Deep Cue Learning:
A Reinforcement Learning Agent for Playing Pool

Peiyu Liao
Stanford University
pyliao@stanford.edu

Nick Landy
Stanford University
nlandy@stanford.edu

Noah Katz
Stanford University
nkatz3@staford.edu

Abstract

In this project, four different Reinforcement Learning (RL) methods are implemented on the game of pool, including Q-Table-based Q-Learning (Q-Table), Deep Q-Networks (DQN), and Asynchronous Advantage Actor-Critic (A3C) with continuous or discrete values. With two balls on the table, Q-Table performs the best in terms of average reward received per episode, but A3C with discrete action is found to be a more suitable method on this problem considering trade-offs between model performance, training time, and model size.

1 Introduction

Over the last few years, Deep Reinforcement Learning (DRL) techniques have seen great success in solving many sequential decision problems involving high-dimensional state-action spaces. Deep Learning (DL) techniques allow Reinforcement Learning (RL) agents to better correlate actions and delayed rewards by modeling the relationship between states, actions, and their long-term impact on rewards, leading to better agent generalization.

The goal of this project is to build a RL agent for playing the game of pool. It is an interesting topic to explore in that when considering a hit, we may not simply want to hit a ball into the pocket, but also want the next ball position to be convenient for future hits.

The problem is formulated as a Markov Decision Process (MDP) and solved with four different RL methods, including Q-Table-based Q-Learning (Q-Table), Deep Q-Networks (DQN), and Asynchronous Advantage Actor-Critic (A3C) with continuous or discrete values. These algorithms attempt to approximate the optimal values, Q-values, or policies for playing the game. At each time step, the current state of the game table is the input to the model, and the model outputs the best action to take to maximize future rewards.

One of the team members (Noah Katz) completed this project for CS229 and CS229A. RL was not taught in CS229A, however the applied use of neural networks and the skills needed to understand and debug issues with neural networks were covered in the coursework of 229A, and have been helpful in this project.

The code for the project can be found on GitHub [1].

2 Related Work

One of the most notable areas of research in RL is in building agents to play games. Games provide a simplified environment that agents can quickly interact with and train on. In the case of pool, plenty of work has been done in applying RL or other AI techniques to the game. Traditional AI techniques used include search [1] and heuristic [2] based methods. Some recent work also utilizes image data and DL [3]. Both types of techniques worked well in their respective environments, however they both have their issues. Search or heuristic based methods require the agent to have full, deterministic understanding of its environment. And while image based methods can generalize well, they require large amounts of computation to train. We decided to look for a compromise between these two methods by developing a method that had lower computational needs but could still generalize well on unseen environments without human-tuned heuristics.

[1] https://github.com/nkatz565/CS229-pool

DL methods have recently been especially successful in solving video game environments. DRL requires many episodes of training compared to other types of RL methods, but it has seen superior generalization ability and is thus capable of achieving human-level performance on many game tasks. Many groups have worked on applying DL methods to RL to improve agent generalization. In 2015, Google DeepMind showed that Deep Q-Networks (DQN) could be used to solve the Atari game at scale [4]. Later in 2016, DeepMind proposed a number of new asynchronous DRL algorithms, including Asynchronous Advantage Actor-Critic (A3C), which DeepMind argued was the most general and successful RL algorithm to date due to its ability to learn general strategies for complicated tasks purely through visual input [5]. Their success with DRL largely inspired our work.

3 Problem Formulation and Environment

3.1 Problem Formulation

The problem is formulated as an MDP with the following set of state, action, and reward definitions:

$$\begin{aligned} s &= [x_1, y_1, \ldots, x_m, y_m] \\ a &= (\theta, F) \\ R(s, a, s') &= -1\{s_{2:m} = s'_{2:m}\} + \alpha(\text{numBalls}(s) - \text{numBalls}(s')) \end{aligned}$$

where m is the number of balls, x_1 and y_1 are the x and y positions of the white ball, x_i and y_i are those of the i-th ball, $\theta \in \mathbb{R}$ is the angle of the cue within range $[0, 1]$, $F \in \mathbb{R}$ is the force applied to the cue within range $[0, 1]$, α is the relative reward weight between hitting no ball and pocketing one ball, and numBalls(s) returns the number of balls still in play at state s. $s_{2:m}$ is the list of elements in s other than the first element, i.e. the positions of all balls other than the white ball. In other words, negative reward is assigned if no balls are hit, zero reward is assigned if the white ball makes contact but does not produce a pocketed ball, and a positive reward is assigned if some balls are pocketed. In this project, α is set to 5.

