reinforcement learning algorithms.
- Spent a significant amount of time training the AlphaZero chess algorithm and analyzing the results.
- Helped discover sources for the Geometry Dash portion implementation.
- Contributed to all progress check power point presentations.
- Wrote the 'METHODS' section and chess portion of 'IMPLEMENTATION, SIMULATION AND RESULTS' in the report.

Signatures for peer approval:- Zachary Giorno, Shubang Mukund, Kaushal Rathi

Name: - Zachary Giorno
Contributions:-

- Researched Geometry Dash's Reinforcement Learning Self-play technique.
- Recorded Geometry Dash simulation results.
- Contributed and edited through all power point presentations.
- Discovered sources for the Geometry Dash portion implementation.
- Helped research chess implementation.
- Wrote the geometry dash portion of 'IMPLEMENTATION, SIMULATION AND RESULTS' in the report.
- Edited different sections of the report for errors.

Signatures for peer approval:- Jesse Short, Shubang Mukund, Kaushal Rathi

Name:- Shubang Mukund
Contributions:-

- Researched Alpha zero (reading papers, and articles)
- Researched different chess reinforcement learning algorithms and GitHub codes.
- Helped analyze the chess self-play algorithm.
- Helped analyze results of the other gaming self-play algorithms.
- Contributed and edited through all progress check Power point presentations and report.
- Wrote the 'ABSTRACT and INTRODUCTION' sections in the report.
- Edited report for any grammatical/semantics and conceptual errors.

Signatures for peer approval:- Jesse Short, Zachary Giorno, Kaushal Rathi

References

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

[2] https://github.com/Zeta36/chess-alpha-zero.

[3] https://github.com/OfficialCodeNoodles/AI-Geometry-Dash.

[4] https://github.com/suragnair/alpha-zero-general.

[5] https://inst.eecs.berkeley.edu/ cs188/sp11/projects/reinforcement/reinforcement.html.