# CS 440 Programming Assignment 4

3 Credit Section Q. Dec 11, 2017

Group members: Dachun Sun(dsun18), Yuxi Gu(yuxigu2), Ruoxi Yang(ryang28)

For the first part, we used the high-quality package NumPy to implement the multiclass perceptron in Python. We loop through the whole dataset for several epochs to train the weight vector according to the wrong classified test examples. Then, we multiply the weight matrix with the example to get the scores for each class and find the largest one as the classified label.

For Part 2, we use Python to implement the Pong game logic, and PyGame to implement the GUI. For each simulation cycle, we choose an action based on the Q-table and the exploration policy and then perform the TD update.

We have done Part 1 and Part 2 with all the extra credit.

Statement of individual contribution:

| | |
|---|---|
| Dachun Sun: | Implement basic multiclass perceptron classifier, implemented Pong game using PyGame created the Q-learning algorithm structure. |
| Ruoxi Yang: | Performed multiple training session with different parameters, wrote the report for Part 1 |
| Yuxi Gu: | Modified the MDP in Part 2, trained, and tested the model. Summarized and wrote the report for Part 2. |

## Part 1.1

**Algorithm Overview:**

We inherited data parsing method from Programming Assignment 3 to obtain the dataset from the text file. As for training, we use the multi-class perceptron method for implementation. Keeping a weight vector $w_c$ for each class $c$ (from 0-9, each represents a label), we use $c^* = \underset{c}{\operatorname{argmax}}(\boldsymbol{w}_{\boldsymbol{c}}^T \boldsymbol{x} + b)$ for decision rule. To update the weights or bias, if example from class $c$ gets misclassified as $c'$:

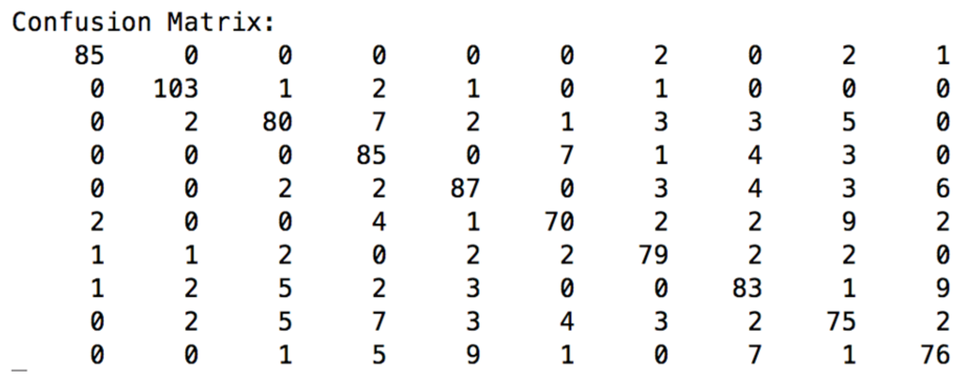
- Update for $c$: $\boldsymbol{w}_{\boldsymbol{c}} \leftarrow \boldsymbol{w}_{\boldsymbol{c}} + \alpha \boldsymbol{x}$, $b \leftarrow b + \alpha$
- Update for $c'$: $\boldsymbol{w}_{\boldsymbol{c'}} \leftarrow \boldsymbol{w}_{\boldsymbol{c'}} - \alpha \boldsymbol{x}$, $b \leftarrow b - \alpha$

where $\boldsymbol{x}$ is a training example, $b$ is the bias, and $\alpha$ is the learning rate. We also permute the training set after each epoch to avoid assuming neighborhood relation between perceptron.

