

CS440/ECE448 Fall 2016

**Artificial Intelligence
Assignment 4**

Work Distribution:

Part 1: Bangqi Wang

Part 2: Yifei Li

2nd Dec. 2016

Part 1: Q-Learning (Pong)

-Bangqi Wang

This part defines the Markov Decision Tree for Pong Game and uses Q-Learning to create agent to play the Pong.

Markov Decision Tree

The Markov Decision Tree uses states, actions, rewards, initial state, and termination to define the decision process.

State for Pong is defined by the tuple (ball_x, ball_y, velocity_x, velocity_y, paddle_y). It is impossible to define the Markov Decision Tree for continuous moving, so this project divides the x and y domain into 12 parts. The total game board will be divided into $12 \times 12 = 144$ different areas. The velocity_x has two directions and velocity_y has three directions. The paddle can move in three directions. The total number of states is $12 \times 12 \times 2 \times 3 \times 3 = 10,369$.

Actions for the paddle can only move in y direction and be chosen from moving up, moving down, or stay. Each state can have 3 different actions.

Rewards only have two cases: -1 for missing the ball, and +1 for rebounding the ball.

Initial state for Pong is to set the ball in the center of the screen and the paddle in the middle of the right boundary.

Terminating the game when the paddle fails to rebound the ball.

Q-Learning

Q-Learning is one of the reinforcement learning models that could be used to find the optimal action-selection policy for any given Markov Decision Tree. The Q-Learning follow the algorithm below: (<http://mnemstudio.org/path-finding-q-learning-tutorial.htm>)

Algorithm

1. Set the alpha, gamma parameter, and environment rewards.
2. Initialize Q value matrix with dimension states * actions.

$$Q : S \times A \rightarrow \mathbb{R}$$

3. For each episode (game):
 - a. Select possible action and calculate the Q value for states reached.
 - b. Find the max Q value from reached states.
 - c. Update the Q value for current state with equation below.
 - d. Set current state as next state.
 - e.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

(From <https://en.wikipedia.org/wiki/Q-learning>)

Parameter

Alpha is the learning rate for Q-Learning. The learning rate defines the weight ratio between old knowledge and new knowledge. The agent will learn nothing if the factor is 0. The factor of 1 will give balance evenly to old and new knowledge.

Gama is the discount factor that determines the importance of future reward. The agent with less Gama factor will become more myopic. The agent has factor of 0 will only focus on current situation.

Epsilon is the factor describes the “spirit of adventure”. The agent with higher epsilon are more willing to take risk action instead of following the Q value matrix.

Part 1.1: Interpretation of Parameter

The parameters in this part directly affect the performance of the agent. In order to find the best performance, I need to define the alpha, gamma, and epsilon parameters one by one.

Gamma: discount factor

This parameter can be defined first. The agent with higher Gamma factor will become wiser in long term. In this case, it is reasonable to make the gamma factor as large as possible. However, if the gamma factor larger than 1, the Q value in matrix may diverge. Therefore, the program needs to find the largest gamma factor in the range from 0 to 1. According to the document:

- For choosing a discount factor, try to make sure that the reward from the previous rebound has a limited effect on the next rebound. In other words, choose your discount factor so that as the ball is approaching your paddle again, the reward from the previous hit has been mostly discounted.

The program needs to choose the largest gamma value that have little influence for the next approaching. The Pong game has 12 states in x dimension and 12 states in y dimension. Therefore, the ball at least changed the state $12 \times 2 = 24$ times before coming back again.

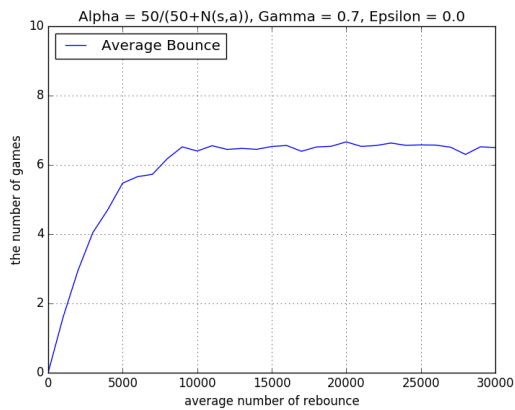
Gamma	Returned Value	
0.9	$0.9^{24} = 0.0798 = 8\%$	
0.8	$0.8^{24} = 0.00472 = 0.5\%$	
0.75	$0.75^{24} = 0.001003 = 0.1\%$	
0.7	$0.7^{24} = 0.00019 = 0.02\%$	✓
0.65	$0.65^{24} = 0.000032 = 0.003\%$	

The return value for 0.7 is about 0.02%, which is small enough to be ignored. Therefore, the program will choose **gamma = 0.7**.

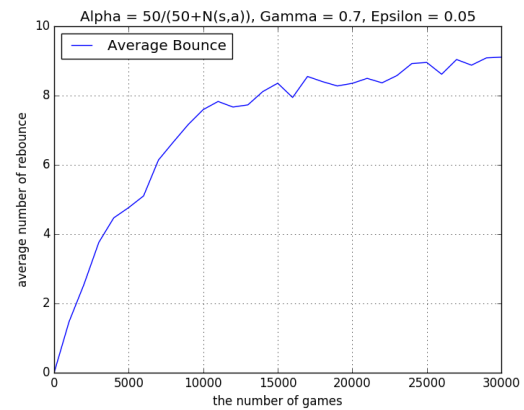
Epsilon: exploration rate

After decided the gamma factor, the program can define the exploration before alpha factor. The epsilon factor stands for the exploration rate. The agent with higher epsilon factor are more likely to explore new action instead of following the Q value matrix. The agent with higher epsilon are more unstable. They may progress or regress significantly. However, if the agent has epsilon factor too low, the agent will have less change to make progress.