Normalization is applied to state and reward in the deep methods to be introduced below, i.e. DQN and A3C, to stabilize training.

3.2 Environment

The game simulation engine is modified from an open-source Pool game implemented in Python [2]. The following modifications are made to fit this project:

1. Created an interface to modify game parameters.
2. Created an interface for the RL algorithm to interact with.
3. Set game graphics up so that they can be turned off for faster training, or turned on for visualization of the training process.
4. Optimized the game engine by removing unnecessary functions.

There is only one player (our software agent) in this game. In addition, instead of applying the complete pool game rule, the experiments are conducted in a simplified setting with a small number of balls, and the goal is simply to pocket the balls disregarding the order. A two-ball scenario would prove that the model can learn how to pocket a ball, or how to setup a subsequent shot so that it can pocket a ball later. The four-ball scenario would prove that the model can have some extra understanding of how the balls interact with additional balls that may be in the way.

In the learning, an episode is a game with a set maximum number of trials. Each episode is automatically concluded when all balls have been pocketed, or the maximum number of trials have been reached.

4 Methods

In order to solve the MDP, three algorithms are implemented, including Q-Table, Deep Q-Networks (DQN), and Asynchronous Advantage Actor-Critic (A3C). For A3C, it is implemented with both continuous action and discrete action.

4.1 Q-Table-based Q-Learning (Q-Table)

Q-learning is an algorithm that learns a policy which maximizes the expected value of total reward in the future. To learn the Q-value, which is the expected total future reward by taking a certain action at a certain state, we iteratively update the Q-value with the following equation upon each experience:

$$\hat{Q}(s, a) := \hat{Q}(s, a) + \alpha(r + \gamma \max_{a' \in \text{actions}(s')} \hat{Q}(s', a') - \hat{Q}(s, a)) \tag{1}$$

[2] https://github.com/max-kov/pool

where $\hat{Q}(s, a)$ is the current estimate of the Q-value, s is the current state, a is the current action, r is the reward received by taking action a at state s, s' is the transitioned next state, α is the learning rate, and γ is the discount factor of the rewards.

Q-Table implements Q-learning using a look-up table to keep the Q-value for each discrete state-action pair.

We use the epsilon greedy method as our exploration strategy, where at each time step, there is a probability ϵ of selecting a random action and probability 1-ϵ of selecting the optimal action from the current estimate of the optimal Q-Function.

4.2 Deep Q-Networks (DQN)

Q-Tables work well on small number of states and actions, but for continuous states and actions that need to be discretized, much information is lost and the learning becomes inefficient. In DQN, a Neural Network is used to approximate the Q-function by taking the state values as input and predicting the Q-value for each potential action. The parameters of the Q-network are then updated using optimization algorithms such as stochastic gradient descent and backpropagation.

In the DQN implementation, a Neural Network with 2 hidden layers is used with continuous state values as input and a discrete action as output. The dimension of the 2 hidden layers are 64 and 256 each. The output layer yields the Q-values for each action at the input state, which are then fed through a Softmax function to create probability distribution for taking each discrete action choice. Actions are then sampled from this probability distribution.

A replay buffer is used to hold all experiences obtained in an episode, which are then shuffled for training. This helps to reduce correlation between experiences, improve convergence, and improve data efficiency by allowing us to reuse data for training.

Aside from the original network used for training, a second network called target network is used as the estimate of the optimal value function. The target network is updated every several iterations as the weighted sum of the parameters of the two networks. With the use of a target network, the Q-values will not be chasing a moving target.

The network parameters are trained with the following equations:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla_{\mathbf{w}} L \tag{2}$$

$$L = \sum_{i=1}^{batch\_size} (f(s_i) - f_{tar}(s_i))^2 \tag{3}$$

where $\mathbf{w}$ is the parameter of the model, α is the learning rate, L is the MSE loss, s_i is the state of the i-th experience in the batch, f is the output of the original network, and f_{tar} is the output of the target network.

4.3 Asynchronous Advantage Actor-Critic (A3C)

A3C consists of a global network that approximates both the value function and the policy, and several workers that interacts with the environment asynchronously to gain independent experiences and send them to the global network for a global update every few actions.