**Parameter Setting:**

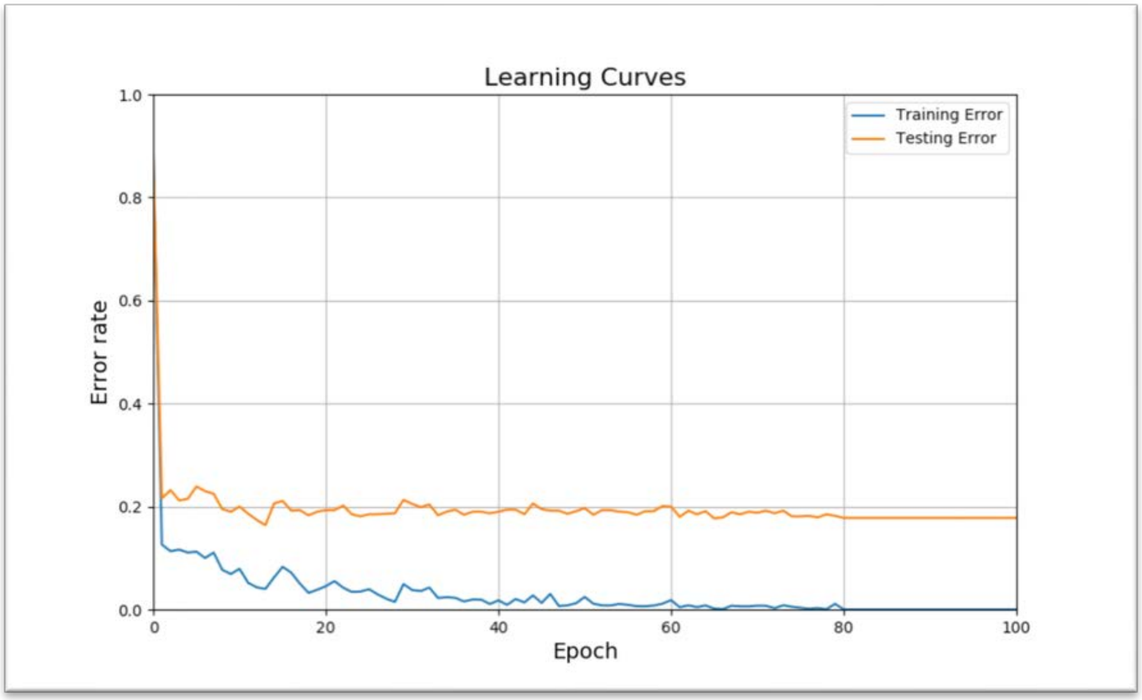
- Learning rate decay function: we choose 1.0 from (1.0, 5.0,10.0) based on performance.
- Bias vs. no bias: We choose bias over no bias based on performance. There is not too much improvement, but the bias allows the decision boundary to shift, which improves the result.
- Initialization of weights (zeros vs. random): We choose zero over random. As the convergence is guaranteed, the training error both drops, but the randomized version converges a little slowly.
- Ordering of training examples (fixed vs. random): We choose random over fixed based on performance
- The number of epochs: among 50, 100, 1000 number of epochs, we choose 100 because it achieves good accuracy in a reasonable time.

**Experiment Result:**

- Overall accuracy: **82.3%**
- 

```
Confusion Matrix:
     85      0      0      0      0      0      2      0      2      1
      0    103      1      2      1      0      1      0      0      0
      0      2     80      7      2      1      3      3      5      0
      0      0      0     85      0      7      1      4      3      0
      0      0      2      2     87      0      3      4      3      6
      2      0      0      4      1     70      2      2      9      2
      1      1      2      0      2      2     79      2      2      0
      1      2      5      2      3      0      0     83      1      9
      0      2      5      7      3      4      3      2     75      2
      0      0      1      5      9      1      0      7      1     76
_
```

- Learning curve:



- Naïve Bayes classifier accuracy: **77.3%**

  We find that the performance of multiclass perceptron is better than Naïve Bayes classifier, and Naïve Bayes classifier training time is faster than multiclass perceptron. The better accuracy results from the fact that the Perceptron considers the linear combination of each feature, while the Naïve Bayes classifier assuming each feature as independent to each other, which prevents it to achieve better accuracy. Multiclass perceptron might train slowly with the increase of the number of epoch number as it must loop through the whole dataset many times in series, but it can achieve better accuracy due to complex interaction and the decision between attributes in the training process. Naïve Bayes classifier training process is mainly counting the frequency, so the calculation can be parallelized.

## Part 1.2

**Distance function**: Among cosine similarity, Euclidean distance and Manhattan distance methods, we find that using Euclidean distance as distance/similarity function achieve the best result regarding the accuracy. On the other hand, in the realm of image processing, Euclidean distance also calls SSD (Sum of Squared Differences), which is mainly used as template matching. Euclidean distance treated each dimension equally, thus avoid extreme cases.

The testing time is about 52 seconds. With the increase of k, the runtime does not change much. This is first because of $k \ll n$, and the implementation we use is to sort the whole difference array to select the nearest neighbors, thus increasing of $k$ does not make much difference to the overall test time.

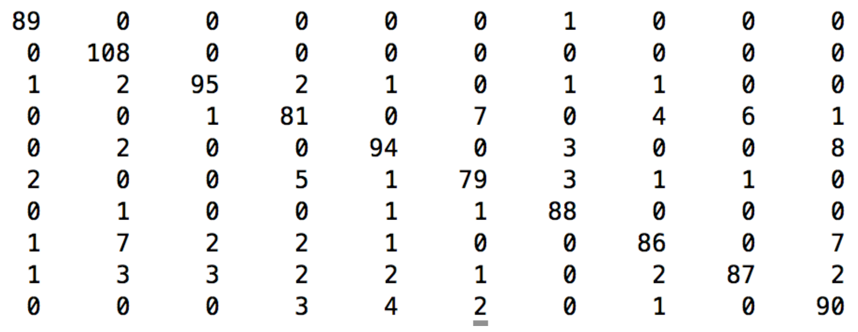
The overall accuracy using K-Nearest Neighbor is higher than using Naïve Bayes classifier or multiclass perceptron. One possible reason for this result is the data corresponding to each label might share many similarities thus easy to be categorized and selected by the similarity function. Moreover, K-Nearest Neighbor is easier to build since its model is the dataset itself. The cons of using K-Nearest Neighbor is its relatively slow testing time comparing to other two methods.

