Agenty McAgentFace

Project Report

Foundations of Intelligent & Learning Agents (CS 747)

Team Members

Anand Dhoot	130070009
Maulik Shah	13D100004
Samarth Mishra	130260018

This project required us to create an agent to play the game of carrom. Our agent was built on this open-source simulator, made in pygame + pymunk. The agent ran in two game modes -

- Single player: The agent aims to clear the entire board in as few moves as possible.
- Two player: The goal in this setting is to play the game against an adversary and aims to win the game under the standard rules of carrom.

We created several individual agents, both rule-based and by learning optimal actions using Reinforcement Learning techniques. Detailed descriptions of the implementation, analysis, observations and conclusions for each method are in the sections below.

Problem Details

The game board was modelled using a state space of size 800x800 pixels.

For every turn, the agent had to choose an action comprising of a triple - (position, angle, force). The agent is to choose a value between 0 and 1 for the position of the striker on the baseline and the force with which to hit the striker and a value between -45 to 225 for the angle at which the striker must be shot. However, the simulator adds a mean gaussian noise to each value in the triple.

Formally, the agent receives a reward of 1 when it pockets a coin and a reward of 3 on pocketing and covering the queen, if it does not result in a foul. The objective of the game, however, is to win the game by clearing the board in minimum number of turns.

Strategies for the one-player agent

Deep Deterministic Policy Gradients method

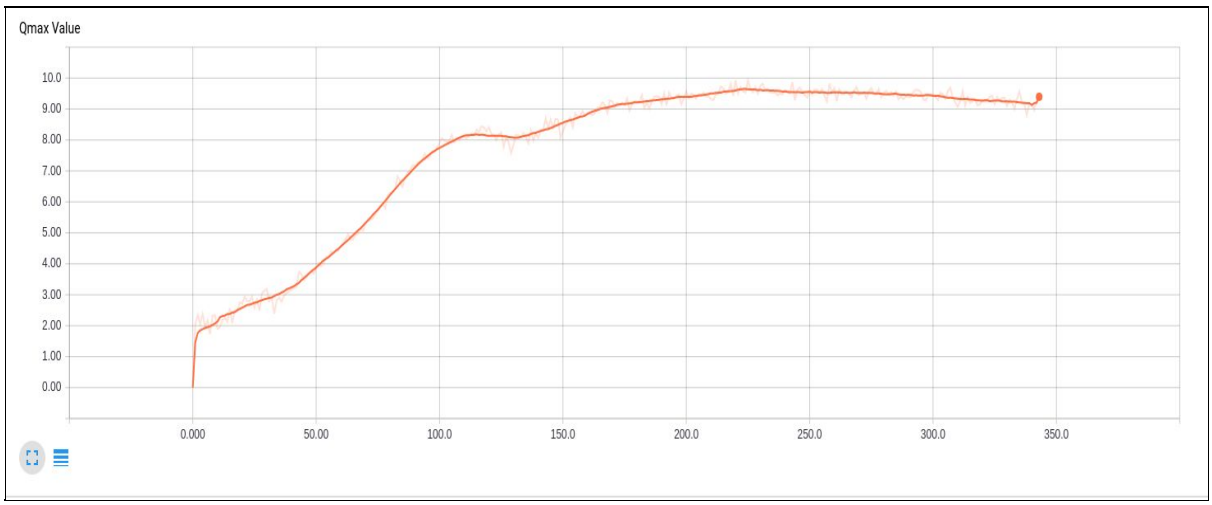
In [1], the authors have produced a policy gradient actor critic method which they have named Deep Deterministic Policy Gradients. Patrick Emami's blog [2] implements this method using tensorflow neural net library. The original implementation was built to play and train on games in the OpenAI Gym. We build on this implementation for our deep reinforcement learning task.

