Amoeba agent using multi-agent reinforcement learning

Amőba ágens több ágensű megerősítéses tanulással

Attila Juhos
IGCWW0

Dávid Bánóczi
W87ORP

Péter Tóth
HCNIQ7

Abstract

We present a neural network based agent that plays a version of the game Amoeba, also known as Tic-Tac-Toe. This version requires a five symbol long line to win, and is played on an arbitrarily large map. The learning methodology used takes inspiration from Google's AlphaGo and AlphaZero systems, which are the current bleeding edge of reinforcement learning. The agent learns from itself or any number of other teaching agents by repeating the cycle of playing games and learning from them, making it a multi-agent reinforcement learning process.

Abstract

Munkánkban egy, a sokak által ismert amőba játékot játszó, neurális hálózat alapú ágenset mutatunk be. A játék során egy tetszőlegesen nagy méretű négyzetrács celláiba helyezik el az ellenfelek a saját szimbólumaikat. Azon játékos nyer, aki a leghamarabb elhelyez egy öt hosszú, egyirányú szimbólumsorozatot. Az eszköz tanítási metodológiájának alapgondolatát a Google által fejlesztett AlphaGo, illetve AlphaZero rendszerek módszertanából kölcsönöztük. Ezek a rendszerek a megerősítéses tanulás területén a csúcskategóriába tartoznak. A tanítás úgy végezzük, hogy a tanítómintákat olyan játékokból nyerjük, amelyeket az ágensünk saját magával, vagy más, esetleg nagyon más elven működő ágenssel játszik. A multiágensű, epizodikus tanítás során a nagyszámú játék mintáit több epochon keresztül tanítjuk.

1 Introduction

Similarly to how AlexNet[1] shook the world in 2012 with its exceptional image recognition ability, in 2016 came AlphaGo[7] beating not only every previous AI, but also the best humans at the game of Go. At the time it was estimated it would take another 10 years for AI to catch up to humans, since given the vast action space, methods that decades earlier successfully tackled Chess and other games proved insufficient. AlphaGo still used recorded human games for initial training, and only afterwards did it play against itself. AlphaZero transcended it by learning completely from zero, hence it's name. This resulted in a a powerful general algorithm that can learn to play any board game to superhuman levels within hours of training. Our goal is exploring ways to build similar systems on a smaller scale. We try different approaches, see their limitations, and try to pinpoint the ways in which they fail, while trying to get closer to a competent agent.

1.1 The game of Amoeba

Amoeba or Tic-Tac-Toe is simple board game liked especially by students since it can be played using only a piece of checked paper and a pen. The rules are simple, the two players choose from two symbols X and O. The place place on symbol each, until one player manages to get a continuous five long line of their own symbol. This line can be either vertical, horizontal or diagonal. The game is simple and relatively easy to implement, yet has a large action space and a deep level of strategy is involved. This makes the game ideal since it does not take long to implement the game logic, but still provides a challenging task for an AI agent.

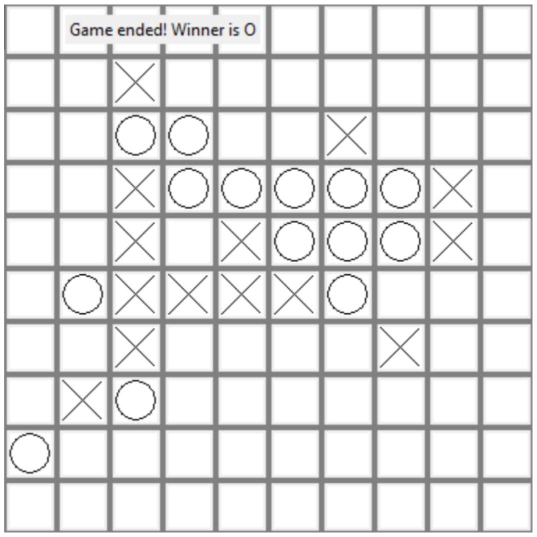
Game ended! Winner is O

Figure 1: Example of a finished game on the graphical interface created to make testing and evaluation easier

1.2 AlphaGo

Since AlphaGo was the main inspiration for the project, it is fitting to make a summary of its main features. AlphaGo and it's evolution AlphaZero[9] are systems created by Google to tackle the game of Go, but AlphaZero can be used for a wider range of games including poker.

