ECE448 - Machine Problem 4 : Reinforcement Learning and Perceptrons

Shalabi, Yasser
yshalab2@illinois.edu

Luo, Chongxin
cluo5@illinois.edu

Wang, Haoran
hwang293@illinois.edu

November 2016

Contents

Extra Credit:

1 Introduction and Overview

2 Part 1: Q-Learning (Pong)

2.1 Part 1.1: One Player Pong

2.1.1 Introduction

Q-learning is a model-free reinforcement learning techniques that being widely used in machine learning and AI implementations. In this machine problem, A Q-learning based AI agent was implemented for the game pong. The game board was translated into a discrete, finite state space in order to perform Q-learning implementations. The parameters of Q-learning algorithms was studied and tested in order to find the best learning strategy.

2.1.2 Implementations

The pong game is given as figure shown below. The game board is in a 1x1 square, which player1 positioned at the right edge of the board with the paddle size of 0.2. The ball is represented as a single pixel in the game board. The ball has a initial velocity of $v_x = 0.03, v_y = 0.01$. The ball is able to bounds off all three walls (top, bottom, and left) and player1's paddle, and the game is terminated when ball fall off right edge without rebounds back from player1's paddle.

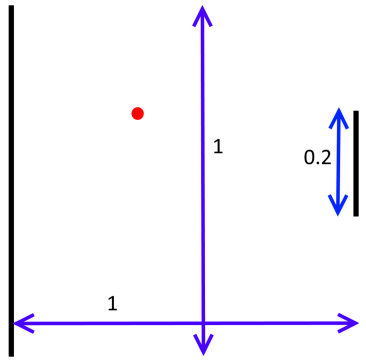


Figure 1: Pong game implementation

The game of pong is continuous. In order to achieve Q-learning implementation, the game was being distracted and transformed into a discrete, and finite state space game with following rules.

1. Treat the entire board as a 12x12 grid, which the position of the ball can only be in one of the grid at anytime. Therefore, there are total of 144 possible ball locations.

2. Discretize x-velocity of ball to +1 and -1 (+1 going right, -1 going left).

3. Descretize y-velocity of ball to +1, 0, and -1 (+1 going down, -1 going up). The y-velocity equal to 0 when $|v_y| < 0.015$

4. Descretize the paddle location into 12 locations, with location 0 when paddle on the top most position and postiion 11 when paddle on the bottom most position
   According to the discrete transformation, there are total of $(144)(2)(3)(12) + 1 = 10369$ game states. For each state, the paddle can chose the action of moveup, stay, or movedown. Therefore, a dictionary with 10369*3 = 31107 entries was implemented.

The Q-learning algorithm works by performing actions in the real word, and update the utility(Q) for each specific states according to an action-value function. The Q value for each states was being updated according to the formula below:

$$Q(s_t, a_t) < -Q(s_t, a_t) + \alpha * (r_{t+1} + \gamma * maxQ(s_{t+1}, a) - Q(s_t, a_t))$$

In the equation, $Q(s_t, a_t)$ is the Q value when at state s and taking an action a. Because Q-learning is an iterative algorithm, it requires initial Q values for the first update to occur. The initial Q values Q_0 influence the exploration rate of the algorithm, a high value of Q_0 indicate the algorithm values exploring new states and actions over the old ones, this could lead the algorithm to find a better strategy overall. As trade off, a high initial Q value will lead to a longer converge time for the algorithm to settle on a relatively good strategy. The initial Q value in this project was being set as 5 for the best performance.
α is the learning rate for the Q-learning algorithm, it influence the rate that the algorithm will learn. The α value is chosen between 0 and 1, with a low α value, it indicates that the algorithm will prefer old information more than new information, and with a 0 α value, it means that the agent will not learn anything new. With a large α value, the agent will prefer new information over old information, which might lead to difficulties on converge to a good strategy in the end. In our Q-learning implementation, α value was being set to 0.4 for best perforamnce.
r_{t+1} is the immediate reward in certain game states, The reward is positive when the agent achieved its goal, and the reward is negative when agent fails or the game terminates. In our implementation, the reward is assigned to 1 when the agent successfully rebounds the ball back, it is -1 when the agent failed to bounds the ball back and leads to the termination of the game, and 0 everywhere else. The reward evaluation was being modified in later implementation in order to achieve a better performance.
γ is the discount factor, the value of γ simply reflects how far away is the agent looking for the reward. A small γ value makes the agent "short-sighted", which means it prefers immediate rewards, and a large γ value makes the agent looking for strategies to have best rewards in the long run. In our implementation, the γ value was set to 0.8 to achieve best performance.
In our Q-learning implementation, the agent starts at the initial state, and using the Q value at the specific state with corresponding actions, it picks the action with the highest Q value and move to the next state. The Q values from the next state with the rewards from that state are being used to update the Q value from previous states. The updating function is being stated below:

Listing 1: Function to update the Q value of previous action

```
def update(self, prev_action, max_Q, reward):
    prior_Q = self.state_map[prev_action]
        if prev_action == "":
                return
        if reward == -1:
                delta = -1
        elif reward == 1:
                delta = 1
        else:
        delta = reward + self.Gamma * max_Q - prior_Q
        self.state_map[prev_action] += self.Alpha*delta
```

If the game terminates, the game will be rest to initial state, and start all over for the next learning period.

2.1.3 Results and Discussions

The Q-learning algorithm was being implemented according to the game discretize method discussed above. All Q-learning parameters, α, γ, Q_0 and learning rate decay factor C are being adjusted and studied in order to find the best Q-learning performance. With each adjustment of parameters, the agent was trained over extensive period of time (over 50k training sections) in order to show it's influence on the learning algorithm.

For detailed parameters settings and discussions, please look at the discussion paragraph below. We found out that in our best parameter setting, we could rebound the ball back for average 11 times for 1000 tests. The graph of rebound times is also shown below.

2.1.4 Learning Rate α and Decay Factor C

The first parameters that we studied is the learning rate parameter α and the learning rate decay parameter C. α is the learning rate of Q-learning algorithm which is set between 0 to 1, and it decays follows with the decay constant C as the formula $C/(C+N)$, where N is the number of visited for certain state-action pair. Different α and C values are being tested and the test results are shown below in figure. According to the figure, we conclude that with a larger α and C values, the algorithm is able to converge to a better performance after 100K game training sections. The best performance values that we chosen are $\alpha = 0.6, C = 5000$, which the learning rate will decay to half every 5000 time visit.