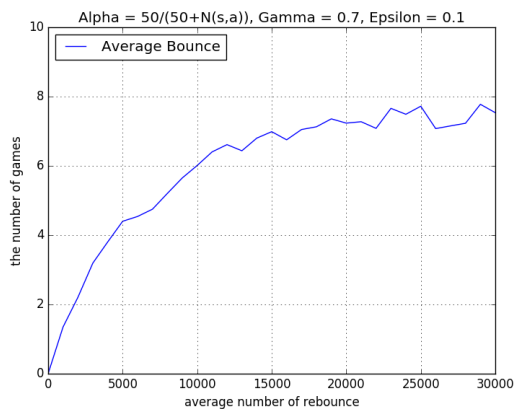
Epsilon in short running (30K)



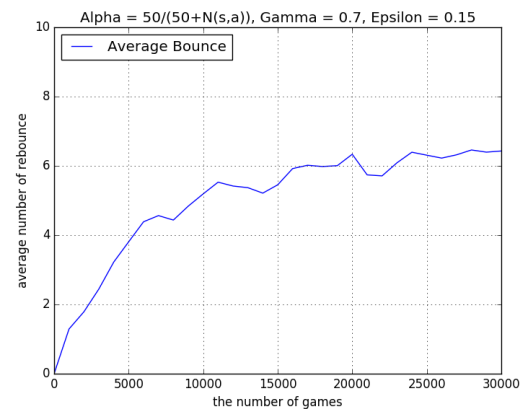
Epsilon = 0.0



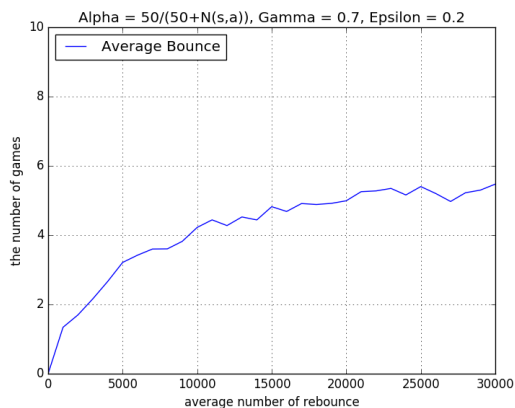
Epsilon = 0.05



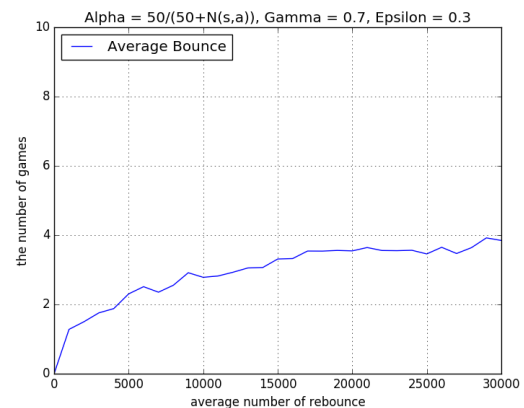
Epsilon = 0.1



Epsilon = 0.15



Epsilon = 0.2



Epsilon = 0.3

The agent with higher epsilon will improve slowly but they may have more breakthrough in long term. According to the table above, the agent with lower epsilon will increase faster but have less opportunity to breakthrough. The agent with epsilon 0.05 has 9 rebounds in 30K games. Therefore, we will choose **epsilon = 0.05**.