The network of AlphaGo uses as a ResNet[10] type network using 20 identity blocks, and 256 filters within each convolutional layer. The network has two outputs, hence it's nickname "two headed monster". One output is the policy head, telling us how good the network thinks each move is. The other is the value head telling us how good a game state is, in other words how likely is the agent to win from there.

This network architecture is needed to implement a Monte-Carlo tree search[5] algorithm that explores the possible ways the game can go from the current state. It essentially thinks ahead, and checks how each action plays out. This MCTS algorithm is not new, but using a neural network to determine what moves to check (policy head) and also using a neural network to see how desirable the result of a move is (value head) is the innovation of AlphaGo.

As it is quite complex, we did not implement MCTS during the project, rather tried out how far other approaches can go.

2 System description

The system does not use any reinforcement learning framework because none of the ones checked supported multi agent training sufficiently. On one hand this led to having to code much more than in a more conventional supervised learning task, on the other it allowed to gain a deeper understanding of the methods used.

Learning is done by doing several episodes. An episode consist of the following steps:

1. The learning agent plays multiple games against one or multiple teaching agents
2. Extracting training data from the games played

3. Training the learning agent using this data
4. Evaluating the performance of the agent on a number of reference agents

Each step is described in greater detail below.

2.1 Playing games

Learning involves playing a large number of games. These games can be parallelized to allow for faster runtimes. For this reason the system uses GameGroups that makes agents play multiple games at the same time, allowing to send game states as batches to a GPU.

A game requires two participants. One is obviously the learning agent, the other is a teaching agent. Various teaching agents may be used:

Random agent This agent makes mostly random moves. However the moves may be restricted to the vicinity of already placed symbols. This agent is useful mostly to check if the learning agent suffers from overfitting problems

Handwritten agent This is a manually coded agent that plays on the level of a beginner/intermediate human. It can be used to teach the learning agent in a similar way to how human games were used to train AlphaGo. It is also a great benchmark to see how the learning agent preforms, since beating it requires skilled play

The learning agent When the learning agent plays against itself, it is called self play[8]. This is the main way AlphaGo, and the only way AlphaZero was trained. It is essential to be able to surpass other agents such as humans, which is the end goal.

The learning agent is a neural network, implemented using the keras framework.

The input is a one-hot encoded game board. The network does not know if it plays as X or O, rather it receives which symbols are its own, and which are the opponent's, this removes the need for it to distinguish between the symbols.

The output has the same dimensionality as the input, as it contains the move probabilities for each cell. This includes the cells already occupied by symbols, to make sure no illegal move is played, these are removed when selecting a move.

Move selection presents the exploration-exploitation dilemma[2]. If we select the highest probability move every time, it leads to monotone games that do no explore the possibilities. If we make a random move, it makes the exploration unfocused. To strike a middle ground, the network exploits with a given probability, otherwise it draws a random sample from the distribution given by the network.

The two main types of network structures used are the following.

2.1.1 Shallow Network

This is a relatively lightweight network that has a moderate number of layers. Its main advantage is that it can be run without high performance accelerators such as a GPU. This allows for quicker training sessions, allowing to us to iterate on the model, and try out new things more easily. The structure is this:

1. 9x9 convolution with 32 filters, ReLU activation
2. 3x3 convolution with 64 filters, ReLU activation
3. pooling layer with 2x2 window size and 2-2 strides
4. 3x3 convolution with 64 filters, ReLU activation
5. a dense layer with 256 neurons, ReLU activation
6. mapsize number of neurons with softmax activation

The loss function used is categorical crossentropy, selecting a move can be interpreted as a classification task, where the right move is the correct class. The optimizer is Stochastic Gradient Descent, as it seemed to perform better than more advanced optimizers such as Adam.

2.2 ResNet

As AlphaGo used the ResNet network architecture, it has proven to be very capable. It is built from identity blocks which contain two convolutional layers, but more importantly have a skip connection, that adds the output of the convolutions to the input of the block. This helps to avoid the vanishing gradient problem. AlphaGo used 20 identity blocks with 256 filters in every convolutional layer. This would be very large, therefore we used 8 identity blocks with 128 filters each. Weight regularization[6] is also introduced to help with overfitting[4].

This type of network did not provide immediate improvements over the shallow network, and its computationally intensive nature prevented quick hyperparameter optimization. For this reasons we experimented only a little with this network.