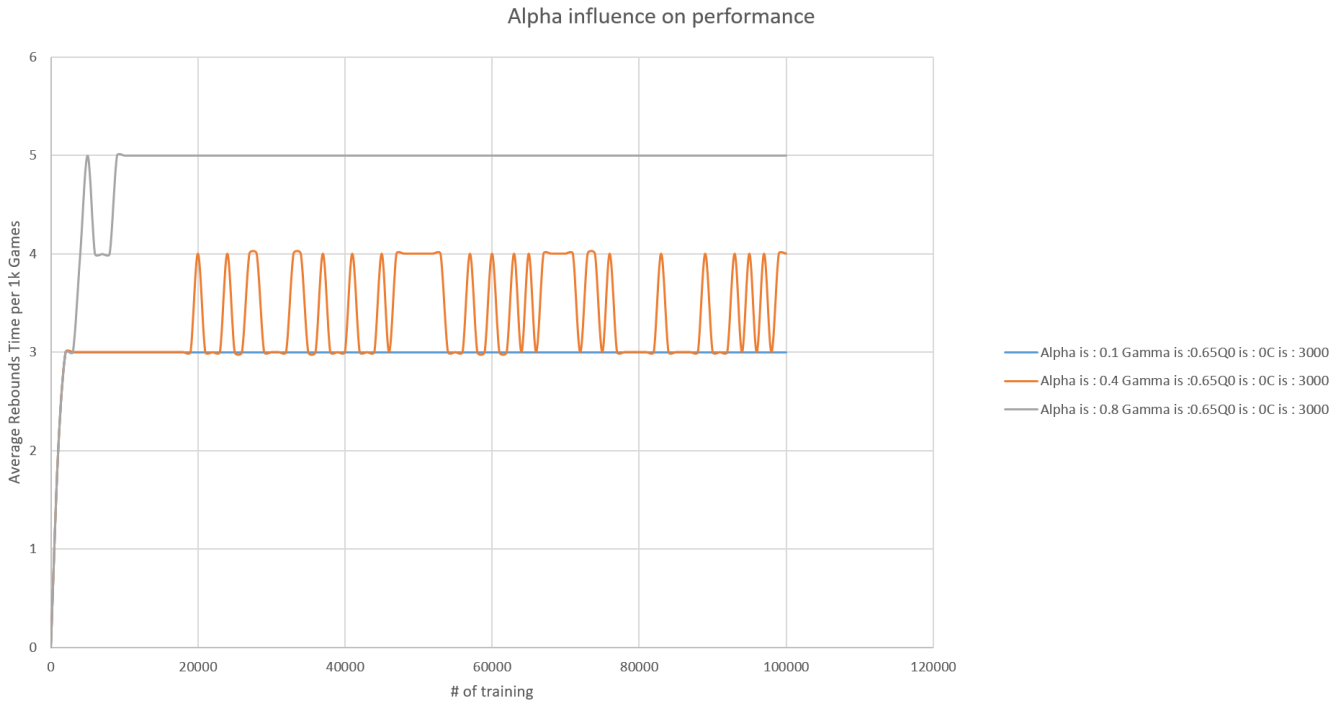


Figure 2: Learning rate Alpha influence on policy performance

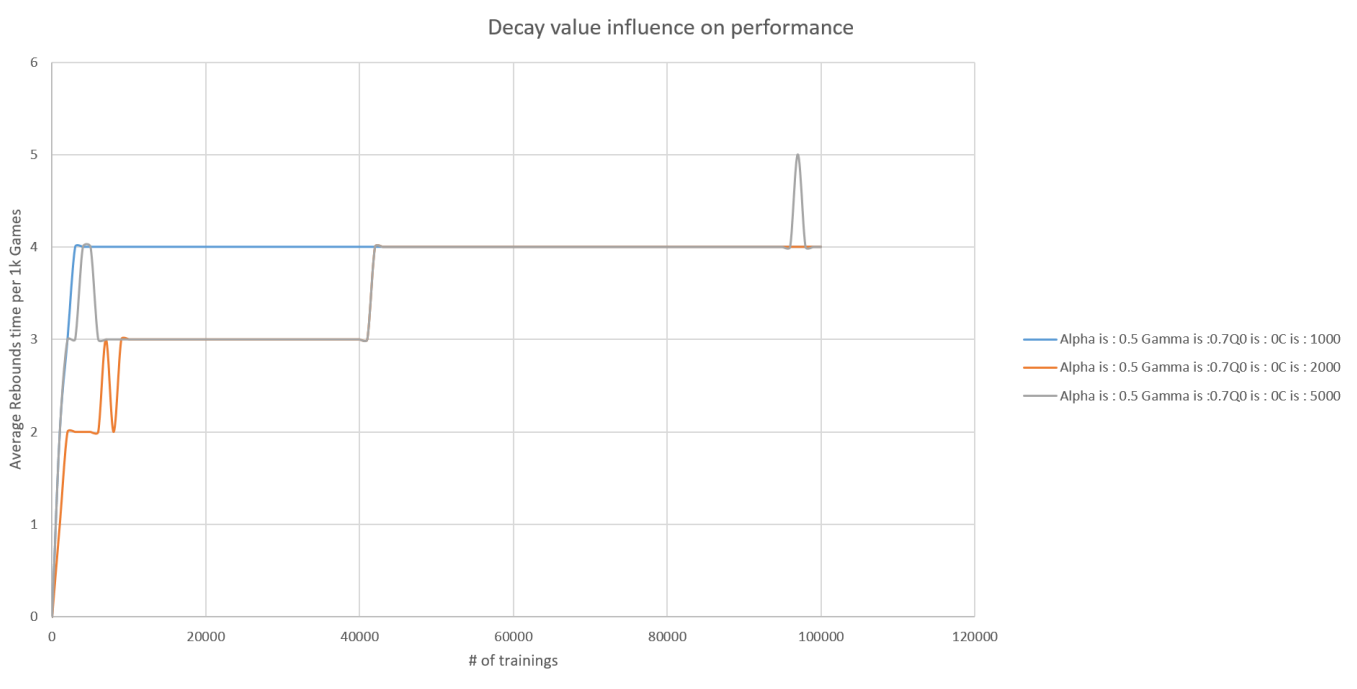


Figure 3: Learning rate Decay influence on policy performance

2.1.5 Discount Factor γ

The second parameter that we put under test is the discount factor γ. The discount factor dominates the algorithm's preference for future awards. In this implementation, we want the algorithm to have minimum influence between two rebounds. In another words we want to choose a discount factor that previous rebounds reward will have limited effect on the next rebounds. The number of game states that between two possible rebounds are around 22 states. Therefore, a discount factor that between 0.6 to 0.7 which results to a factor $\tilde{0}.0004$ after 22 states is suitable for our implementation.