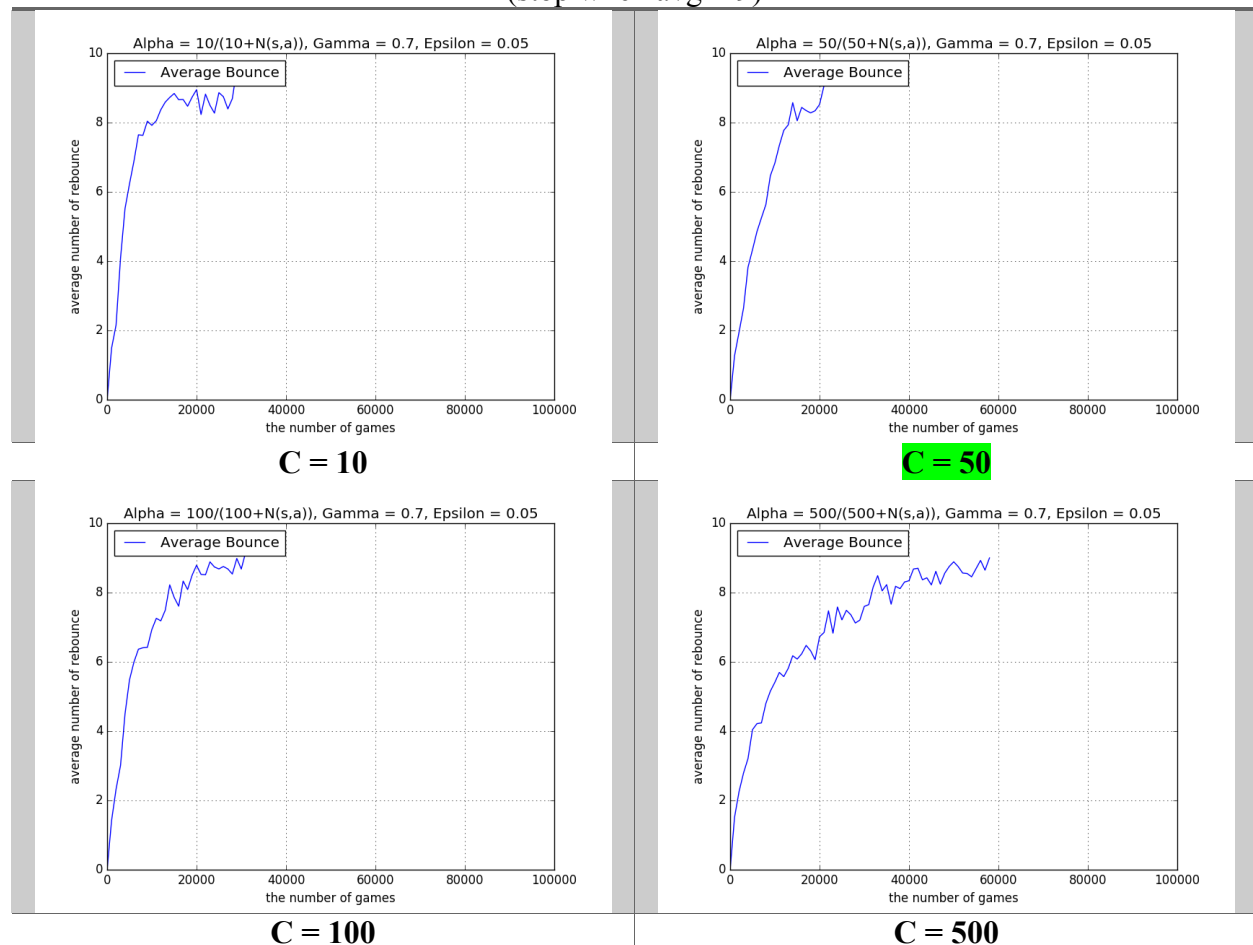
Alpha: learning rate

After determining all other factors, the program needs to determine the learning rate. For the exploration, the program needs to require that each state action be performed infinitely often. The program needs to modify its estimation alongside the learning process. The learning rate controls how fast the program modify its estimation. The agent with higher learning rate will decay faster but has less change to learn and improve.

Learning rate in long running (100K)

$$\text{Alpha} = C / (C + N(s,a))$$

(stop when avg > 9)

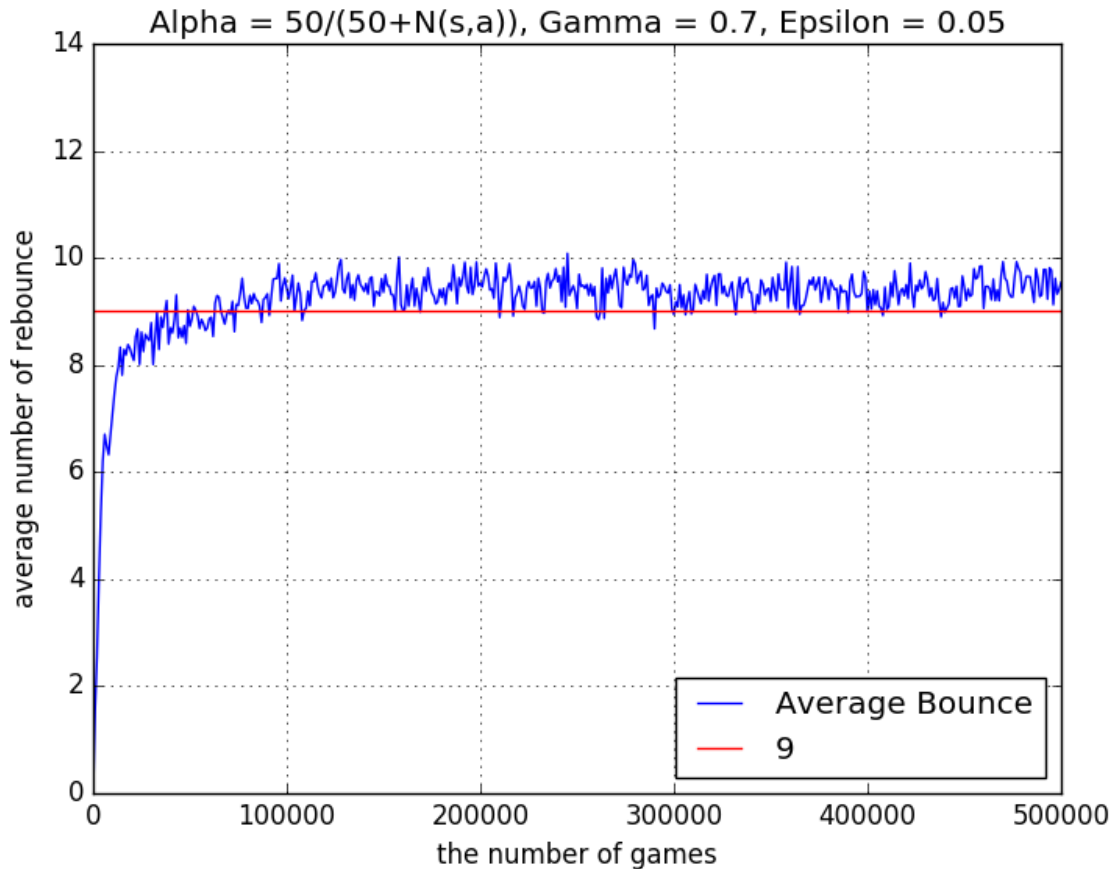


The learning rate will decay to half after saw the state C times. The total running for optimal solution is about 100K and the total states is about 30K. According to my observation, the number of states is around 20k, because some states is extremely hard to reach. Therefore, each states will show about 5 times in the training process. However, states have different possibility to appear. According to the table above, I believe that C = 50 is the most reasonable factor,