In this algorithm, the benefits of both policy iteration and value iteration are combined, and the policy can be updated more intelligently with the value estimate. In addition, multiple agents learning asynchronously on different threads speeds up the overall training.

Continuous Action Continuous action values are chosen by sampling from a normal distribution with the mean and variance predicted from the network. The variance itself serves as the exploration factor.

Discrete Action The same as in DQN, discrete actions are chosen based on the Q-values predicted for each action.

5 Experimental Results and Discussion

5.1 Experiments

Four algorithms, Q-Table, DQN, A3C with continuous action, and A3C with discrete action are first evaluated in the simplest environment with two balls, i.e. one white ball hitting another ball. Each algorithm is trained for 1000 episodes, each episode allowing a maximum of 25 hits. The trained models are then run through an evaluation over 100 episodes with exploration strategies turned off, and the model performance is represented by the average reward per episode.

Figure 1: Average rewards over 1000 training episodes for Q-Table, DQN, A3C with continuous action, A3C with discrete action, and random policy in a two-ball environment.

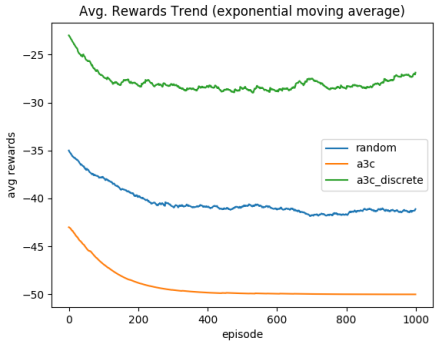

Figure 2: Average rewards over 1000 training episodes for A3C with continuous action and discrete action in a four-ball environment.

The two A3C algorithms are then trained in an environment with four balls to evaluate their generalization ability to a larger state space. While other settings remain the same, the maximum hits allowed are increased to 50.

In Q-Table, the state is discretized into 50 buckets for both x and y positions, the angle into 18 buckets, and the force into 5 buckets. In both DQN and A3C with discrete actions, the angle is discretized into 360 buckets, while the maximum force is always chosen.

To interpret the numerical values of the rewards, note that the minimum reward is $(-1 \cdot \text{max\_hits})$ per episode for not hitting any ball at all during the episode, and the maximum is $(5 \cdot \text{num\_balls})$ for pocketing all balls.

A random policy is also evaluated and serves as the baseline for other algorithms.

All experiments are conducted on a machine with 16 GB of memory and an 8 core 6th Gen Intel i7 processor running at 2.60 GHz.

5.2 Results

Two-Ball Environment The moving average rewards received over the training period of all five algorithms is shown in Figure 1. The evaluation results, training time, and model size information are provided in Table 1.

Methods	Average Reward	Training Time	Model Size
Q-Table	**-6.4**	136 min	1.12 GB
DQN	-21.3	27 min	162 KB
A3C (continuous action)	-19.44	**13 min**	**8 KB**
A3C (discrete action)	-18.46	17 min	149 KB
Random	-22.8	-	-

Table 1: Evaluation results over 100 episodes, training time, and model size information in a two-ball environment.

It is seen that Q-Table outperforms others in both the training and evaluation results. Q-Table has learned the exact steps to hit in the ball from the starting position, hence the good performance. However, both its training time and model size are significantly larger than others and scale up as the state space increases, thus this method is limited to two-ball environment.

For the deep methods, the training performance is similar for DQN and A3C with discrete actions, but the two A3C methods has achieved a better performance than DQN in evaluation. All three methods have efficient training time and model size, in particular A3C with continuous action.

In DQN, the model appears to do marginally better than applying a random policy. When trained for fewer episodes (approx. 250), the model learns only 1 or 2 moves that tend to get better total rewards, and it tends to perform them again and again. The model might improve more if given the opportunity to explore over more episodes. This examination was not conducted due to time and memory constraints, and it is left as future work.

For A3C with continuous action, the performance during training is initially degrading, but after around 700 episodes, it has possibly escaped the poor local minimum and the performance starts to increase rapidly. Overall, it has a disadvantage of

unstable training, and this is probably due to the fact that it predicts the mean and variance of the normal distributions for each action, and it is difficult for the sampled values to settle in the bounded value range.

A3C with discrete action has a better and more stable performance compared to A3C with continuous action, sacrificing only a small amount of training time and model size. The difference between the two outcomes is probably because classification training is more effective than predicting bounded continuous values.