We encode the state as a 38x1 vector by reshaping the pairs of coin locations. The location for coins that have already been pocketed are set to (0,0).
Two deep neural networks have been used:

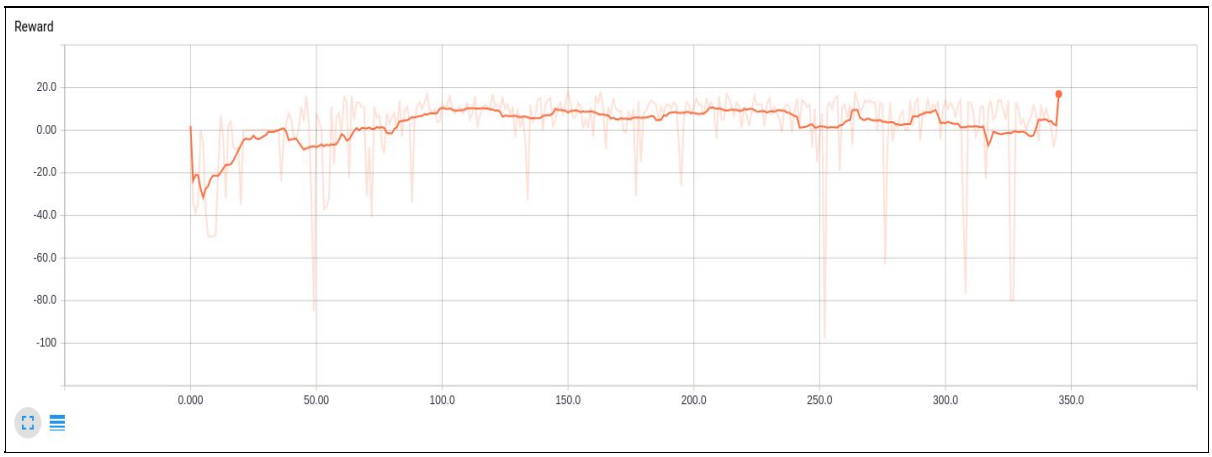
- An actor network, which outputs an action, given the current state. The action is a continuous value within its valid ranges. Hence, the output layer in our actor network has 3 nodes, one each for position, angle and force. The output layer outputs values between -1 and 1, which are then rescaled appropriately. The actions used by the agent also have an added exploration noise, which we reduce as the agent gets trained for more number of episodes.
- A critic network, which outputs the Q-value given the current state and the action given by the actor.

In each turn (a move of the player), a TD(temporal difference) error is generated on which the critic network is trained. The actor network weights are updated using policy gradients.

We also use experience replay to augment our training data. (State, action, reward, next state) tuples are stored in a replay memory and random mini-batches are sampled from it for training the networks. For training, we reshape the rewards a bit and give a reward of -1 to the agent if the striker is pocketed.



The above graph shows the mean Q-max value vs number of episodes trained for. Here, Q-max is the mean over all steps in an episode, of the maximum Q-values for a given state and all possible actions.



The above graph shows rewards collected by the agent for successive training episodes. Initially the rewards are negative because the agent tends to pocket the striker more often, without pocketing many coins. Later, it has learned to reduce the number of fouls it makes and hence, the reward values become positive.

We use a maximum episode size threshold of 500 steps. To speed up training, we also used some hacks like breaking out of an episode if the agent hasn't pocketed any coins for over 100 continuous time steps. After using this, each episode took much shorter time to finish.