2.1.6 Exploration Parameter Q_0

Because Q-learning is a iterative learning algorithm, therefore, it requires initial Q values to start the updating learning process. The initial Q values influences the program's preference on exploring new states-action pairs. In our experiment, several initial Q values were tested, and the results are shown in figure below. From the figure, we can conclude that with a larger Q_0 value, the algorithm is able to converge to a better policy in reasonable amount of time, however, as trade off, the larger the Q_0 value is, the longer it takes for the algorithm to converge. In our experiment, we were able to achieve learning strategies that converge to average rebounds of 50+ after 100k training games.

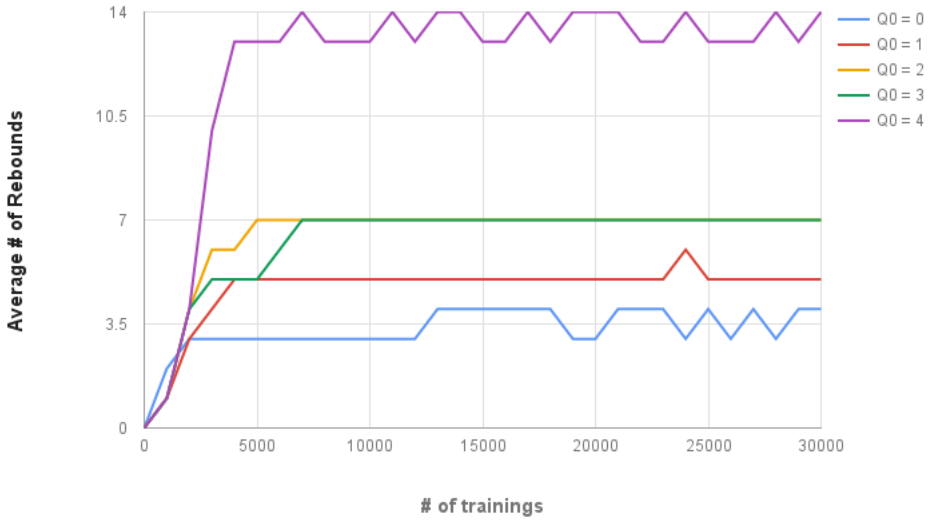


Figure 4: Initial Q value influence on learning policy

2.1.7 Discretize Parameter Adjustment

The parameters for the discretized game has also put into study. In our experiment, the size of the game was changed to 24x24 instead of 12x12 which gives a better resolution of the game. This game results to a increasing of state space from 31107 states to 124428 states. Which dramatically increased the number of states in the game. The optimal Q-learning strategy with parameters $\alpha = 0.8, C = 5000, \Gamma = 0.65, Q_0 = 0$ was applied to the new MDP and results are shown below.

Figure 5: Additional Resolution learning curve

According to the results shown, the discretize parameter adjustment with increasing the resolution of game to 24x24 did not achieve a better learning performance after 10k training games. The results with expanding the game board resolution to 24x24 shows even lower performance comparing with resolution of 12x12. This might happened because the game states dramatically increased, and the Q-learning parameters for the new game states need to be readjusted in order to achieve a good learning strategy.

After the study of different parameters and their influence on Q-learning strategy, a good Q-learning parameter set is chosen, which $\alpha = 0.8, \Gamma = 0.65, Q_0 = 0, C = 5000$. With this input parameters, the learning curve of the algorithm is shown below, which after 11k training games, the algorithm is able to converge to a policy with average rebounds time of 14 over 1k games. So the result is better than 12*12 discretize states setting, which is an improvement(14 vs. 11) based on the discretize method of Q learning. The reason is that the states are divided more precisely and the agent needs to learn from more states and move faster to get the desired state in order to bounce the ball back. So this is the reason why we choose to discretize it to 24*24 and add velocity to the agent in order to acquire better performance.

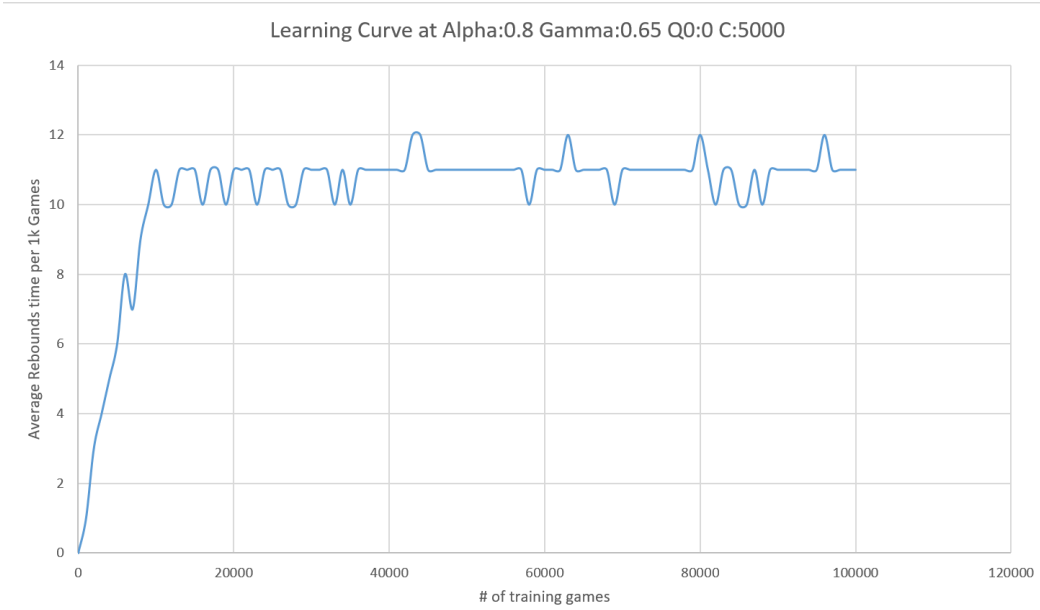


Figure 6: Good learning strategy learning curve(rebound avg = 11)

2.2 Part 1.2: Two Player Pong

2.2.1 MDP modifications

In order to achieve a better learning strategy, the MDP of Q-learning algorithm was being modified as following.

1. The states that represent the game was being modified in order to achieve a better understanding of the game board. Because the velocity of the ball was randomized every time it hits the paddle, the randomize in velocity makes the agent a lot harder to determine a good policy from reinforcement learning. Therefore, two more state factors was added into the game state transformation. The x-velocity of ball is modified into four speed states, which when $v_x > 0$ and $v_x < 0.03$, the value is set to $+1$, and when $v_x > 0.03$, the value is set to $+2$. Similarly when $x_x < 0$ and $v_x > -0.03$, the value is set to -1, and when $v_x < -0.03$, the value is set to -2. The new transformation addes two additional state options which leads to a total of 62,208 states in the game.