because agent with $\alpha = 50 / (50 + N(s,a))$ reaches optimal (average number of rebound is 9) fastest. Therefore, I will choose $C = 50$, and $\alpha = 50 / (50 + N(s,a))$.

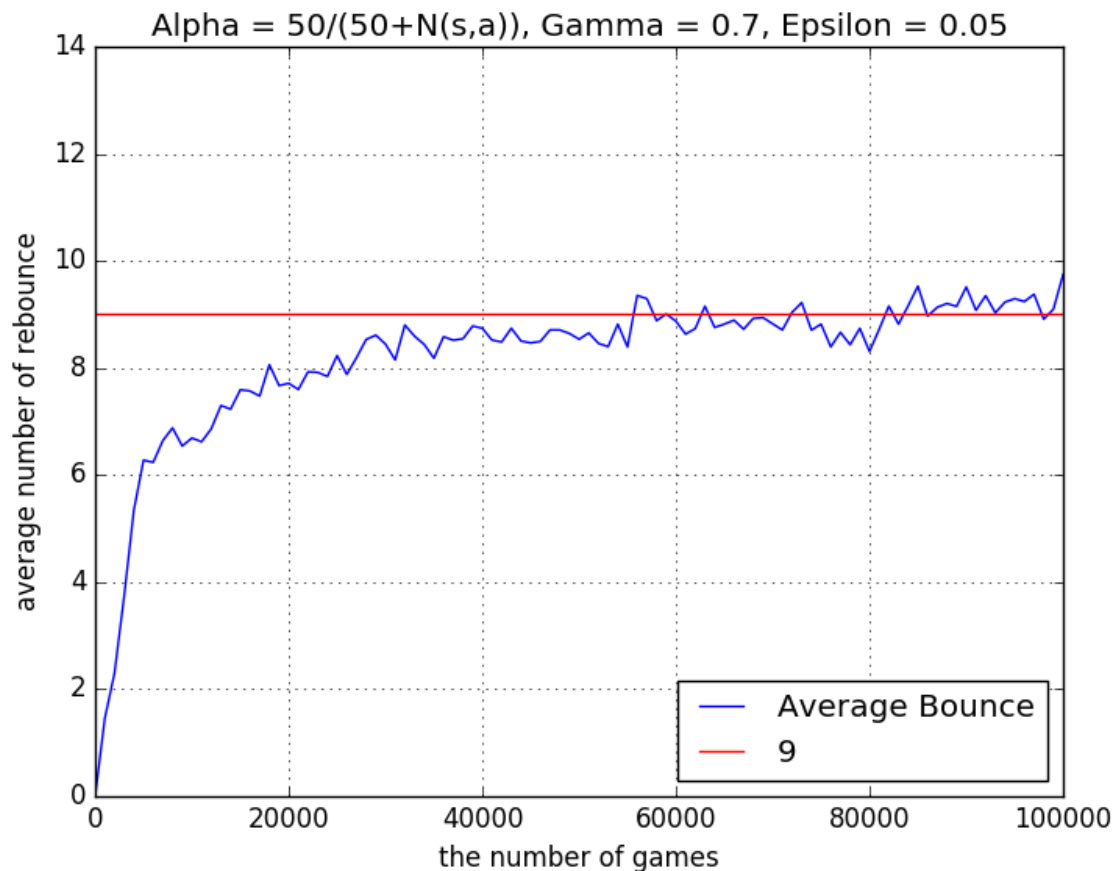
Long term running

I will run the program with $C = 50$, $\gamma = 0.7$, and $\epsilon = 0.05$ for 500K times. The performance reaches optimal in about 100K. The y dimension shows the average number of rebound for most recent 1000 games. The average is in the range of 9 to 10.



MDP Modification

The game board is divided into 12 parts in x dimension and 12 parts in y dimension. Each part has length of 0.084 and velocity_x is larger than 0.03. Therefore, the time for passing one part is at most 2.8 seconds. If the ball has same y value with the middle of paddle. In this case, the paddle does not need to move unless the ball will go beyond the range of half of the paddle height in 2.8 seconds. Therefore, the velocity_y should smaller than 0.036. In this part, I changed the MDP states. The velocity_y is 1 when velocity_y is larger than 0.03, -1 when velocity_y is smaller than -0.03, and 0 in between. As the graph shown below, the performance has similar average bounce but it is more stable.



Part 1.2: Modification

The program in this part has one more agent in the left side.