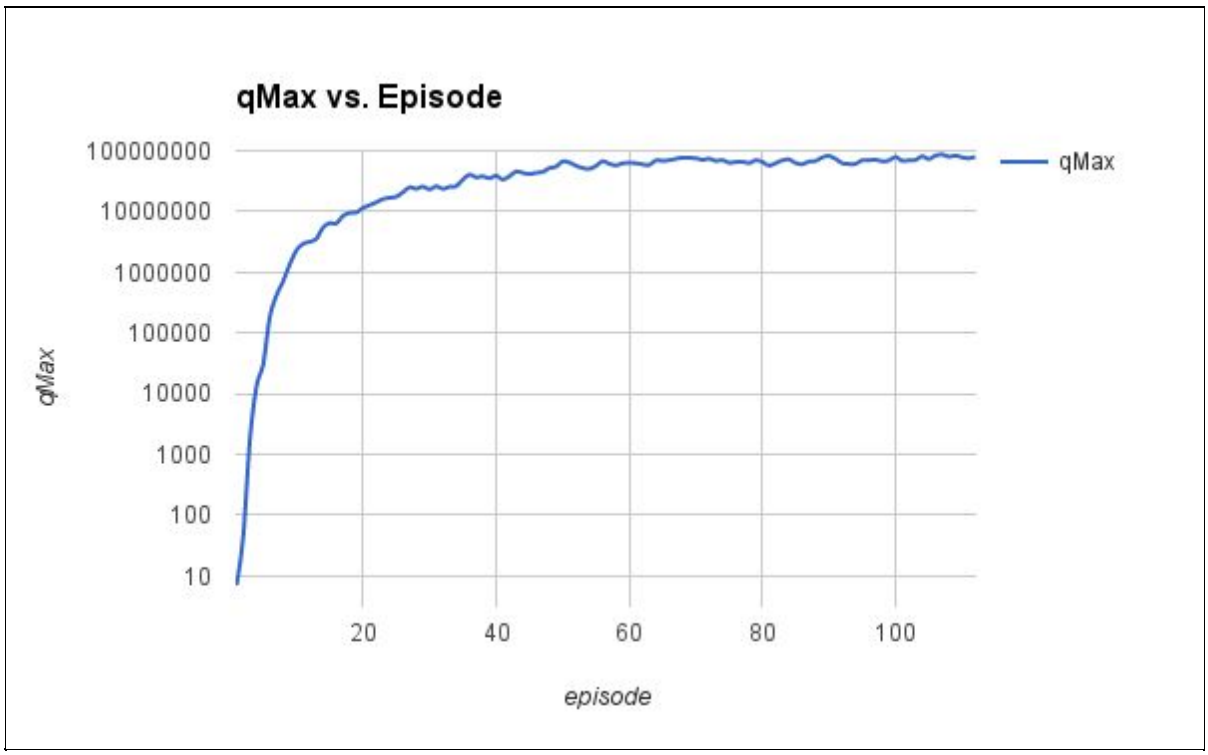
When the agent seemed to have been trained enough (the Q-max value saturated), its performance was pretty good in the initial stages of the game, but once the number of coins dropped below 3-4, it rarely could pocket any of them. On an average, the agent brought the number down to 5 coins in ~20 turns and about 3 coins in ~30 turns, but beyond this it is rarely able to pocket even 1 or 2 more coins in 100 subsequent moves. This problem was even worse before and the agent couldn't make points any further after a larger number of coins. We tried to tune the exploration noise added to the action, to overcome this, and the above was perhaps the best result we could achieve.

Q-learning

We implemented a Q-learning model using deep neural network based on Ben Lau's blog [3]. The game board was considered as a grid of size 16 X 16. The state is represented as an array of size 256 where the $(16*X+Y)^{th}$ entry denotes the number of coins in that block, X and Y being the coordinates in the 16 X 16 grid. The neural network takes in this state as input, and generates an output which is an array of 256 numbers, where the $(16*X+Y)^{th}$ entry represents the Q(s,a) value for that action. The action here is different from the action which the agent takes(x, angle,force).