The overall accuracy using K-Nearest Neighbor:

| $k$ | Accuracy | Test Running Time(sec) |
|---|---|---|
| 1 | 89.7% | 52.702 |
| 2 | 87.9% | 52.607 |
| 3 | 88.7% | 52.630 |
| 4 | 89.3% | 52.585 |
| 5 | 89.1% | 53.273 |
| 6 | 87.4% | 55.221 |
| 7 | 87.3% | 54.977 |
| 8 | 87.1% | 54.877 |
| 9 | 86.5% | 55.131 |

**Experiment Result:**

- Overall accuracy: **89.7%**
- Confusion Matrix:



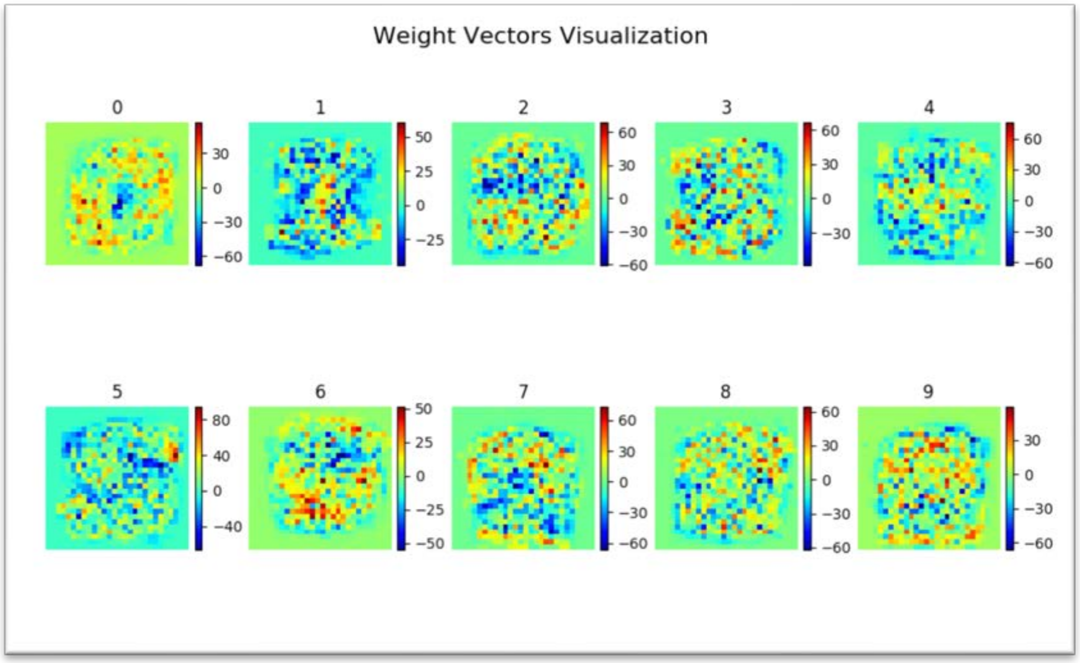
```
89       0      0      0      0      0      1      0      0      0
 0     108      0      0      0      0      0      0      0      0
 1       2     95      2      1      0      1      1      0      0
 0       0      1     81      0      7      0      4      6      1
 0       2      0      0     94      0      3      0      0      8
 2       0      0      5      1     79      3      1      1      0
 0       1      0      0      1      1     88      0      0      0
 1       7      2      2      1      0      0     86      0      7
 1       3      3      2      2      1      0      2     87      2
 0       0      0      3      4      2      0      1      0     90
```

## Part 1 Extra Credit:

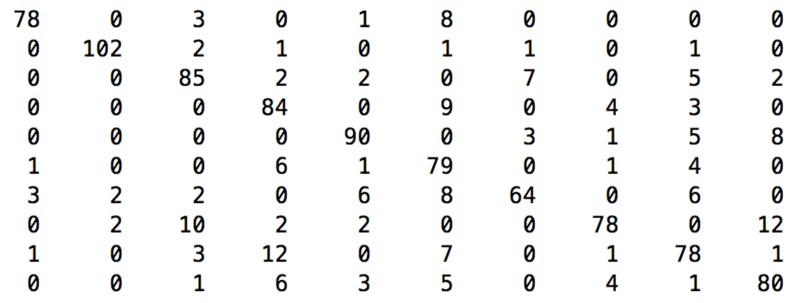
1. With 2*4 patches and enable of overlapping patches, we could achieve a better accuracy. This is similar to first convolute the digit image, essentially transform the feature.
   - Overall accuracy: **82.8%**
   - Confusion Matrix:

```
87     0     0     0     0     0     1     0     1     1
 0   103     2     1     1     0     1     0     0     0
 0     2    88     4     1     0     3     3     1     1
 0     0     2    85     0     5     1     4     3     0
 0     0     2     1    90     0     3     2     1     8
 2     0     3     3     0    66     5     2    10     1
 1     1     3     0     2     1    81     1     1     0
 1     2     4     3     3     0     0    79     1    13
 0     2     6     6     3     7     3     2    68     6
 0     0     2     4     6     2     0     4     1    81
```

2. From weight vectors visualization down below, we find out that each weight vector is very intuitive to the label it corresponds to. The most discriminative part of each digit in the graph is the clusters that either has large positive weight or small negative weight. We can interpret these weights as forming templates of the output classes. With the value of weight being high, we can anticipate that it tends to line up with high-valued pixels to making the score high as well. Also, with the value of weight being very negative, it discourages the example. For example, in label 0's weight vector, the high value (yellow and orange region) corresponds to the circular pattern of 0, the low value (deep blue region) in the center means that if the test example contains pixel in the middle, it is unlikely to be classified as 0.