Modification for MDP

1. State (ball_x, ball_y, velocity_x, velocity_y, paddle1_y, paddle2_y)
2. Rewards:
 - a. Lose = -1
 - b. Win = 1
 - c. Hit = 0.1 (the expectation of rebound is 9 to 10 times with wall)

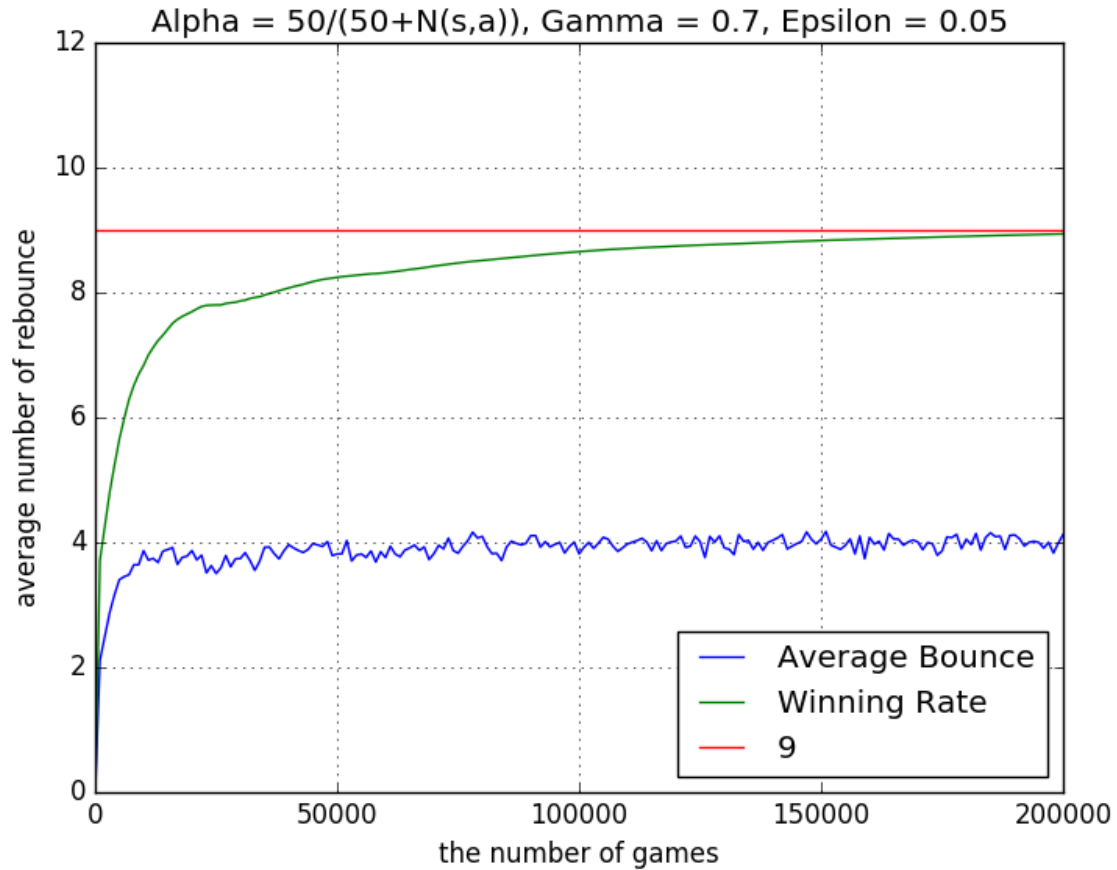
Add one more parameter in state so the side effect is the increase of training times. Modify the reward function so that the win has highest rewards and 10 hits have the same reward.

Modification for Code

Add one more paddle. Add the check function, rebound function, and moving function for this paddle. Add the drawing function for animation.

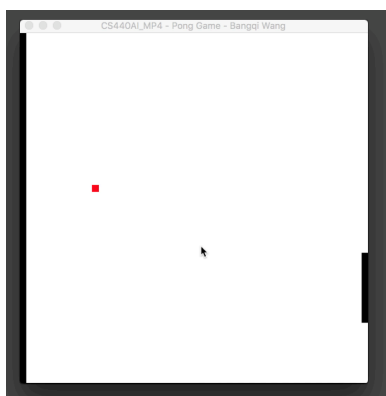
Result

The graph below shows result of this part. The green line represents the number of win out of 10 games. The blue line represents the average number of rebound. The winning rate is near 90%

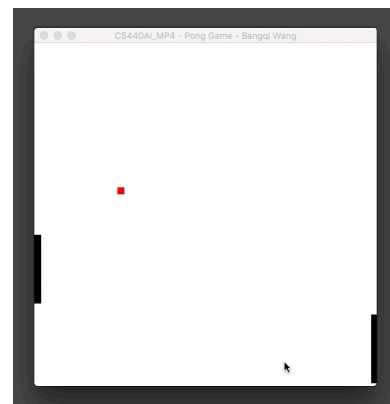


Extra Credit

Video for part1 and part2. Please see the screen recording in the video folder.



