CS 440/ECE448

Assignment 4:

Reinforcement Learning and Deep Learning

May, 2018

YUHAO CHEN,
ZIRUI FAN,
JUNHAO PAN

Credit Hour: 3

1 Introduction

1.1 Overview

In this assignment, we have chosen python as our language to implement the functionalities. We majorly use Jupyter notebook as the developing environment.

The assignment includes two parts: Q-Learning and Behavioral Cloning and Deep Learning. The common goal for those two learning methods is to train an agent which plays the game Pong. The setting of the game is fairly simple, in which a ball moves in the space, and a paddle tries to make the ball bounce as many times as possible.

In part 1.1, we have implemented Q-Learning and SARSA agents, which are quite similar, to play the game. In part 2.1, we have developed a three layer neural network which classifies the states into actions.

1.2 Files

./Q-Learning.ipynb	jupyter nootbook for part 1
./Cloning.ipynb	jupyter nootbook for part 2
./Playground.ipynb	jupyter nootbook to play around
./Pong.py	supporting python file for part 1
./Cloning.py	supporting python file for part 2
./src/*	pictures and misc.

2 Q-Learning

2.1 Single-Player Pong

2.1.1 Implementation

Our goal is to design a Q-Learning agent to play the game Pong. The success is rated by the number of re-bounces of the ball on the paddle controlled by the agent.

In this part, we have designed a class Pong to represent the settings of the game. This class contains methods which describes the the move of the ball and the paddle and virtually play the game. In addition, the class Pong contains two sub-classes, Agent and State.

The Agent class is responsible for deciding the actions of the paddle, specifically the decision to move up or down, or stay still. It holds an array of the action-utility values, which stores information about frequency and utility of a state. It is trained with the standard MDP + Q-Learning/SARSA algorithm. The State class represents the state of the game

and feeds the information to the agent for decisions. Some important variables which it holds are ball_x, ball_y, velocity_x, velocity_y, and paddle_y.

2.1.2 Parameters

As described in the documents, the learning rate $\alpha = c/(c+N)$ in which c is a constant to be chosen on our own and N is the frequency of the previous appearances of this specific state. γ is the discount factor, and Ne is the threshold for exploration. If γ is set high, the agent looks at a more long-term reward, which affects the convergence. If γ is low, the agent only considers the current reward.

We have tried different parameter combinations. After narrowing it down, we believe that $c = 10, \gamma = 0.9, Ne = 100$ is one of the better choices. Please see the following output from our tests which highlights the selection of parameters:

```
10 0.8 150
not input file: ./training_result/training_c10_gamma0.8_ne150_file.txt
average bounces per round 12.101
stop training
time spent:  113
10 0.8 200
not input file: ./training_result/training_c10_gamma0.8_ne200_file.txt
average bounces per round 11.488
stop training
time spent:  94
10 0.8 250
not input file: ./training_result/training_c10_gamma0.8_ne250_file.txt
average bounces per round 11.103
stop training
time spent:  85
10 0.9 50
not input file: ./training_result/training_c10_gamma0.9_ne50_file.txt
average bounces per round 11.503
stop training
time spent:  312
10 0.9 100
not input file: ./training_result/training_c10_gamma0.9_ne100_file.txt
average bounces per round 16.482
stop training
time spent:  232
10 0.9 150
not input file: ./training_result/training_c10_gamma0.9_ne150_file.txt
average bounces per round 11.092
stop training
time spent:  110
10 0.9 200
not input file: ./training_result/training_c10_gamma0.9_ne200_file.txt
average bounces per round 13.517
stop training
time spent:  98
10 0.9 250
not input file: ./training_result/training_c10_gamma0.9_ne250_file.txt
average bounces per round 13.376
stop training
time spent:  86
10 1.0 50
not input file: ./training_result/training_c10_gamma1.0_ne50_file.txt
```

As suggested by the picture, this combination of parameter is able to bounce the ball 16 times on average. The other combinations which are numerically close also achieves good results, which suggests that parameters in this range can produce fair policies.

2.1.3 Discussions and Results

We train each of the Q-Learning and SARSA agents for 150k rounds, which is a reasonable amount of rounds to play to make sure the policies converge. The difference between a Q-Learning agent and an SARSA is minor. Instead of updating the Temporal Difference with *argmax*, SARSA updates with the Q-value of the action which is actually taken by the agent. The minor difference is not enough to distinguish one from another especially after many rounds. The graphs can confirm this speculation.

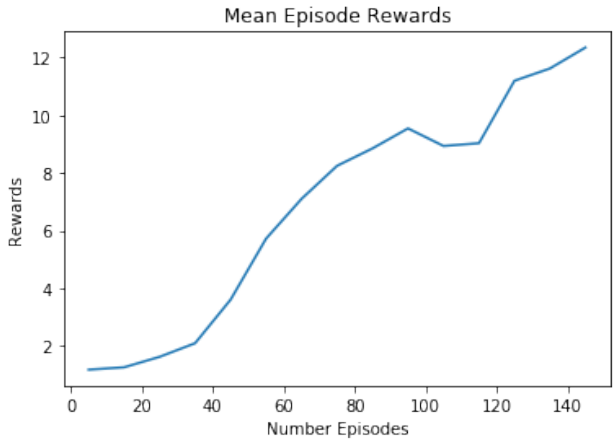

Figure 1: Rewards vs. Episodes Q-Learning

Figure 2: Rewards vs. Episodes SARSA

As we are testing the environments, it appears to us that the change of board size from $(12, 12)$ to other settings can affect the training.

