COMP 424 - Pentago-Swap

Reinforcement Learning Approach

Louis Amaudruz, ID : 260872326

1.Motivation and structure

Nowadays most of the state of the art game playing agents use what we call deep reinforcement learning which is a mix of reinforcement learning and deep learning. Such examples are for instance AlphaGo[1] from DeepMind, the first AI to beat a professional player in the game GO in 2015 which was considered nearly impossible task using algorithms such as those seen in class mainly due to its gigantic branching factor.

The Pentago-Swap is also a game where the branching factor can get very high, there can be up to around 200 possible moves near the beginning of a match which makes it hard to use algorithm such as MinMax or AlphaBeta pruning. In order to make them work, we would require a precise evaluation function which is naturally hard to learn due to the non-existent data. This led me to look towards Monte Carlo tree search or deep reinforcement learning and I ultimately chose the latter due to its known performances as seen above.

The idea in deep reinforcement learning is, without getting too much into the theory which we will do shortly, to let the agent learn using a neural network in an environment where he is given a reward for each move/decision he makes. This reward is generally a measure of how good the move he did with regards to his goal which in our case would be to win the game. The agent would then proceed to simulate a substantial amount of games where he will learn to maximize the overall rewards he receives per game and thus learn to play the game.

My work to implement this technique can be divided into two main parts :

1. Implementation of a neural network
2. The deep Q-learning algorithm

To be noted that my work uses a lot the code provided for the game and in some cases I even did some modifications to the provided code. For the full working code you can look at the my GitHub repository[2] (even clone it to test) which has been made public as the deadline is already passed.

[1] See https://deepmind.com/research/alphago for more details

[2] Repo : https://github.com/amaudruz/COMP424Pentago-Swap

Neural Network. The main part of the work has been done in the *NN2layer* class which is a simple neural network with one hidden layer and outputs a vector of predictions. The theoretical aspects will be discussed in the next section. It uses a lot of matrices and vectors for which I have coded multiple methods in the *MatrixUtil* class. For this project, the neural network takes a binary 1D array (0 or 1) as input which represents the state of the board and outputs a 1D array of double in which each element is a prediction of the expected sum of rewards until the end of a game for the specific move that correspond to index of the element. The conversion between *PentagoBoardState* and 1D array and between move and integer (position of the move in the array) has been done in the *PentagoStateRepr* class.

Deep Q-Learning. The algorithm itself is in the *reinforcementLearning* class where I create two agents with both a different neural network to model their predictions and I make them play against each other a certain amount of games. Each move they take are given either by the predictions of their respective model (with probability 1-epsilon) or using the alpha-beta algorithm with iterative deepening and hand written evaluation function (see *MinMaxABit* class). The probability of taking a move from the alpha-beta algorithm (epsilon in code) decreases as the number of games played increases so that the agent will gradually look more towards its strategy during training to perfect it. After each move, a reward is given and the model learns from this move.

When training is done, the weights of the model for one of the players are written to the text file *weights.txt* in the data folder. The student player then loads the data at the start of the game and then plays as the model predicts. The policy for taking a move as the model predicts is to simply take the **legal** action that was predicted to have the best overall reward.

2.Theoretical basis

In this section I will briefly look more in detail of the theoretical aspects of those techniques and link some of the sources I used. I expect the reader to know the basics as seen in class.

The neural network I use has one hidden layer with a certain amount of neurons in it and uses reLU activation function for this layer. Here would be the prediction Yh of an input X Given the weight matrices W1, W2, b1 and b2

$$Z1 = (W1 \times X) + b1,\ A1 = reLU(Z1),\ Yh = (W2 \times A1) + b2$$

Usually neural networks train on batches of data but due to not being able to use any external libraries it became quickly complicated so my neural network can only learn with one input/output pair at a time. The algorithm used to learn from an input/output pair is gradient descend using backpropagation as seen in class. For the loss function I had to use mean square error in order to use the neural network for deep Q-learning. Some good examples[3] can be found online of the implementation of simple 2-layer neural networks with backpropagation.

Implementing backpropagation from scratch can easily lead to errors so I approximated the gradient and checked the difference between the approximate gradient and the one I computed with backpropagation. Here is how I computed the approximate gradient : Let $L(\theta)$ be the loss function given weight θ, then my approximate derivative of loss given θ is

$$\frac{L(\theta + \varepsilon) - L(\theta - \varepsilon)}{2 \times \varepsilon}$$

With small epsilon this gives a precise approximation.