part1



part2

Digit Classification with Discriminative Machine Learning Methods

Part 2.1: Digit Classification with Perceptrons

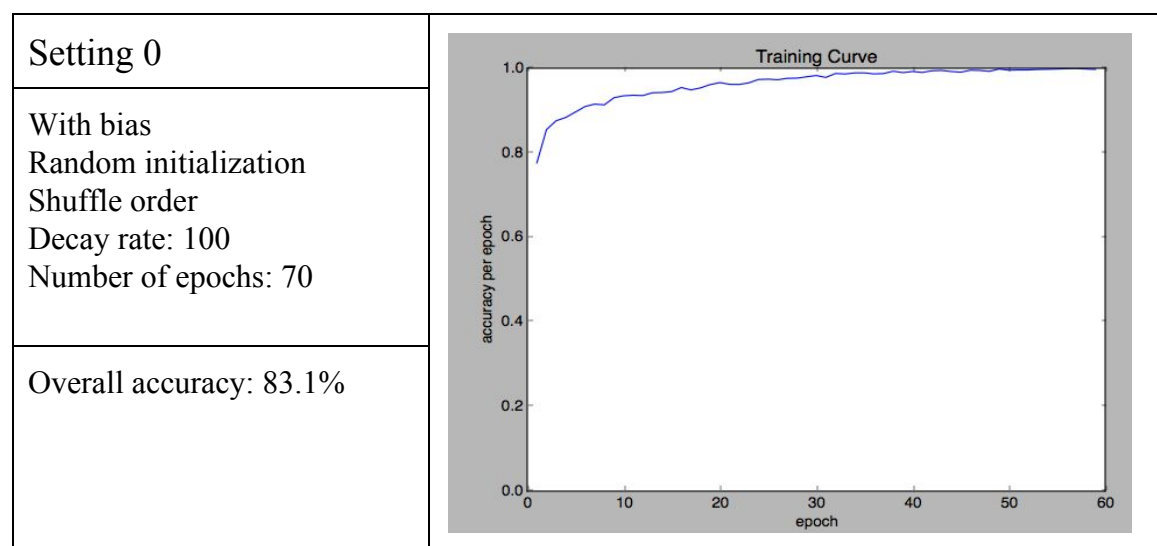
In this part, we will Apply the multi-classes perceptron learning rule from to same digit classification problem from Assignment 3. The feature set consists of a single binary indicator feature for each pixel. Specifically, the feature $F_{i,j}$ indicates the status of the (i,j) -th pixel. 1 and 0 differentiate foreground and background. The images are of size $28*28$, with 784 features. We need to calculate a weight matrix containing weight vector of multi-classes for each image to predict the results.

Implementation

The implementation for perceptrons is straightforward. We first read data from both train data and train label. Then we initialize the weight matrix to be all zero or random numbers depending on different parameter settings. After that, we loop through each training instance number of epoch times. In each iteration, we alter the weight vector according to the result of prediction. If the prediction is the same as label, we remove one instance of misclassification. Otherwise we reduce the weight vector of prediction class by the amount of decay function multiplied by current instance and increase the weight vector of truth class. There are several implementation tricks to be discussed later.

Results & Parameter Setting

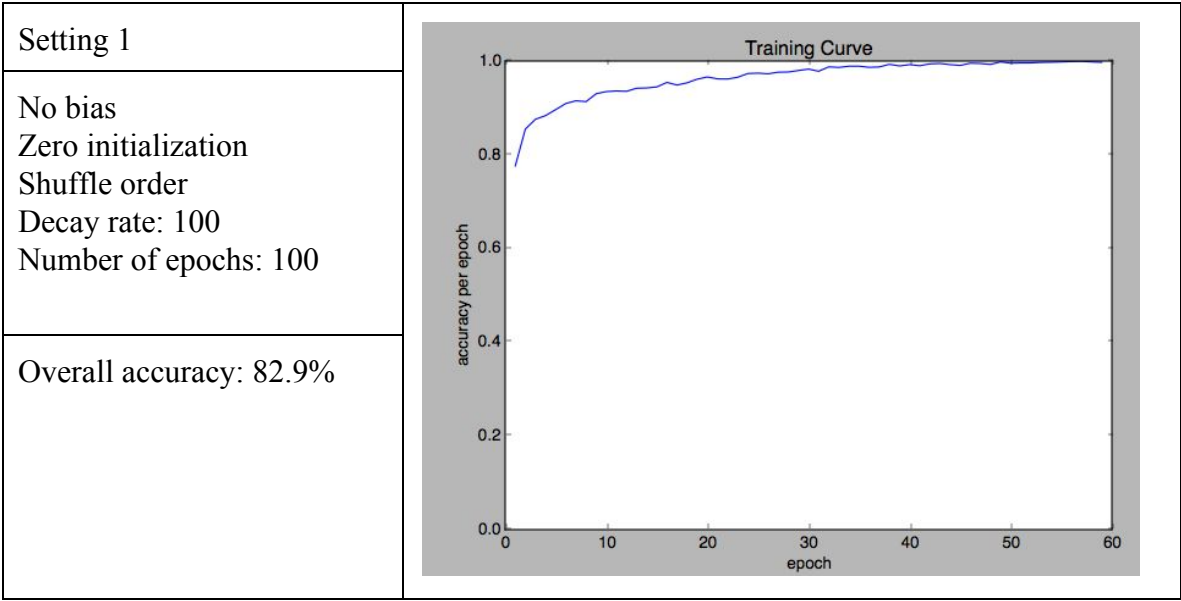
There are five parameters in total, and I make up a table to describe the performance difference between settings. It is complicated to display all setting at once. So we pick three most representative examples.