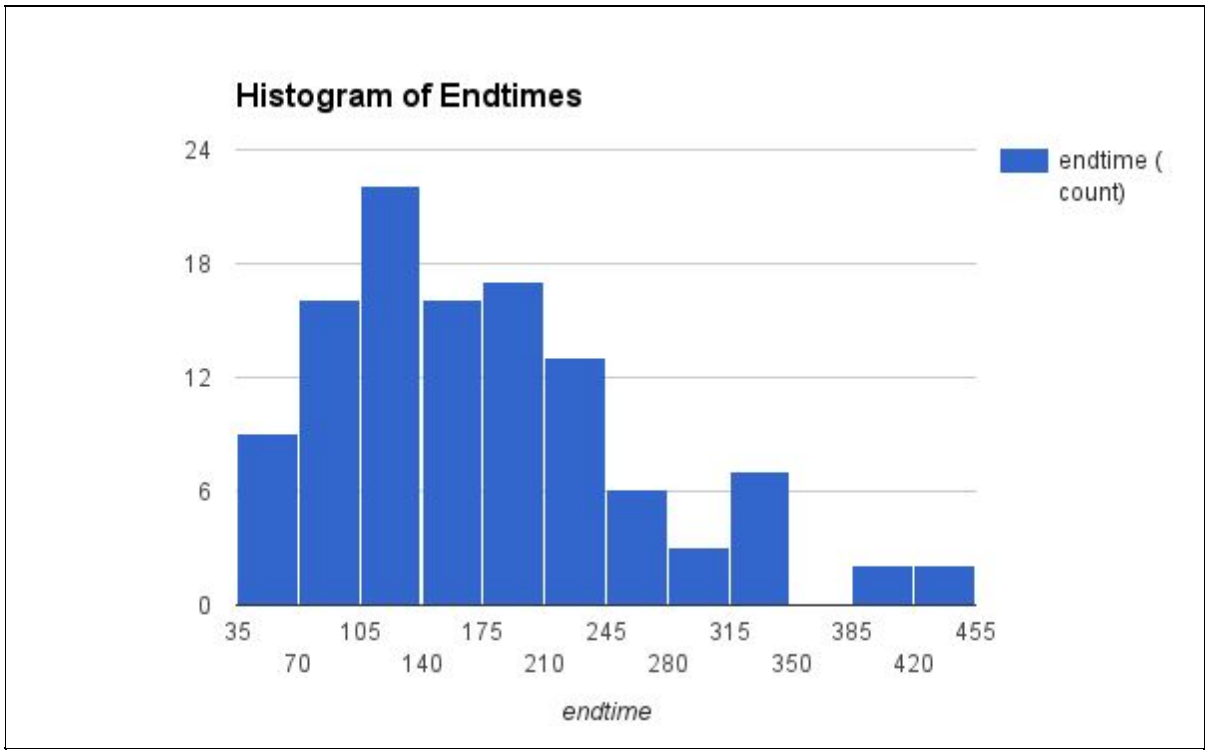
Here, the "action" corresponds to the block where the target should be set. In other words, we must target our striker to hit an imaginary coin at this point. Once we get the Q values corresponding to the 256 actions, we pick up the action which gives the maximum Q value. This argmax "action" determines the direction towards which we will be hitting. The (x,angle) values of the agent's action is then determined by placing the striker on the base_line in such a way that a coin placed at the action block goes to the nearest pocket. We always hit at the coin with force 1, since we observed over many(~100) sample test runs that using a higher force empirically cleared the board in fewer moves.

We trained the network using a reward of +1 for a coin, +3 for the queen, and -1 for a striker foul. With the introduction of negative reward for striker foul, we observed that the network converged faster in the qMax vs Episode graph compared to no negative reward. We also use experience replay in the same way as mentioned in DDPG algorithm. We trained the network for 120 episodes, and for ensuring that the network didn't get stuck within an episode for long, we bound the upper limit for the number of steps that could be taken in an episode to 500. We found the following results :