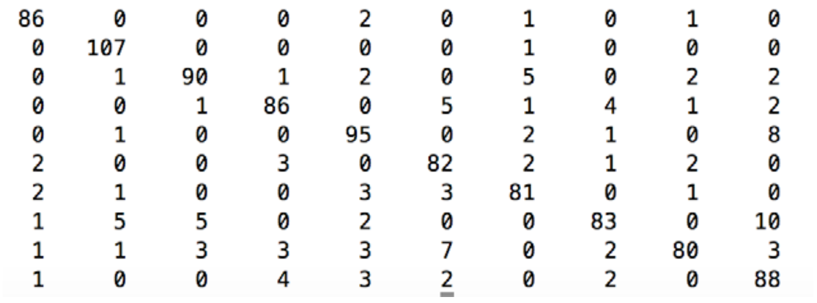


3. By using SoftMax function as classification rule and updating the weights using Gradient Descent, we implement perceptron with differentiable nonlinearity. Using the same parameter setting as used in Part 1.1, we get result roughly the same as the previous one using multiclass perceptron. This is due to the expressivity of the model since this model is still based on the dot product the weight vector and the example, there is not much nonlinearity involved in the decision boundary.
    - Overall accuracy: **81.8%**
    - Confusion Matrix:



```
78     0     3     0     1     8     0     0     0     0
 0   102     2     1     0     1     1     0     1     0
 0     0    85     2     2     0     7     0     5     2
 0     0     0    84     0     9     0     4     3     0
 0     0     0     0    90     0     3     1     5     8
 1     0     0     6     1    79     0     1     4     0
 3     2     2     0     6     8    64     0     6     0
 0     2    10     2     2     0     0    78     0    12
 1     0     3    12     0     7     0     1    78     1
 0     0     1     6     3     5     0     4     1    80
```

4. By using SVM library provided by scikit-learn, we achieved the following result.
    - Overall accuracy: **87.8%**
    - Confusion Matrix:



```
86     0     0     0     2     0     1     0     1     0
 0   107     0     0     0     0     1     0     0     0
 0     1    90     1     2     0     5     0     2     2
 0     0     1    86     0     5     1     4     1     2
 0     1     0     0    95     0     2     1     0     8
 2     0     0     3     0    82     2     1     2     0
 2     1     0     0     3     3    81     0     1     0
 1     5     5     0     2     0     0    83     0    10
 1     1     3     3     3     7     0     2    80     3
 1     0     0     4     3     2     0     2     0    88
```

## Part 2.1

$\alpha = 0.7$, $\gamma = 0.6$, and the exploration function: $f(u,n) = \begin{cases} R^+ \ if \ n < N_e \\ u \ otherwise \end{cases}$

where $u$ is the Q value of a state-action pair, $n$ is the number of times a state-action pair is chosen, R$^+$ is an optimistic estimate of the best possible reward obtainable in any state, and $N_e$ is a fixed parameter, acting as a threshold.

We use TD algorithm to train our agent and fill the Q-table with the following update:

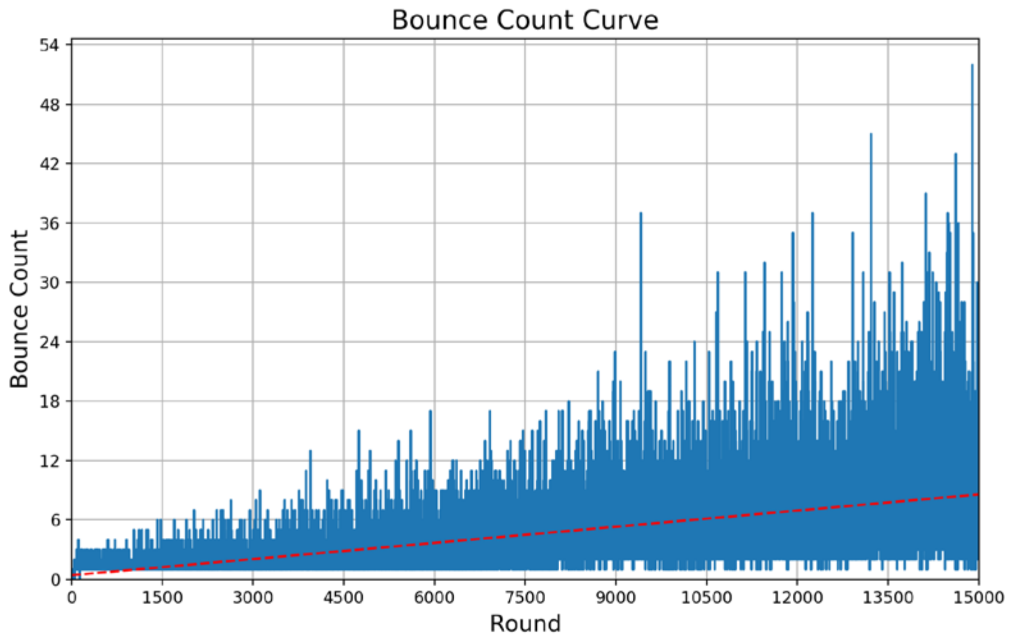
$$Q(s,a) \leftarrow Q(s,a) + \frac{\alpha C}{C + N(s,a)}\left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

where $(s, a, s')$ is current state, action, and successor state sequence, $\alpha$ is the learning rate, $C$ is the decay factor, $\gamma$ is the discount factor, and $N(s,a)$ is the count of a certain state-action pair is chosen.