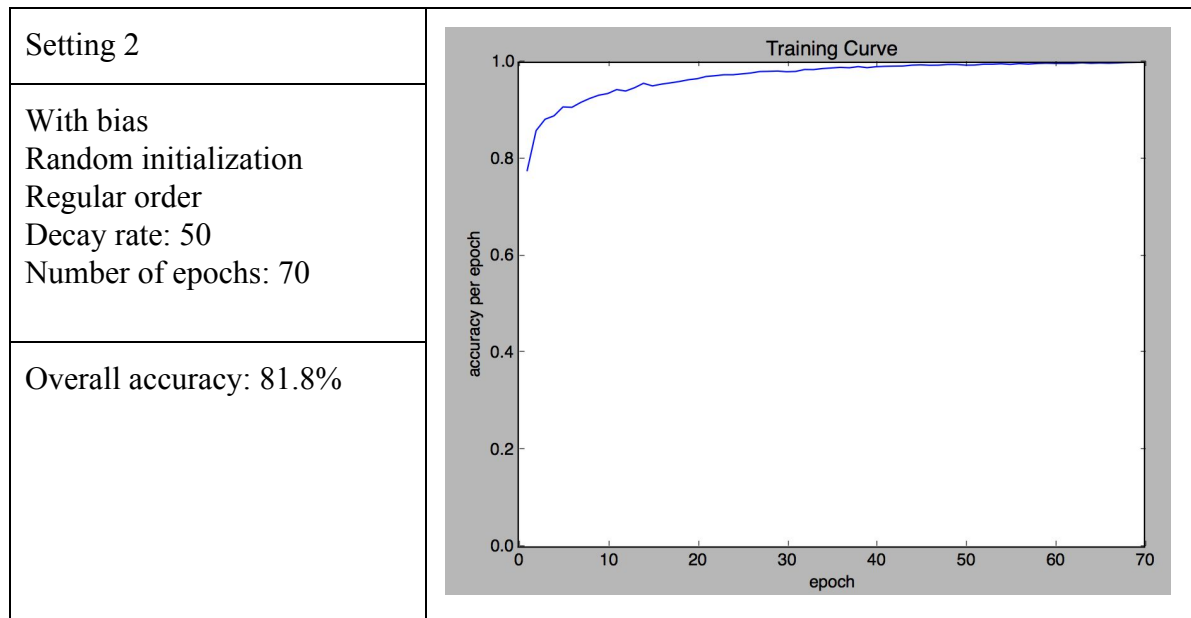


Confusion Matrix for setting 0

0.92	0.00	0.00	0.01	0.00	0.00	0.02	0.00	0.03	0.01
0.00	0.97	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.00
0.00	0.02	0.82	0.03	0.02	0.00	0.03	0.04	0.04	0.00
0.00	0.00	0.03	0.83	0.00	0.07	0.01	0.04	0.02	0.00
0.00	0.00	0.02	0.02	0.82	0.0	0.03	0.03	0.01	0.07
0.01	0.00	0.00	0.05	0.01	0.8	0.01	0.03	0.05	0.02
0.01	0.01	0.02	0.00	0.02	0.03	0.88	0.01	0.01	0.00
0.00	0.03	0.04	0.02	0.03	0.01	0.00	0.82	0.01	0.05
0.00	0.02	0.06	0.07	0.04	0.05	0.03	0.01	0.70	0.03
0.00	0.00	0.01	0.04	0.09	0.01	0.01	0.06	0.01	0.77

Confusion matrix don't vary too much so we only display the best setting





From above results, we observe that decay rate and number of epochs have most significant effect on performance even though our test is not exhaustive.

Perceptrons vs Naive Bayes Classifier

Apparently the perceptron achieves better accuracy than naive bayes classifier in general. The accuracy obtained by Naive Bayes is around 75% while perceptron can gain accuracy of 83%.

Part 2.2 K-nearest neighbors classifier

In this part, we are responsible for implementing a KNN classifier. The general algorithm behind is to, in the training set, find K nearest neighbors for each testing images Then the classifier votes for the most frequent classes as it prediction.

	K=3	K=6	K=10
Euclidean distance	Running Time: 43.568s Accuracy: 90.3%	Running Time:45.198s Accuracy:90.8%	Running Time:48.552s Accuracy:89.7%
Manhattan distance	Running Time: 43.236s Accuracy: 90.1%	Running Time:45.578s Accuracy: 90.2%	Running Time:47.945s Accuracy:88.8%

Confusion Matrix for Euclidean distance, K=6

1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.019	0.039	0.883	0.01	0.00	0.00	0.01	0.029	0.01	0.00
0.00	0.00	0.00	0.86	0.00	0.06	0.00	0.03	0.03	0.02
0.00	0.009	0.00	0.00	0.897	0.00	0.028	0.009	0.00	0.056
0.022	0.011	0.00	0.043	0.00	0.859	0.033	0.011	0.011	0.011
0.00	0.011	0.00	0.00	0.00	0.011	0.978	0.00	0.00	0.00
0.00	0.085	0.009	0.00	0.009	0.00	0.00	0.83	0.00	0.066
0.019	0.019	0.00	0.029	0.01	0.049	0.00	0.019	0.825	0.029
0.01	0.00	0.00	0.02	0.03	0.02	0.00	0.01	0.01	0.9

From observation, we notice that running time is proportional to K and euclidean distance is slightly better than Manhattan distance measure. The best K selection is 6 according to my experient.