For example, as we use a smaller board size $(6, 6)$, the result are quite awful when we use the same parameters and play for 100k rounds. We are suspecting that a 6×6 board is not big enough to give adequate state for the agent to analyze; thus, the agent is not able to develop a reliable model for the game.

We have also tried larger boards. However, it seems that often it takes more than 100k plays to converge. When we let it play for 200k rounds, it starts to look better and gives good results. Nevertheless, we think it is not quite necessary to spend much more time and resource to gain very limited marginal improvements.

Therefore, it is safe to conclude that as the board get small, the agent finds it more difficult to develop a model to suit the game. When the board gets larger, there is a trade-off between the accuracy and the training resource to gain marginal improvements.

2.1.4 Extra Credits

We have successfully implement a graphic user interface to represent the states with pygame. There is a video of the agent playing with itself which I have included in the package.

We have also added the ability for a human to play as well. There is only need to add event listeners to the agent game. However, due to the quality of the graphics, we are not able to defeat the agent easily. In fact, we lose the most of the times; therefore, we will not include a video of that.

The strength of the agent, from our playing experience with it, is that it is quite consistent. Even though consistency brings predictability, in this game, consistency is important.

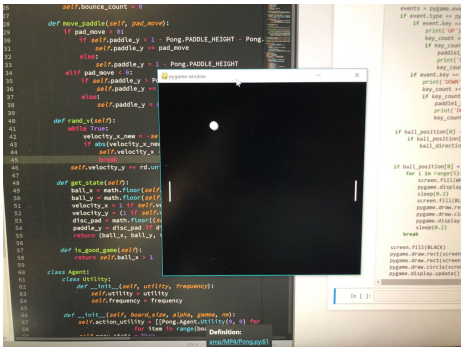

Figure 3: Human vs. AI

3 Behavioral Cloning and Deep Learning

3.1 Policy Behavioral Cloning

3.1.1 Implementation

The goal of this section is to design neural networks which helps the agent to play the game of Pong. Specifically, we use the Minibatch Gradient-Based Optimization technique to train the network, and within the netword, we use softmax cross-entropy loss algorithm for classification.

Instructed by the documents, we have implemented the forward and backward propagation functions for the network. Forward and backward propagation is equally essential to training the network. During a forward propagation, we find the output from the net and compare with the golden labels to get errors. Then, in a backward propagation, we find the derivative of error with respect to the weights and then subtract the values from the weights. By propagating forward and backward, we find out the performance of the model and then try to minimize the error and updates the model.

The algorithms for building the net are explicitly introduced in the documents. We have followed the formulas and implemented the THREE_LAYER_NETWORK and MINIBATCH GD based on the provided pseudo-code.

3.1.2 Parameters

When training the neural net, there are mainly three parameters to consider: number of epochs, size of batches, and number of features. Intuitively, more epoch and more features can lead to better models. Yet larger batches do not necessarily produce better results. Therefore, we have tried to use batch size of 100 with 64 features, and this combination gives fair losses and good accuracy.

3.1.3 Discussions and Results

First, answering the question from the documents, one of the significant advantage of neural network over a Q-table is the saving of space. Most intuitively, in order to construct a Q-table, we would need to discretize the game space so that the size of the Q-table is manageable. In other words, a Q-table implement cannot possibly handle continuous space problems due to its discrete nature. In addition, when an agent cannot find a state in the Q-table, he would "lost his mind" because he does not have the specific information to solve.

As for the result of the implementations, we have achieved an overall accuracy of 89.7% under 100 epochs. We used the parameters discussed before, a batch size of 100 with 64 feature. We strongly believe that the results can be better if we give it more training and maybe explore more options of parameters. The following is the confusion matrix of the classification.

	0	1	2
0	0.89	0.09	0.04
1	0.03	0.85	0.03
2	0.08	0.06	0.92

Here is a curve that shows the loss with respect to the number of epochs. It is clear that more epochs can improve on the loss.

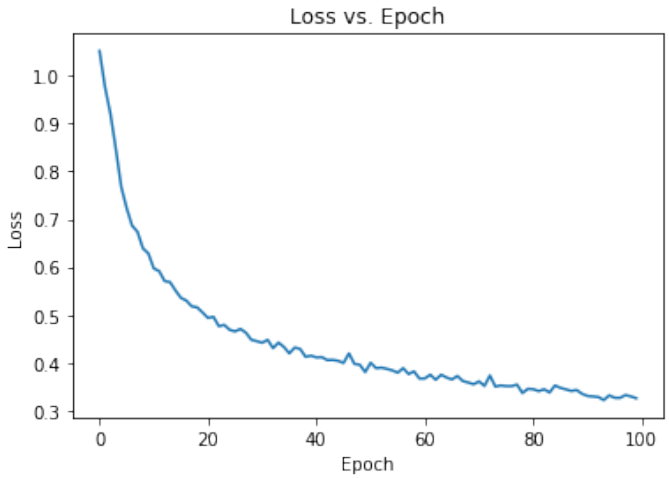

Figure 4: Loss vs. Epoch

With the weights that is produced from the neural net, we are able to have an average bounce of 9.64 in 200 games. It is somewhat lower than the results in part 1. We think it is due to the limited trainings we had on this model.

4 Credits

We worked together wonderfully in coding the solutions. Everyone contributed in constructing the algorithms. Junhao Pan coded and typed the report in LaTeX, and the others reviewed it.