The decision over $\alpha, \gamma$ and the decay factor was obtained through experiments. The table below showed part of the results we obtained from our experiments and justified our choice for $\alpha, \gamma$ and the decay factor. The bounce count referred to the average number of times that the ball bounced off the paddle before it escaped past the paddle per game, and each configuration is trained for 15,000 games.

| $\alpha$ | $\gamma$ | Decay Factor | Bounce Count |
|---|---|---|---|
| 0.1 | 0.1 | 1000 | 5.61 |
| 0.1 | 0.1 | 5000 | 5.34 |
| 0.1 | 0.1 | 9000 | 6.45 |
| 0.1 | 0.1 | 13000 | 5.54 |
| 0.1 | 0.1 | 17000 | 6.38 |
| 0.1 | 0.1 | 21000 | 5.29 |
| 0.1 | 0.1 | 25000 | 5.12 |
| 0.1 | 0.1 | 29000 | 6.25 |
| 0.1 | 0.2 | 1000 | 6.49 |
| 0.1 | 0.2 | 5000 | 6.32 |
| .... | .... | ..... | ..... |
| **0.7** | **0.6** | **1000** | **9.97** |
| 0.7 | 0.6 | 5000 | 4.88 |
| 0.7 | 0.6 | 9000 | 6.75 |
| 0.7 | 0.6 | 13000 | 6.82 |
| 0.7 | 0.6 | 17000 | 6.56 |
| 0.7 | 0.6 | 21000 | 5.95 |
| 0.7 | 0.6 | 25000 | 7.04 |
| 0.7 | 0.6 | 29000 | 6.14 |
| .... | .... | .... | ..... |
| 0.9 | 0.9 | 5000 | 6.25 |
| 0.9 | 0.9 | 9000 | 6.41 |
| 0.9 | 0.9 | 13000 | 7.21 |

*Part of the result was hidden due to a large number of configurations of parameters, please check the folder named "param_tuning_result" for more detailed experimentation results.

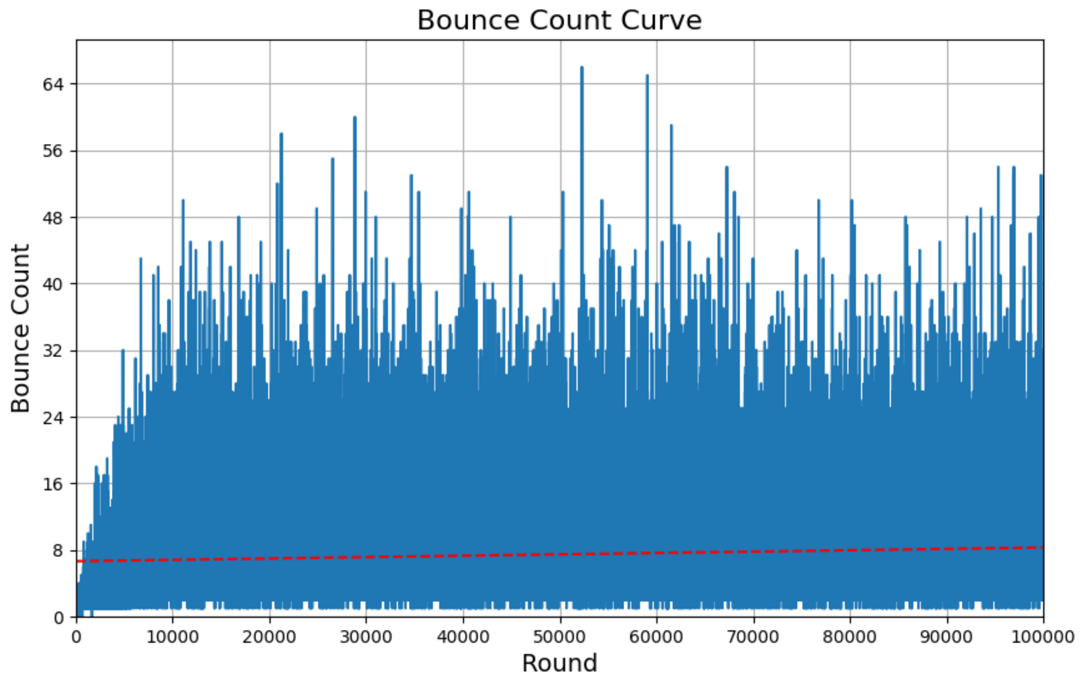


The agent needs to simulate about 15,000 games before it learns a good policy, which can be observed from the red trend line in the above image. In this image, when the round is about 15,000, the bounce counts is about 9.