KNN Classifier vs Naive Bayes Classifier

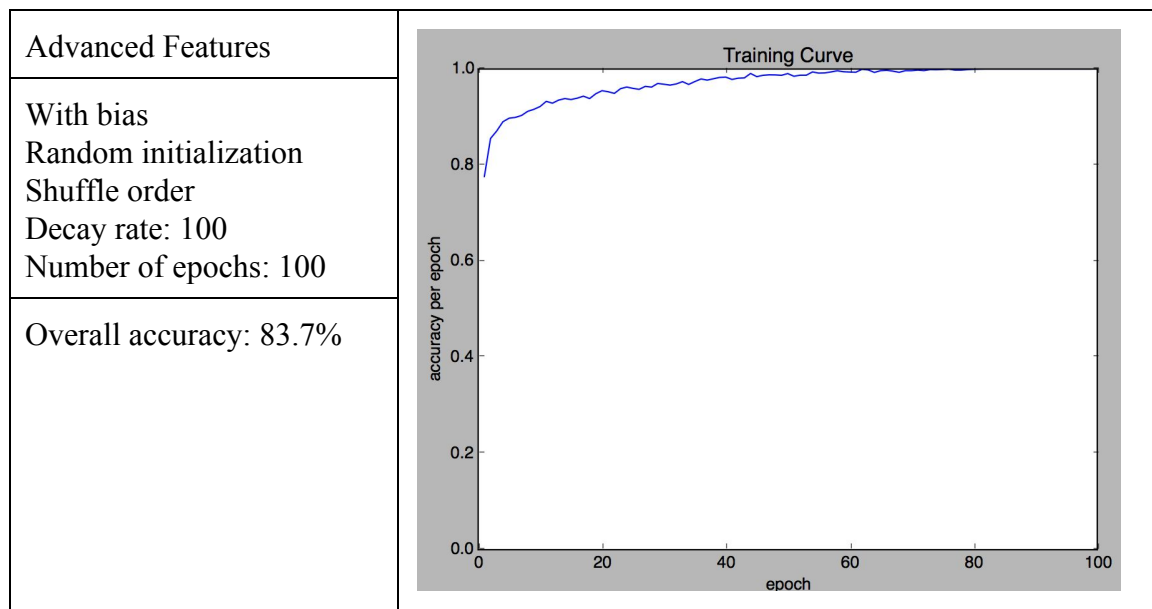
Again, KNN classifier achieves better accuracy than naive bayes classifier in general.

The accuracy obtained by Naive Bayes is around 75% while KNN classifier can gain accuracy of 96%.

Extra Credit

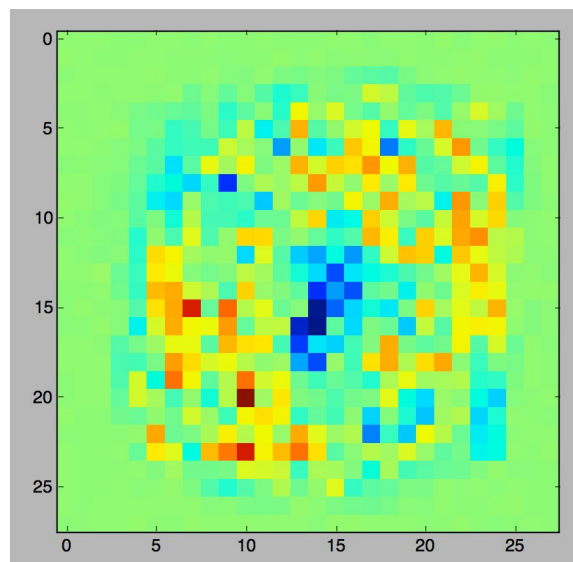
Advanced features:

We incorporate advanced features by simply distribute three values to the digit representation and therefore improved the accuracy by a small amount. Although this is not a boost of accuracy but any improvement should be considered.

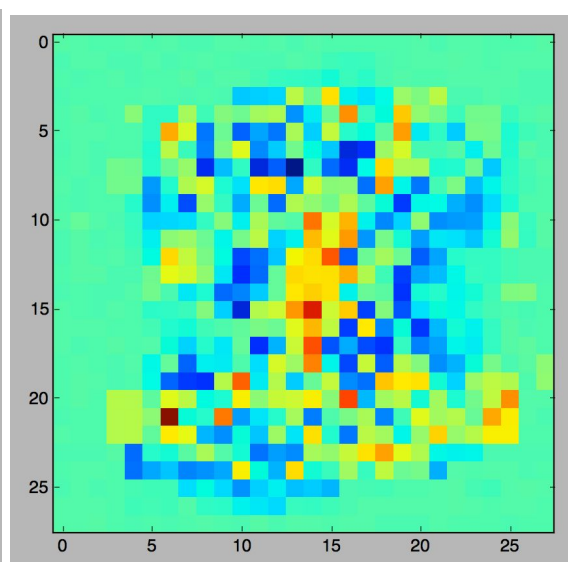


Weight visualization:

We now visualize the perceptron weight in the form of image. In the image, blue represents negative number and yellow represents positive. Green is near 0. We choose 0 and 1 as our example to demonstrate the model.



Weight vector for digit 0



Weight vector for digit 1

From the above images, we can tell that for digit 0, the positive weight forms a cycle in the image, which shows the general pattern of digit 0 while the digit 1 only has positive weight in the middle column, other areas are occupied by negative weights. Therefore, in general, the positive weight represents where the real digit pattern should be and negative weight represents

misclassification. If feature has value in the positive weight area, it will contribute to the classification. On the other hand, value in negative area will counter the classification result.