2.3 Extracting training data from games

One approach would be using every move from every game to train. A training sample may be the game state, and the desired output the move the winner made. The disadvantage of this is that we may end up teaching the network to make moves that did not actually contribute to winning, since in a long game the first few moves may be unrelated, especially when the players are not skilled.

To overcome this problem, we only use the last few moves of a game. Furthermore we weigh these samples according to how close they are to the winning move. The last move has a weight of 1 and every move after that is discounted by a multiplier. This makes the weight decay exponentially, after it reaches a low bound no further moves are considered, since they would have essentially zero weight.

We experimented with using the moves of the losing player for teaching too. This was done by giving the training sample a negative weight, discouraging the move. This did not prove effective, so it got dropped.

2.4 Learning

The training samples are used to train the learning network for a number of epochs. How many epochs to use was hard to determine, since it is not mentioned what settings they used when training AlphaGo. We settled on 5-15 epochs, but of course it is dependent on learning speed and the size of the training dataset.

2.5 Evaluation

The learning agent may be evaluated against any other agent described above. The metric of its performance is the win rate, which is determined by playing several matches between the agents. This win rate is slightly modified to account for draws, which are counted as half of a win, therefore if we two agents get a draw every time, the win rate would be 0.5.

It would be desirable to have an absolute metric for an agents performance too, since it provides a way to visualize the progression throughout a training session, even when the scores against the fixed references don't tell much (for example when the agent consistently beats every fixed reference). The Élő rating[3] is such an method. It is widely used in the rankings of many sports. The win rate of the learning agent against its previous version can be used to calculate this score.

We found that this score proves to be misleading and inaccurate, as it can result in agents having high scores while not performing any better. This may be due to a rock-paper-scissor effect. This effect can result in endlessly growing ratings without meaningful improvement. For example if an agent plays rock all the time, it beats scissors, leading to a high rating, but then the agent discovers to play paper, which leads to it winning against rock, then it rediscovers scissors and so on. Of course Amoeba is a different game but similar situations may arise.

This problem may be fixed by calculating the score against many previous versions, but how much it helps remains to be seen, as it was not implemented.

3 Conclusion

The main challenge in training was managing the quality of the training data. When trying to use self play to train, the network starts from a random configuration, this means the games played are random. This means that in most games only the last move helped with the win, the ones before it are typically random moves at the other end of the game board. When using this noisy data to teach, what we end up with is multiple misleading training samples to each helpful one, preventing convergence towards the correct plays. This can be somewhat counteracted by using a lot of discounting, making only the last move matter. But this prevents the network from learning how to get to that last winning move.

To counteract this problem we used teaching against the hand written agent initially without self play, making the quality of training samples much better.

3.1 Results

The main performance benchmark was the handwritten agent, that we used both for training and evaluation. The best performance we managed against it is around 27% winrate. This is a decent result since if we were to use supervised learning, it would not be able to surpass the performance of the teaching agent. However the system is still far from being able to continuously improve through self play to completely surpass other agents and humans.

3.2 Future work

The best way to improve the performance would be by implementing MCTS. It would help the network to make correct plays even when its suggestions are completely random, since it would try making those play and see whether or not they are actually good. This would probably result in having much more relevant training samples even at the very beginning of training. It would also improve the rate of improvement, since it would rely less on chance to find a winning strategy, as it is actively searching for them.

There is also a lot to be done at improving the framework. Better logging, finding good performance metrics to provide feedback about the progress, better visualization of training results.

References

[1] Md. Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Mahmudul Hasan, Brian C. Van Esesn, Abdul A. S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164, 2018.

[2] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[3] Mark E Glickman and Albyn C Jones. Rating the chess rating system. *CHANCE-BERLIN THEN NEW YORK-*, 12:21–28, 1999.

[4] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[5] Bilal Kartal, Pablo Hernandez-Leal, and Matthew E. Taylor. Action guidance with MCTS for deep reinforcement learning. *CoRR*, abs/1907.11703, 2019.

[6] Gabriel Pereyra, George Tucker, Jan Chorowski, Lukasz Kaiser, and Geoffrey E. Hinton. Regularizing neural networks by penalizing confident output distributions. *CoRR*, abs/1701.06548, 2017.

[7] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[8] 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.

[9] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[10] Sasha Targ, Diogo Almeida, and Kevin Lyman. Resnet in resnet: Generalizing residual architectures. *CoRR*, abs/1603.08029, 2016.