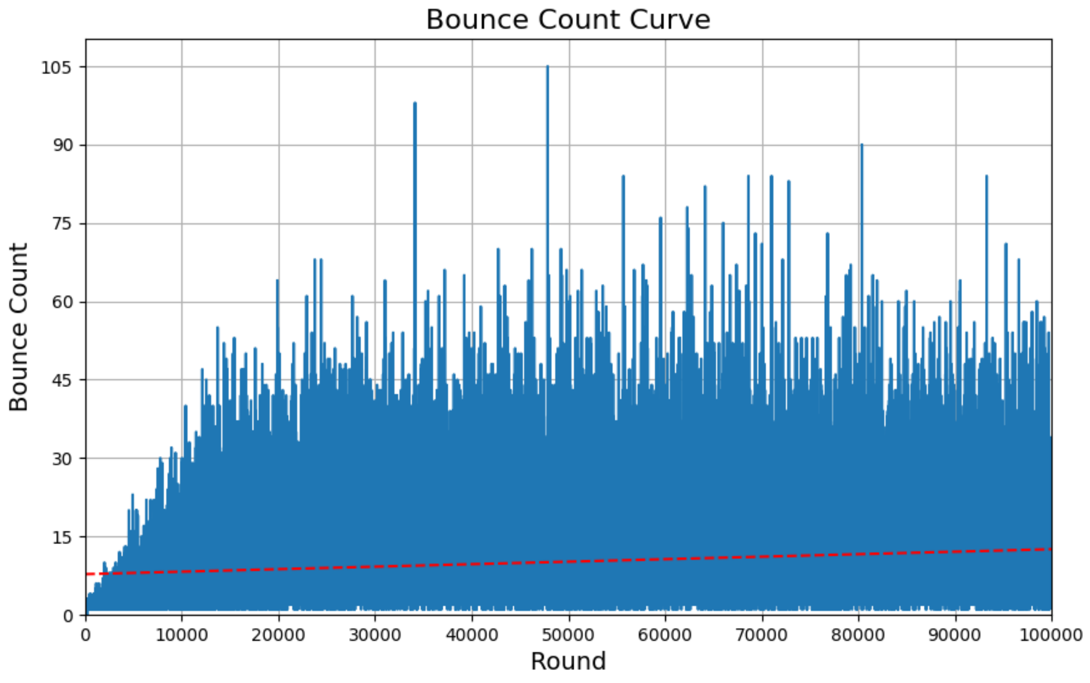
With $\alpha = 0.7$, $\gamma$ = 0.6, decay factor = 1000, and the exploration function specified at the beginning, the training ran for 100,000 games. Average number of times the ball bounces off the paddle per game calculated through 1000 test runs is **14.01**.

We changed the number of discrete value for ball's $x$ position, ball's $y$ position, and paddle's $y$ position. We did this by adding a functionality that allows these three numbers to be assigned manually. The game simulation is not affected by how we discretize the continuous state space, so the training and testing time for the agent does not change much. The fluctuation is more likely due to changed policy would take different time to finish each round of simulation, and the efficiency of Hash map implementation in Python.

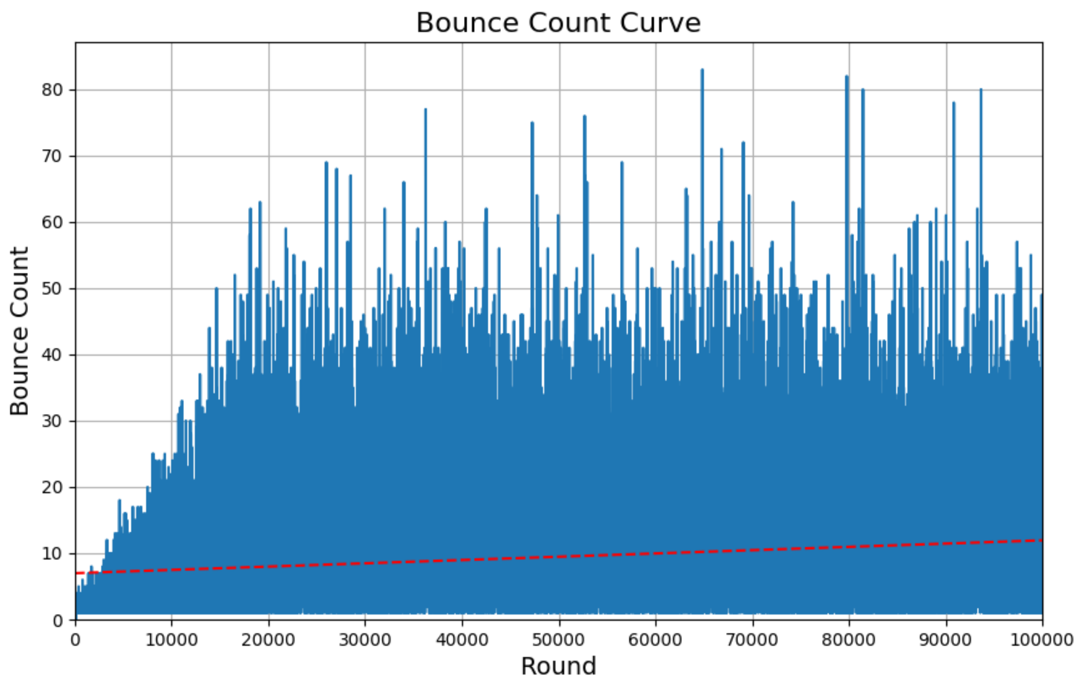
The interesting fact we observed was that the smaller value we assigned to the number of discrete value for ball's $x$ position, $y$ position and paddle's $y$ position, the bounce count curve converged to a good policy more quickly, but not as good. Below are some figures that proved our observation, with $\boldsymbol{xd}$ represents the number of discrete value for ball's $x$ position, $\boldsymbol{yd}$ represents the number of discrete value for ball's $y$ position and $\boldsymbol{pd}$ represents the number of discrete value for paddle's $y$ position.