2. The reward evaluation function was modified in order to achieve a better reward strategy. The new reward strategy optimize the agent to use the center of the paddel to bounds the ball back, and trying to avoid the ball to fall off too far away the paddle. The algorithm follows as the code shown below:

Listing 2: The smart update function – gives the value according to the failed and success position

```
def smart_update(self, prev_action, max_Q, reward, ballY, paddley):
        if prev_action == "":
                return
        priorQ = self.state_map[prev_action]
        if reward == -1: # case missed
                delta = -1 * abs(ballY - paddley - 60) * 1/100
        elif reward == 1:
                delta = 2 - (abs(ballY - paddley - 60) * 1/60)
        else:
```

```
                delta = (reward + self.Gamma * max_Q - priorQ)
        self.state_map[prev_action] += self.Alpha*delta
        return
```

The algorithm assigns rewards to be -1 when the ball falls farthest away from the paddle, and -0.1 when the ball only barely falls off the paddle. Similarly the algorithm rewards +1 when the ball was rebounds back using the center of the paddle, and +0.1 when the agent uses the edge of the paddle to rebounds the ball. The additional game states increased the overall number of states that the algorithm need to learn on, the training time increased. However, comparing with the increasing game states from previous section, the adjustment of increasing game states with better resolution of x velocity shows a significant improvement in the game learning strategy. With the new award function, the algorithm did not shown significant improvement after 100k training games, which might due to the new parameters should be readjust in order to match to the new reward strategies.

2.2.2 Hardcode AI implementations

A new AI with hardcoded algorithms was implemented and added in the game. The algorithm simply matches it's position with the ball's position, and moves with half the speed of Q-learning AI. The implementation code is shown below:

Listing 3: Hardcode AI implementation

```
def hardcodeAI(ball, ballDirY, paddle1):
    # Case ball on top of paddle
    if ball.y < paddle1.bottom - (PADDLE1SIZE)/2:
        paddle1.y -= 0.02*SCALE
    # Case ball on bottom of paddle
        elif ball.y > paddle1.bottom - (PADDLE1SIZE)/2:
        paddle1.y += 0.02*SCALE
    return paddle1
```

2.2.3 Results and Discussions

Comparing with the MDP modification in part 1, in part 2, the MDP states are modified with additional x velocity parameters. Which 2 additional x velocity parameters are added into the state discretization. The algorithm was put under training with the previous good learning parameters which are $\alpha = 0.8, \gamma = 0.65, C = 500, Q_0 = 0$, after 100k training games, the algorithm was able to converge into a better game policy. The learning curve is shown in the graph below:

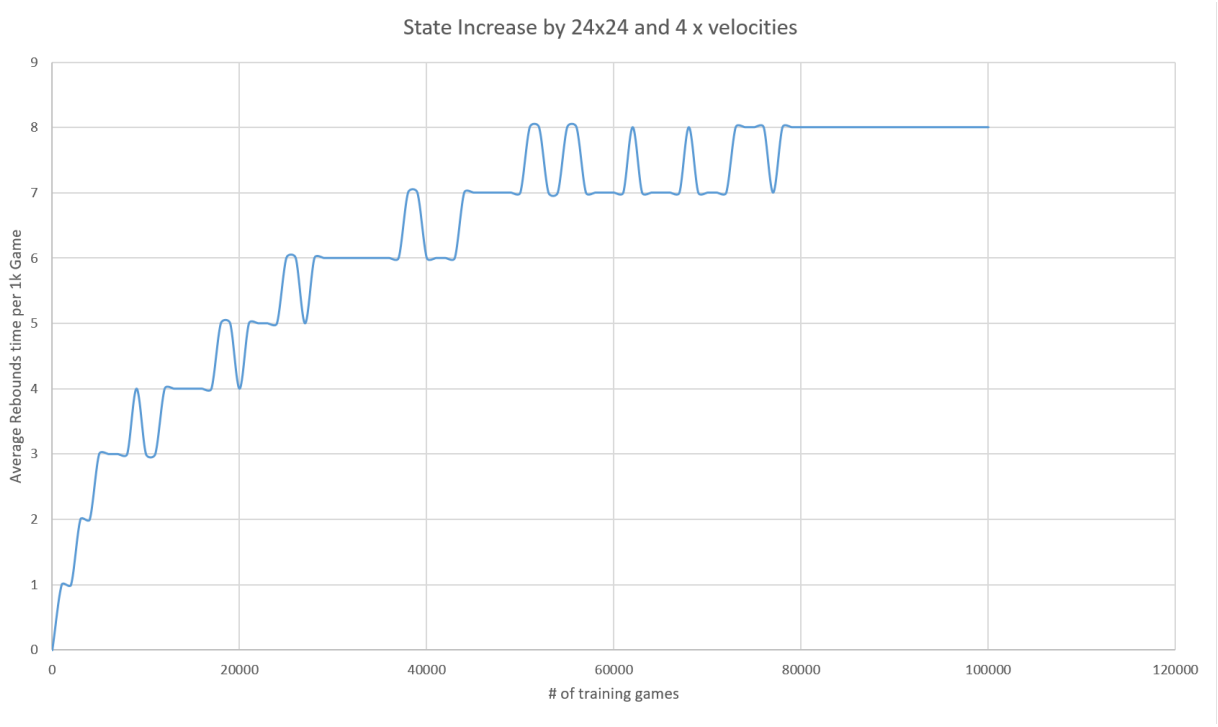


Figure 7: Learning Curve with MOP modification on resolution and speed

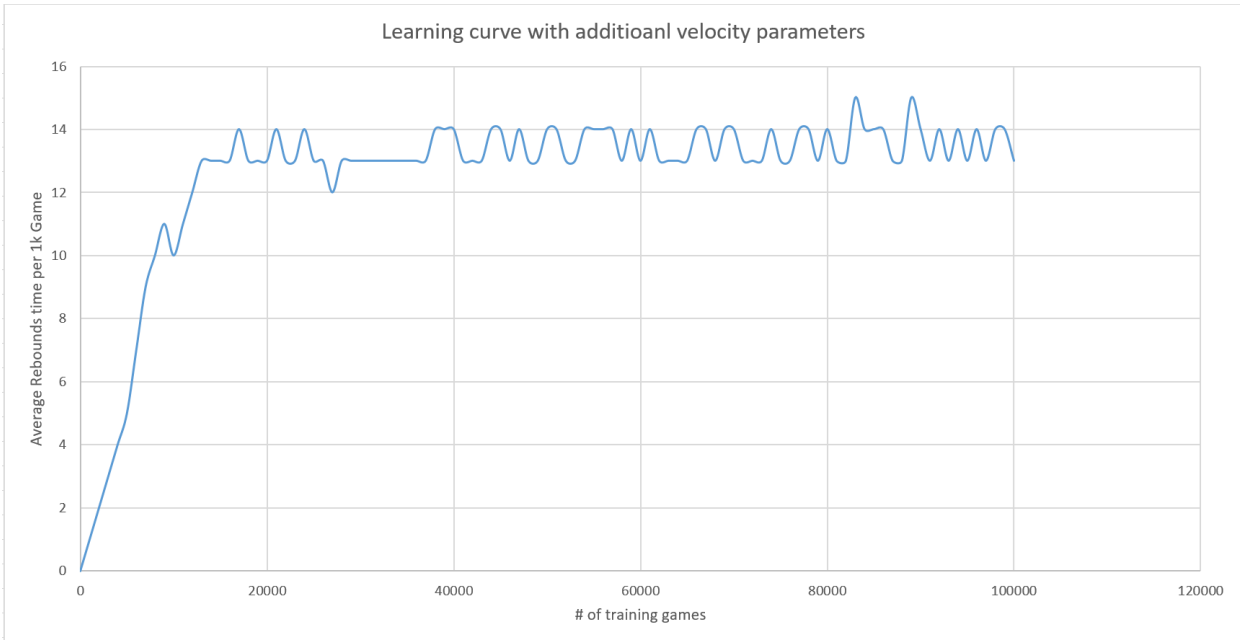


Figure 8: Learning Curve with additional x velocity parameter

With the Q-learning AI versus hardcoded AI, the Q-learning parameters are chosen in order to have the best performance. The Q-learning parameters are $\alpha = 0.8, C = 5000, \Gamma = 0.65, Q_0 = 5$ After 16k game trainings, the algorithm converged to 33 average rebounds time. Than it is played against the hardcoded AI for 1000 games, and the win rate is being calculated as 86.3%.

2.3 Part 1 Extra Credits

2.3.1 GUI Animation and Game functionalities

The game was animated using GUI, which has player control and AI control. For game animation presentation, please check attached video file.

2.3.2 Player vs AI

Player versus AI was implemented, which the player controls the paddle on the left, and AI controls the paddle on the right. The player uses the mouse location to control the ball. The control is slightly harder, therefore, in certain situations, the player will lose to the AI. (When ball is hitting the corner, or ball's speed increases dramatically) However, the player's paddle moves a lot smoother than the AI paddle.

3 Part 2.1: Digit Classification using Discriminative Machine Learning – Perceptrons

3.1 Perceptrons Background

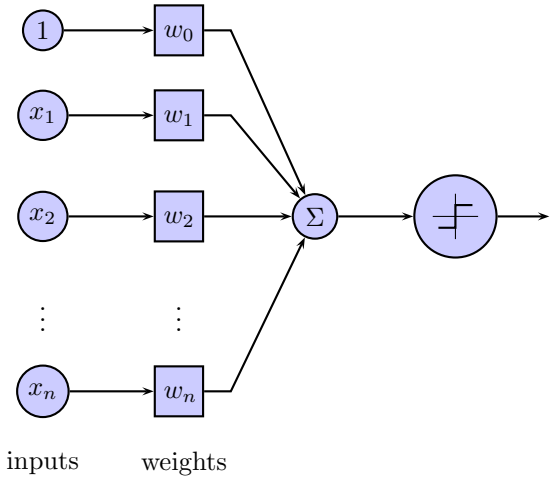


Figure 9: Perceptron Architecture (biased version)

Perceptrons are used for building linear classifiers to be used in binary classification. A perceptron can be described by the model shown in Figure 9. On the left are the input components x_1 to x_n and a base factor 1. To the right of these inputs are corresponding weight components w_0 to w_n. Thus, the perceptron can be characterized by the weight vector of length $N + 1$ where N is the number of dimensions of the feature space. When a sample x needs to be characterized, the sign of the dot product between $< 1, x_1, ..., x_n >$ and $< w_0, ..., w_n >$ will classify the data as 1 or 0.

Training the perceptron proceeds in the following manner:

1. First initialize the weight vector
2. Repeatedly cycle through training data samples..
3. For each sample obtain classification y
4. if $y \neq y'$ where y' is true class label
5. update weights in opposite direction of error

The perceptron is a binary classifier but it can be extended to multi-class classification by using techniques like "One vs All" [7]. In OVA a separate perceptron is trained for each class label [7]. Each perceptron P_c classifies a label as either class c or other-than c. By using the decision rule $c = argmax_c W_c \dot{X}$ to classify sample X we will pick the class whose perceptron considered X to be most distinct from the rest of the classes.

3.2 Tune on Parameters

As specified in the implementation of perceptrons, we need to tune on the following five parameters to figure out which combination performs better:

- Learning rate decay function;
- Bias vs. no bias;
- Initialization of weights (zeros vs. random);
- Ordering of training examples (fixed vs. random);
- Number of epochs.

So, we have tried to tune all of the above parameters and found out that there exists some trend about these parameters.

3.2.1 Learning rate decay function

First, for the learning rate decay function. The mathematical expression of the learning rate α should be:

$$\alpha = \frac{c}{c+t} \tag{1}$$

Where c stands for the decay constant, t stands for the current number of epoch. So we could see that α will change quickly from 1 to a smaller value if c is smaller. (e.g. $c = 1, t = 1$. Then α is changed from $\frac{1}{1+0}$ to $\frac{1}{1+1} = \frac{1}{2}$) So it indicates we will update the weight quickly at the beginning of our total epochs as the learning rate decreases faster. And the learning rate will gradually converge to 0 if t increases to a big number. So this result means that the perceptron will execute a small learning rate through lots of epochs and it is highly possible that it couldn't learn something from the training. That's should be a bad choice for the perceptron training. We tested the decay constant: $c = \{1, 10, 100, 1000, 10000\}$ with other fixed parameters. And we found that we only get 80.5% overall accuracy if we choose $c = 1$, and we found that we could get better accuracy when $c = 1000$ compared with other constant values of decay rate.

3.2.2 Bias vs. no bias

For the implementation of the bias, we add 1 bias at the initialization of features and the initialization of weights. So, we totally have $28 * 28 + 1 = 785$ features to train. Intuitively speaking, we should add the bias because the bias allow the shift of the activation function in perceptron and gives more accurate result in the classification. We tested the perceptron classifier with bias and without bias cases. Here is the result:
(epoch = 100, decay = 1000, random order of training = false, random initialization of weights = false)

```
HINT: The accuracy scale is (0, 1)
Accuracy(Bias): 0.82
Accuracy(No bias): 0.803
```

So we should include bias in the perceptron.