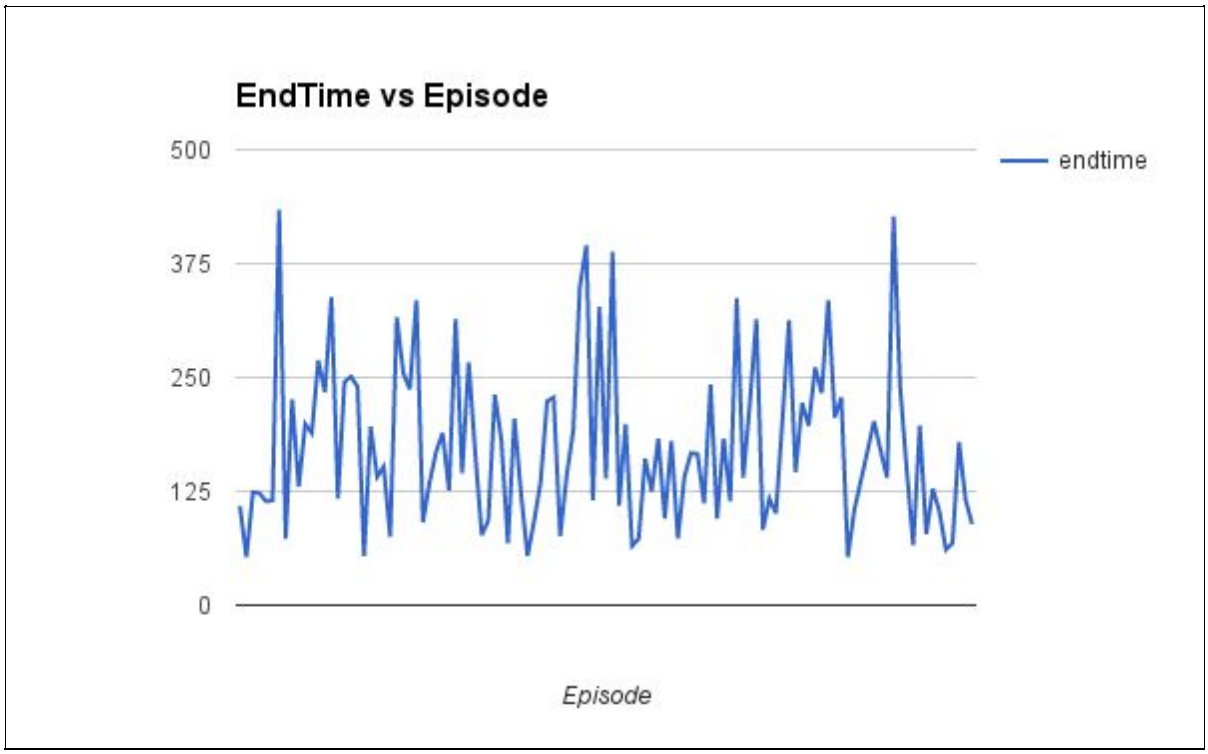


Here, qMax is the mean over all steps in an episode of the maximum Q(s,a) values generated for the state in that step. The graph shows that after about 30 episodes the action-value function converges.

The Endtimes for various episodes were distributed as follows:



As we can see, the values of endtimes ranges from 53 to 434, with most values at the low end. The median of the data is 161.



The above graph shows the running times of an episode(number of moves taken to clear the board) vs the Episode number. The network did not show any significant performance improvement for the overall runtimes. However, we did observe that after some episodes, the first 8-10 coins used to be pocketed within the first 10-20 moves, the next 5 within the first 50, and the last few (3-4) coins took anything from about a 100 moves to 200 moves.

Rule-based

This method involved hard-wiring actions of the bot, depending on specific scenarios. At a high-level, the agent played three types of actions -

- Opening move
- High Force
- High Precision

Opening Move

On simulating our agent for several games, we observed that the key to pocket all coins in a small number of moves is to play a ‘good’ first turn.

To find a good action, we explored the action space (the state is fixed for first move) over several actions using the one_step simulation, available along with the board simulator. The noise was turned off when finding the best first move for accuracy purposes.

For each candidate first move, we collected information about the total number of coins pocketed. Few of these actions pocketed as many as 10 coins apart from the queen! We chose one of these actions as our opening move.

High Force

This was played during the initial stages of the game, with an aim to sink as many coins as possible. The motivation for this strategy is that in the first few moves, most coins exist as small clusters. One approach that can be adopted in such scenarios is to nudge the cluster such that some coin becomes ‘free’ and then pocket it using a precise shot.

However, it was empirically observed that adopting such an approach resulted in large number of turns. Instead, our agent finds a cluster of coins on the board and shoots the striker towards one coin in the cluster with the maximum possible force. This ensures that all the coins get spaced out over the board (some even get pocketed!) resulting in ‘better’ board state for subsequent turns.