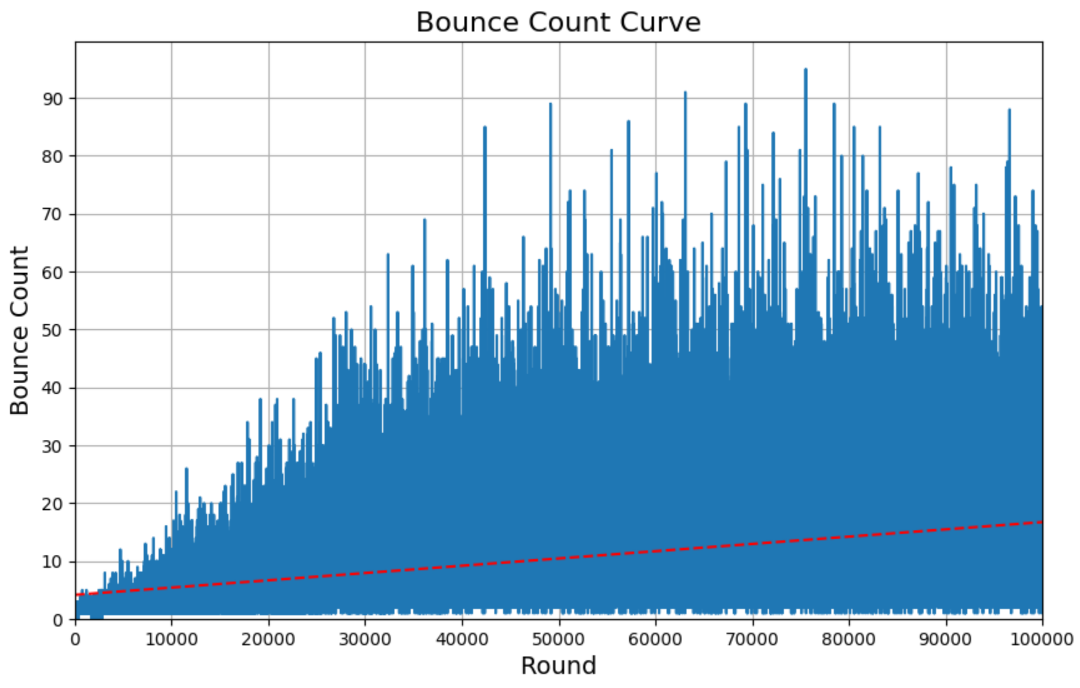
Bounce Count Curve ($xd = 8, yd = 8, pd = 8$)



Bounce Count Curve ($xd = 10, yd = 10, pd = 10$)



Bounce Count Curve ($xd = 12, yd = 12, pd = 8$)



Bounce Count Curve ($xd = 14, yd = 14, pd = 14$)

The following are the results after running 1000 test games with trained agents under different ways of discretizing states,

| $xd$ | $yd$ | $pd$ | Average Bounce Count |
|---|---|---|---|
| 8 | 8 | 8 | 7.63 |
| 10 | 10 | 10 | 11.66 |
| 12 | 12 | 8 | 10.54 |
| 14 | 14 | 14 | 13.93 |

We can also conclude from the above data that if the continuous state space is converted into a more discrete, finite state space, the average number of times that the ball bounces off the paddle per game before it escapes past the paddle will become larger for some configurations and then decrease.

## Part 2.2

We did not make any change to actions in the MDP. However, we added one additional state to represent whether the right player (refers to the player using the right paddle, e.g. the agent) wins. It is because, in Part 2.2, one player plays against another player, so we need an additional state to represent the fact the right player wins. We also changed the reward model so that when the ball passes the left paddle, the fully trained AI agent will get a reward of 1 since its opponent loses.

We changed the number of $\alpha, \gamma,$ and the decay factor. For Part 2, we used $\alpha = 0.3,$ $\gamma = 0.7$, decay = 5000. The value was obtained through experiments. The table below shows part of the results we got from our experiments and justified our choice for $\alpha, \gamma,$ and the decay factor.