3.2.3 Initialization of weights (zeros vs. random)

We tested on the initialization of weights by all zeros and random numbers from $(0, 1)$, the result shows that the random initialization of weights could give better accuracy than all zeros initialization. The result is:
(epoch = 100, decay = 1000, random order of training = true, random initialization of weights = false/true)

```
Accuracy(Zeros): 0.829
Accuracy(Random): 0.834
```

So we could see that the performance of random initialization of weights is better.

3.2.4 Ordering of training examples (fixed vs. random)

We tested on the two choices with other fixed parameters. And we found out that the random order of training is better. Because for each epoch, the perceptron classifier would receive different training cases. The reason is that we would give random order of training data at each epoch. So it could make the adjustment of weights more accurate to the true weights of each digit label. Instead, if we keep fixed order for each epoch, the classifier would repeat the same update for each training data, which is not ideal for the update of weights. Here is the result:
(epoch = 100, decay = 1000, random order of training = true/false, random initialization of weights = true)

```
Accuracy(Fixed order): 0.814
Accuracy(Random order): 0.834
```

3.2.5 Number of epochs

We tested several cases of the number of epochs n. And we found out that the number of epochs should be appropriate with the number of decay in order to get the better overall accuracy. And the better epoch shouldn't be small because we want the training curve to converge when n is large enough. Ideally, the curve should converge to 1, which means we could 100% classify the training data after receiving n training epoch. Therefore, we found $n = 100$ could give us better result. Because we noticed that when the training curve will converge when n is around 100. And there is not necessary when the curve converges to some value, so it means we should stop at some n instead of continuing. Here is a comparison:

```
Accuracy(n=1): 0.761
Accuracy(n=10): 0.808
Accuracy(n=100): 0.834
Accuracy(n=500): 0.825
```

3.3 Results

In sum, we found out that the following combination could give us better result:

- Decay = 1000
- Bias = 1
- Initialization of weights = random
- Ordering of training examples = random
- Epoch = 100