As for the deep Q-learning part, my ideas came mostly from the paper[4] from DeepMind about their game playing AI they trained with deep Q-learning to play Atari games. In this paper they explain how Q-Learning and deep Q-Learning works as seen in class with some modifications. There is even the pseudo code of the algorithm at page 5. The main difference between their algorithm and mine is I do not use experience replay due to the limitations of my neural network (not being able to learn on batches) and with probability epsilon I chose an action with regards to the alpha-beta algorithm and not randomly.

The reward I used is bigger to moves that continues lines of pieces of the player or blocks opponent lines of pieces (lines being horizontal, diagonal or vertical) and obviously way worse to moves that make the agent lose and way better to moves that make the agent win. I also give a bad reward as the game goes so the agent does not simply make the game last in order to get the biggest reward.

[3] See https://towardsdatascience.com/the-keys-of-deep-learning-in-100-lines-of-code-907398c76504 : Full implementation of a 2-layer NN in python. Main difference with mine is he only has a single output and I have no activation function for output layer.

[4] https://arxiv.org/pdf/1312.5602v1.pdf

3.Advantages/disadvantages, expected failure and weakness

With deep Q-learning there are no issues with regards to time for choosing a move during a game, you just have to use the model to make the predictions because all the thinking (training) has already been done before the game and the knowledge was given through the weights. This is a big advantage to other techniques such as MinMax or Monte Carlo where the computations are made during the game.

However In order for the agent to learn the game in a good way, the reward function must be extremely good in a sense that maximizing the reward means having only one goal which is winning otherwise the agent will not really learn to win the game. For this reason **the reward is the most crucial part** after having implemented everything else.

Sadly I did not have too much time after the implementation of the neural network and deep Q-Learning to perfect my reward function. If you try to play against my bot, you will clearly see some patterns in the way he plays and it is easy to win against him. He does win 95% of the time against a random player but I expect him to lose against alpha-beta or Monte Carlo with a good handwritten evaluation function. Despite all this, I am pretty confident that with a good reward function and enough training it could beat most the agents using the traditional approaches that have been shown in class.

4.Oder approaches

Going for deep reinforcement learning was clearly not the first idea I had. The first thing I tried was to implement a simple MinMax algorithm to play the game (see *MinMax* class). Using this method to play the game I quickly realized that it was not possible to use it as it took way to much time.

My second approach aimed to overcome the time problem using alpha-beta pruning with iterative deepening (see *MinMaxABit* class) with a simple evaluation function (1 for winning and -1 for losing otherwise 0). The iterative deepening part would take care of the time problem so that if the time is up, I would stop and use the result I computed from the previous iteration. The issue was that for this method to become efficient and to give a good chance of winning or to implement other time saving tweaks such as move ordering, I would need a precise evaluation function. This led to a dilemma, either I hand crafted it

with my knowledge of the game or I gave a model (linear for example) of some features I thought would be important and I would learn it, and I chose to try and learn it. The next problem I encountered was to find a way to learn my evaluation function, there are no data available so any machine learning, deep learning or simple linear regression were not possible. One way would have been to use genetic algorithms, start with a population of n agents with each a random evaluation function and I would make them play against each other and keep only some of the best to generate offsprings and so on. The issue with this technique was that each game takes close to 1 minute to fininsh, so making the population play against each other would be impracticable as it would take too much time.
This then led me to look into what has been done recently and I saw that reinforcement learning had been used in most of the state of the art game playing agents. But as I mentioned before I presume my agent will lose against a alpha-beta or Monte Carlo with good evaluation function.

5. Potential improvements

Deep Q-Learning has been shown to be the best technique there is for game playing AI due to its success **if done properly**. The thing is, there are a lot of improvements that I could not implement due to not having any external libraries at disposition. Not having those improvements might have resulted in my agent not learning fast or well enough.
For example experience replay in which at each move you not only learn on the move you just made but also randomly on some previous move you previously made and stored in memory which makes the neural network converge faster to the optimal solution.
There are a lot of documentation[5] online on potential improvement that can be made to the basic deep Q-learning algorithm that I implemented.
But most importantly, had I had more time I would have spent most of it to perfect my reward function because as I mentioned already several times **it is the most crucial part**.

[5] For example :
https://medium.freecodecamp.org/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682