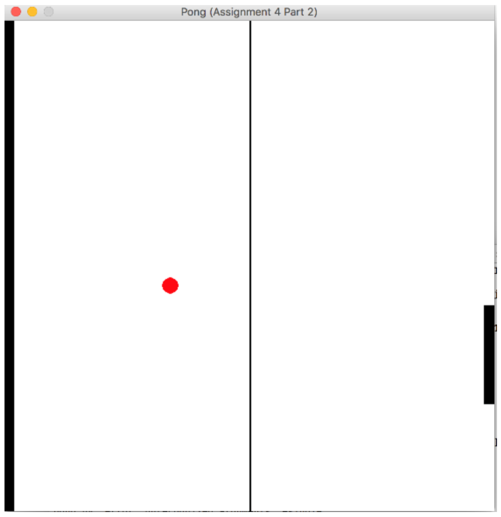
| $\alpha$ | $\gamma$ | Decay Factor | Bounce Count | Winning Rate |
|---|---|---|---|---|
| 0.1 | 0.1 | 1000 | 2.29 | 58.76% |
| 0.1 | 0.1 | 5000 | 2.18 | 55.06% |
| 0.1 | 0.1 | 9000 | 2.22 | 55.76% |
| 0.1 | 0.1 | 13000 | 2.29 | 58.36% |
| 0.1 | 0.1 | 17000 | 2.20 | 59.56% |
| 0.1 | 0.1 | 21000 | 2.23 | 59.86% |
| 0.1 | 0.1 | 25000 | 2.32 | 56.56% |
| 0.1 | 0.1 | 29000 | 2.22 | 57.26% |
| 0.1 | 0.2 | 1000 | 2.29 | 59.96% |
| 0.1 | 0.2 | 5000 | 2.25 | 60.26% |
| .... | .... | ..... | ..... | .... |
| 0.3 | 0.7 | 1000 | 2.42 | 65.87% |
| **0.3** | **0.7** | **5000** | **2.64** | **72.67%** |
| 0.3 | 0.7 | 9000 | 2.36 | 67.37% |
| 0.7 | 0.6 | 13000 | 2.50 | 63.76% |
| 0.7 | 0.6 | 17000 | 2.47 | 61.60% |
| 0.7 | 0.6 | 21000 | 2.44 | 66.37% |
| 0.7 | 0.6 | 25000 | 2.39 | 66.27% |
| 0.7 | 0.6 | 29000 | 2.56 | 69.47% |
| .... | .... | .... | ..... | .... |
| 0.9 | 1.0 | 21000 | 2.25 | 57.86% |
| 0.9 | 1.0 | 25000 | 2.35 | 60.36% |
| 0.9 | 1.0 | 29000 | 2.42 | 64.66% |

*Part of the result was hidden due to a large number of configurations of parameters, please check the folder named "param_tuning_result" for more detailed experimentation results.

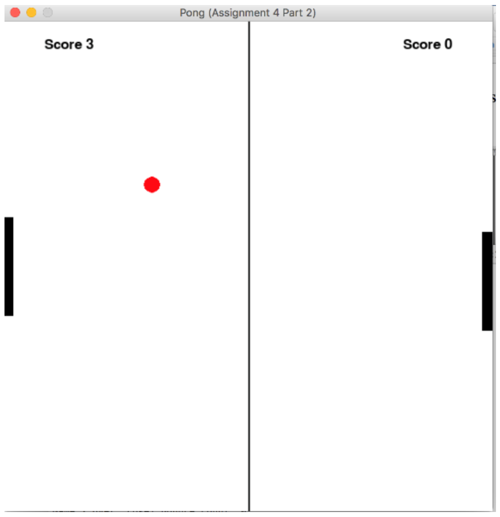
With $\alpha = 0.3$, $\gamma = 0.7$, decay factor = 5000, and the exploration function specified at the beginning, the training ran for 100,000 games. The percentage of the time that our fully-trained AI defeats the hard-coded agent through 1000 test is **85.49%**.

## Part 2.2 Extra Credit

- We use PyGame to create a GUI for our game. Following are the screenshots for the GUI of Part 1 and Part 2.



GUI For Part 1



GUI For Part 2

- AI agent is good at the Pong game. It could respond very quickly no matter how fast the ball moves towards the right paddle. On the other hand, human players may suffer from the randomized speed changes and not be able to respond fast enough. However, since the initial state of the ball is moving to right-bottom, the agent tends to go up after bouncing off the ball, this is probably learned from the returning of the ball after its first hit is usually right-top. Also, the agent cannot respond effectively to the situation where the ball is bouncing up and down very quickly with a slow velocity in the x-direction, but since human players also struggle from the situation, it is predictable that the agent cannot learn how to deal with the situation from the existing state representation.
- For the animation of the agent playing the game for both Part1 and Part2, please check the folder ‘part2/animation_ec3/’ in the source folder.
- We added a gravity to the ball, with the gravity's value equals to 0.001 and its direction points to the right. And we found out that our Q-learning agent became more actively moving towards the predicted position of the ball since it moved back and forth at a small distance much more frequently. Also, because the ball approaches the right paddle faster with the applied gravity, the Q-learning agent has less time to respond and take the corresponding action. To adjust to the faster speed of the ball, the agent had to respond earlier and take its steps more actively.