Table 1: Summary Results

Accuracy	0.834
Execution Time	34.973 s

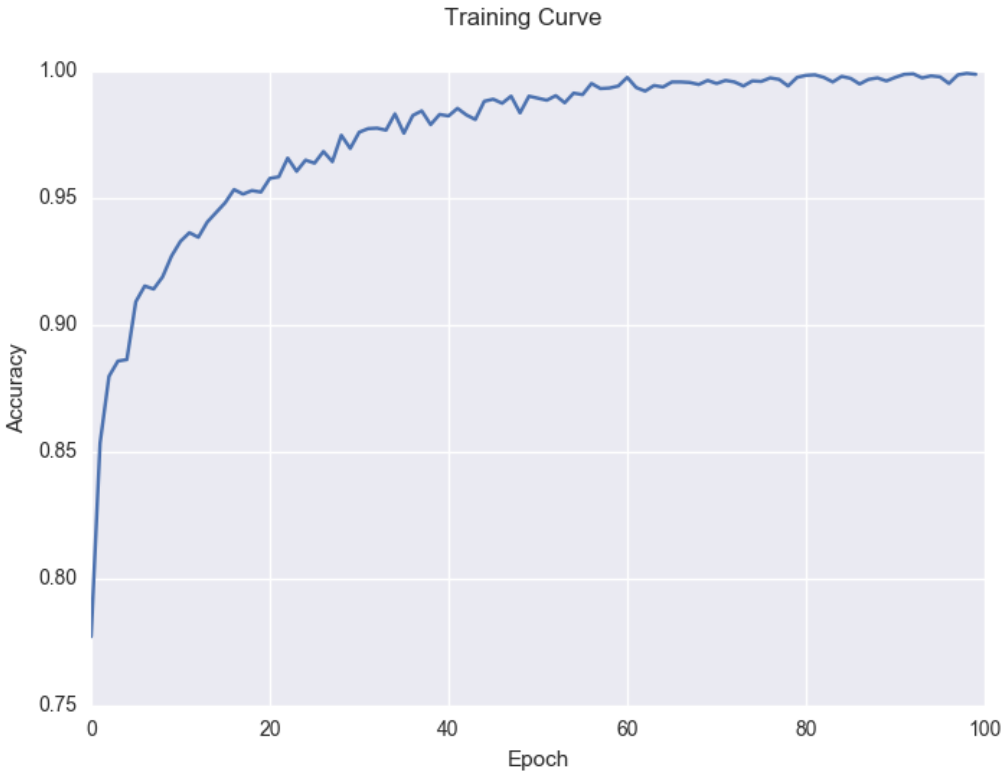


Figure 10: Perceptron Training Curve

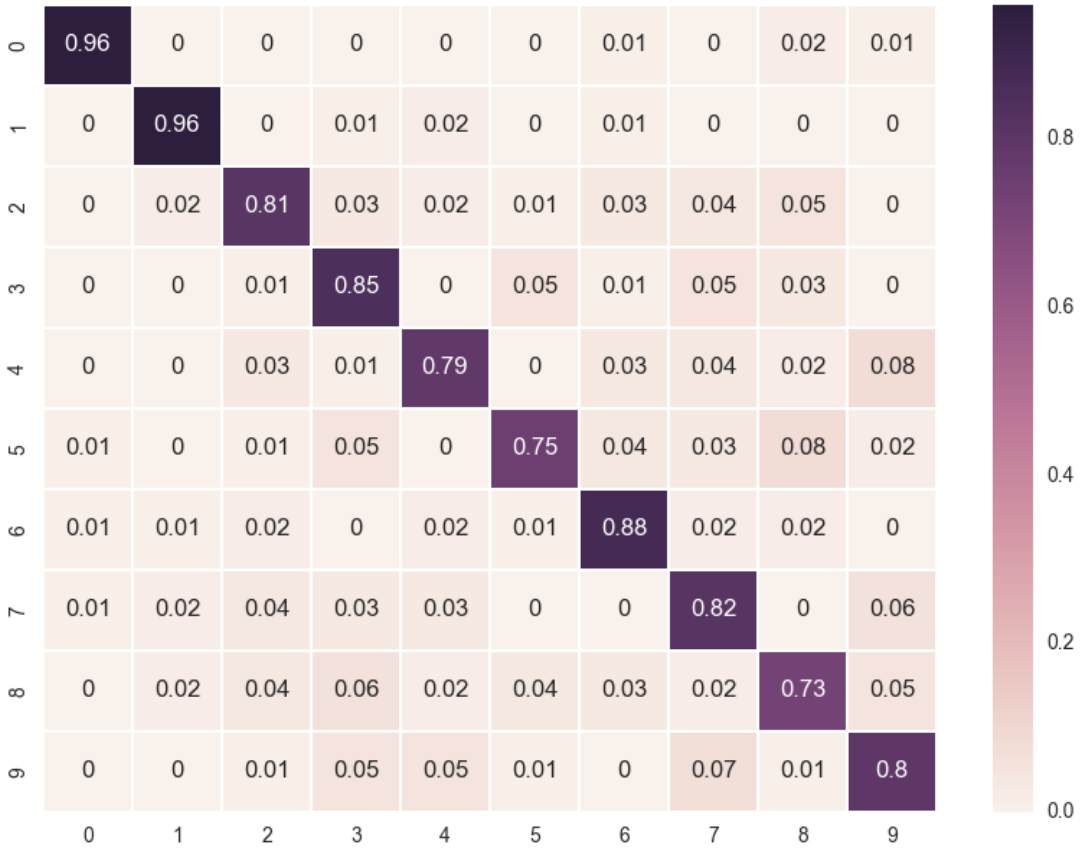


Figure 11: Perceptron Confusion Matrix

3.4 Discussion: Comparisons with Other Classifiers

We found that our Perceptron classifier was less accurate than our best naive bayes model – but it took two orders of magnitude less time to train (30 seconds vs 2 hours!). What was interesting is our study of the weight visualization. There we can see which pixels were given the largest weights. The strength of the perceptron was that some pixels pushed the score down. This allowed the perceptron with unmodified feature sets to achieve 83.4% accuracy wheras the plain naive bayes could only achieve 77% accuracy. The reason for this is that the perceptron learns both which pixels support our confidence in the classification and which weaken it.

We will speak more about the weights visualization in the extra credit section.

4 Part 2.2: Digit Classification using Discriminative Machine Learning – Nearest Neighbor

4.1 Background

It is possible to do digit classification without building parametrized models of the digits. This means we do not need to find features of the digits that can distinguish the digits. Instead, we define a feature space and project our training samples onto it. To classify a test sample first we project a test sample onto the same

feature space. Then we find the nearest K neighbors according to some distance measurement (for example, Manhattan or Euclidean distance). We then assign the test sample the same label as the majority of the nearby neighbors. This is visualized in Figure 12 [6].

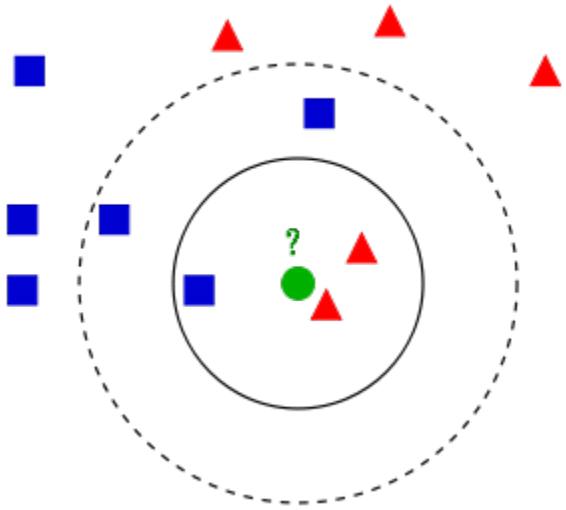


Figure 12: K Nearest Neighbor [6]

The test sample would be classified as belonging to the class of the red triangles for K=3. This is because two out of the three nearest neighbors – contained in the black circle – belong to the triangle class. If K was increased to five, then the nearest neighbors will be contained in the dashed circle. And in that case three out of the five belong to the blue square class – flipping the classification to the blue square class instead.

4.2 Similarity measure

There are some distance measure for this k-NN classification problem. As we want to implement a categorical classification, the first choice should be Hamming distance. But there are still other useful similarity measure for digit pixel image classification. We used the following similarity measure:

- Hamming distance;
- Dot product;
- Euclidean distance;
- Cosine distance;

From the test of different similarity measure, we found that **dot product** could give us better result although Hamming distance could also give similar overall accuracy. Besides, cosine distance gives the worst overall accuracy. It maybe the reason that cosine distance is applied on the text mining about the similarity of two documents, it would count the number of words in the documents as the similarity measure. So it is not helpful for this task.

For Hamming distance, the idea is that we need to calculate the number of pairs (w_i, x_j) is in different, which means $w_i \neq x_j$, where w_i is the train data, x_j is the test data. Therefore, the smaller the number of hamming distance is, the closer the test data to this kind of train data. Then we need to select the closest k train data and then collect the labels of them for the vote process.

For the euclidean distance, the calculation is similar with the Hamming distance. The only difference is that we calculate the square distance of the data points.

Here is a table about the performance of each distance measure:

Similarity	K	Overall Accuracy	Running Time
Dot product	4	0.910	14.943582s
Dot product	5	0.899	15.022628s
Hamming distance	4	0.908	552.051808s
Hamming distance	5	0.9	553.966606s
Euclidean distance	4	0.9	583.022539s
Cosine distance	4	0.051	605.6s

4.3 Choice of similarity measure

Finally, we chose the dot product as our similarity measure. To implement dot product, we need to do some prepossessing first. The principle of dot product is that we view the sequence of pixel values as a vector. And then we could apply dot product on the training data and test data. For example:

$$sum = [A_0, A_1, ..., A_n] \cdot [B_0, B_1, ..., B_n] \tag{2}$$

Where A_i denotes for the pixel value of train data, B_j denotes for the pixel value of test data. So the total amount of elements in the vector is $n = 28 * 28 + 1 = 785$

The original pixel value is: 0 for background, and 1 for foreground. And we need to make some adjustment here. Because we want to measure the similarity between two images. It is actually a logic XNOR expression of this idea:

A	B	Similarity
0	0	1
0	1	0
1	0	0
1	1	1

But $0 \cdot 1 = 0, 0 \cdot 0 = 0$ in the math calculation, so we need to change the representation of the background in order to implement dot product idea. Our solution is that we changed the value 0 to -1 to represent the background value of pixel. In this way, we get the following table:

A	B	Similarity
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

So, when the pixel value of the test image is the same as that of train data, we add 1; otherwise, we add -1 to the final result. Thus, the higher the final sum of the dot product, the higher similarity we get for the test image and train image. This is the idea of how to implement dot product.