Four-Ball Environment The training results are shown in Figure 2. The evaluation results and other statistics are provided in Table 2.

Methods	Average Reward	Training Time	Model Size
A3C (continuous action)	-50.0	**28 min**	**11 KB**
A3C (discrete action)	**-24.86**	52 min	152 KB
Random	-41.024	-	-

Table 2: Evaluation results over 100 episodes, training time, and model size information in a four-ball environment.

The average reward in both algorithms are not increasing, indicating that both of them are not learning effectively. Compared to a random policy, A3C with discrete action outperforms the random policy in terms of training and evaluation performance, while A3C with continuous action has leaned towards choosing actions that are not hitting the balls at all.

With a deeper look into the actual actions taken, it is observed that for A3C with continuous action, the action values are often clipped at maximum 1 or minimum 0, which implies that the predicted values tend to explode or vanish within the network. For A3C with discrete action, only a few certain actions are chosen most of the time at the later stage of training. Further investigation on the reasons causing these behaviours in the models in the four-ball environment needs to be conducted in the future.

5.3 Analysis

Overall, A3C with discrete action is considered the more ideal choice for this problem considering all trade-offs. It is scalable with state space, the training is stable and efficient, and the performance is acceptable. However, in an environment with simpler settings and with potentially unlimited resources, Q-Table has the advantage of being the simplest implementation and having the best performance.

Q-Table has produced a particularly interesting result in that at the end of its learning, it repeatedly executed an exact set of moves that will complete an episode in 6 moves for a total reward of 4. This is an acceptable solution given the problem, but it would be better if it learned how to pocket the ball in one hit. This might be solved with a lower gamma value to discount the non immediate rewards more harshly, or by more random exploration.

From the experiments, several additional observations have been made:

1. Sparse rewards may affect training efficiency. In the design of reward model, positive rewards are only given when a ball is pocketed, which is difficult to achieve at the beginning of training. More timesteps are required for the model to learn a good action, which makes the training inefficient.
2. Normalization is essential in stabilizing the training in neural networks. Without normalization to the inputs, it is found that the values tend to explode as the inputs are forwarded to the end of the network, and it became difficult for the output values to be tuned back to its normal range, hence the output actions are mostly clipped at 1 or 0.

6 Conclusion and Future Work

The game of pool has been formulated into a MDP and solved with four different algorithms, Q-Table, DQN, A3C with continuous action, and A3C with discrete action. All four algorithms successfully outperformed the baseline performance in a two-ball environment.

Q-Table is found to be effective in achieving the best performance despite its simplicity, but the significant training time and model size has prevented it from being applied to an environment with larger state space. Taking into account the trade-offs, A3C with discrete action is possibly the most suitable algorithm among the four for this problem.

In the future, the poor scalability of the models in an environment with more balls should be addressed first. The game can further be made more challenging by enlarging the table size, adding rules, etc. To evaluate the true ability of the models, they should be compared with human performance. Finally, the model can be integrated with hardware as a pool robot for entertainment and educational purpose.

7 Contributions

- Noah Katz modified the game simulator so that it could be interfaced with our algorithms, set up the environment for running the training and evaluation, and handled running the tests needed to gather results. He also implemented the worker class in the A3C method.
- Peiyu Liao created the environment wrapper for MDP and implemented the Q-Table method and A3C framework with its global network class. She also refined and analyzed the A3C algorithms and created the discrete version of it.
- Nick Landy worked on implementing and refining DQN. He also conducted all the experiments and analysis related to the model. He also worked on optimizing the simulator to improve model training speed. He is also the major writer of the report.
- All team members worked together on the report.

References

[1] C. Archibald et al., Analysis of a Winning Computational Billiards Player, In Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09), 2009

[2] J. Landry et al., A heuristic-based planner and improved controller for a two-layered approach for the game of billiards, In IEEE Transactions on Computational Intelligence and AI in games, IEEE Computational Intelligence Society, 2013

[3] K. Fragkiadaki et al., Learning visual predictive models of physics for playing billiards, In ICLR 2016.

[4] Mnih, V. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).

[5] Mnih, V. et al. Asynchronous methods for deep reinforcement learning. In Proc. 33rd Int. Conf. Mach. Learn. Vol. 48 (eds Balcan, M. F. Weinberger, K. Q.) 1928–1937 (2016)