High Precision

Once we identify a coin that may be isolated (by virtue of very few coins being on the board, or otherwise), we aim to set the force and angle of our action so that the coin is shot into the pocket (head-on) in a single turn. To identify the target, we iterate over all possible candidates and choose one that aligns with a pocket, so that a direct hit would be able to pocket it. If no such coin is found, we choose a random coin.

This was done by varying the force linearly with respect to the distance of the striker position from the pocket it is aiming at, from a minimum to a maximum value, both of which are hard-coded (rather, was found after careful experimentation). These min-max values were set to lower values for ‘thumb shots’ (shots below the base line) .

Submitted one-player agent

After several experiments with the above one-player strategies and their combinations, we found that the rule-based strategy performed the best, and hence was submitted for the final evaluation. The following improvements were made to the rule-based bot above to optimize for total number of turns.

The Queen

The queen is a special coin, not just because it gives a reward of 3 points but because one needs to cover it to be able to get those three points. To increase the focus of the agent on the queen, we use an epsilon-greedy like algorithm in our high-precision step. With probability epsilon, we choose the queen as the target.

The value of epsilon itself is chosen depending on the number of coins already on board. We empirically observed that increasing the probability as the number of total coins on board decrease improves the performance of the agent. Currently, it is set to
0.5 + 1/(number of coins). This probability goes to 1 if only two coins remain.

Hitting coins touching the sides of the board

It was observed that if coins that are touching the sides of the board are hit head-on, there is very little possibility of being able to pocket them. Instead, pocketing the coins is much easier when they are hit on the side.

To incorporate this, we play the shot from one of the extremes of possible striker positions (chosen based on the location of the target coin on the board). This results in a side-shot rather than a head-on shot and lesser number of turns to finish the board.

Accounting for fouls

We observed that several of our High Force moves resulted in the striker being aimed directly towards a pocket. With the high force with which it was shot, the striker ended up getting pocketed, resulting in a foul. Therefore, in such cases, the angle of the action decided above was turned by a few degrees towards board frame closer to the coin.

Analysis of performance

We ran ~1000 experiments with this configuration of our agent and we could clear the board in an average of 24.023 turns. This is an improvement over the agent we submitted in Assignment 4, which cleared the board in 25.7 turns on average.

Submitted two-player agent

We extend our one-player agent for the two-player problem. We observed several games and found that using ‘High Force’ (described in the previous section) leads to pocketing of a large number of coins, in most cases involving some coins of the other color as well. We enhanced our agent in the following ways -

Null first action

Unlike the one-player scenario, hitting the striker with a large force towards the initial board state pockets some opponent coins, hence resulting in a foul. Instead, if we are playing as White (ie, player 1), we just play a null shot (try not to hit any coins) by playing the action (position = 0, angle = 0, force = 0). This also meant that after the opponent ‘break’, the coins spread out on the board giving our agent a large variety of shots to choose from.

Consider specific coins

The code of the one-player agent was suitably modified so as not to consider coins of the opposite colour when building our strategies.

Aim at isolated coins

Since, we use only precise shots to play, choosing targets which do not have any obstructions on its way to the pocket is very important. To do this, we find the number of coins in a cone of angle 10^{o} with its axis as the straight line joining the striker to the nearest pocket to the target coin. If there are any coins in the path, then we don’t include the coin in the list of candidate targets.

We ran several experiments using these settings and observed that we won very few matches against the agent submitted in Assignment 4 (suitably modified to hit coins of its own colour). Since, the performance of the player didn’t improve, we decided not to use this strategy in the final bot.

Analysis of performance

We ran some experiments by playing our submitted agent against the following bots -

- The random bot. Chooses a random action. We beat this agent 100% of the time
- Our assignment 4 submission. We beat this agent ~60% times

References

1. **Deterministic Policy Gradients**
2. **Patrick Emami's blog**
3. **Ben Lau's blog**