4.4 Confusion matrix

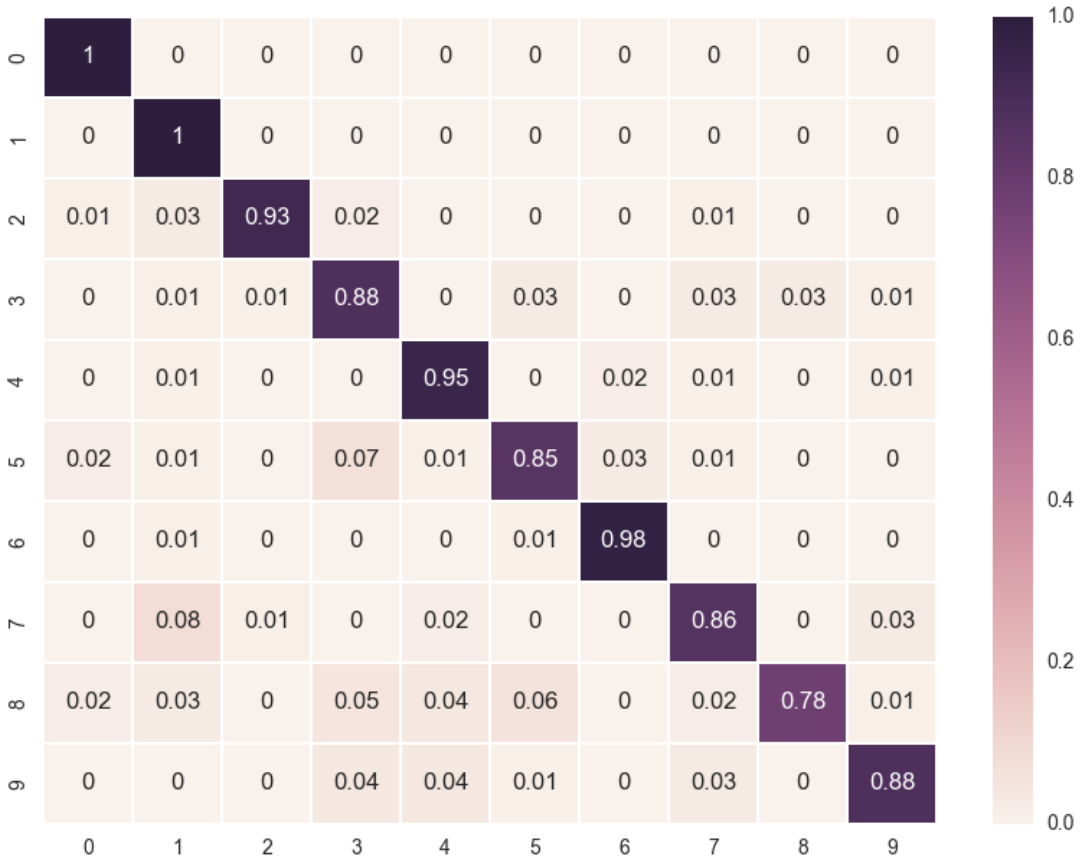


Figure 13: KNN Confusion Matrix (K=4, overall accuracy = 0.91)

4.5 Impact of Neighbor Size on Running Time and Accuracy

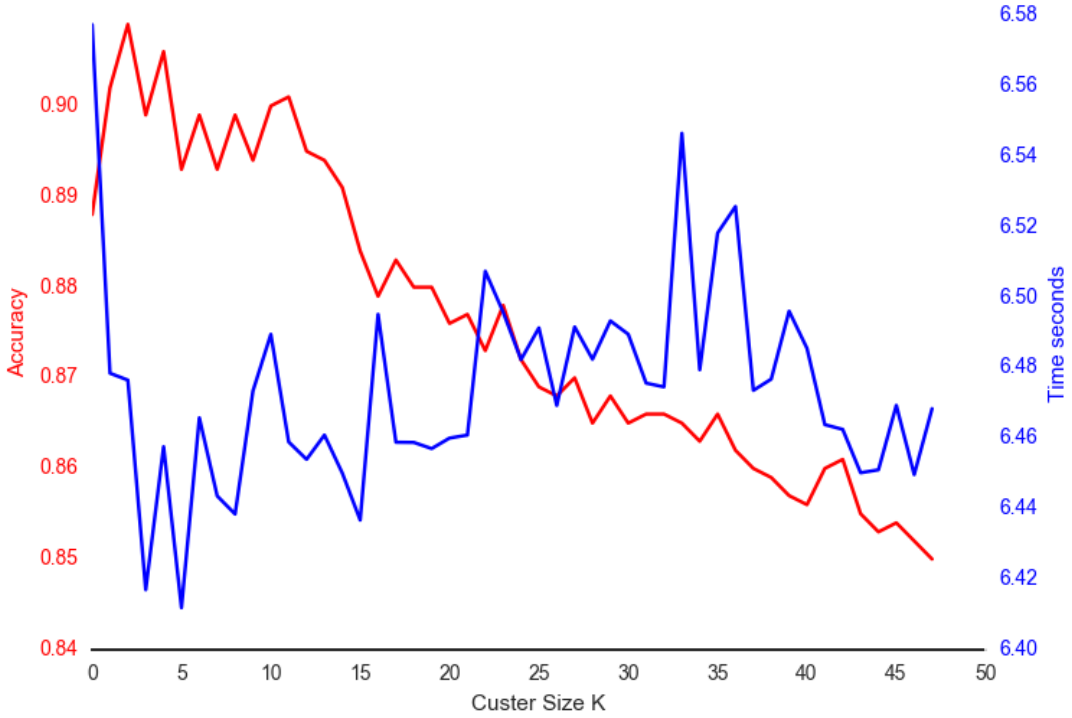


Figure 14: Accuracy and Running Time vs K

The clustering factoring K can impact the accuracy and the running time of the algorithm. Figure 14 shows the relationship between running time and accuracy and the clustering factor. To plot this relationship we took the average of 10 runs for each K, with 5 runs for warmup.

As shown in the figure, the best accuracy we found was for K=4. After this more and more errors started creeping in due to pulling in too many of the neighboring class nodes.

The running time seems to imply a possible linear relationship between running time and nearest neighbor size. This is to be expected because we have to sort the nodes in terms of distance from the node being tested. The slope is very small however, likely because our population size is not that large.

4.6 KNN vs Perceptron/Navie Bayes

In terms of accuracy we found the K-NN to be very strong. The highest accuracy we achieved was 90.5% and we were able to classify 5 digits with greater than 93% recall (as shown in Figure 13.

As shown in Table 1, this technique is about 10% more accuracy than using a perceptron based classifier. In our last MP we implemented a Naive Bayes based classifier. For the best configuration we found, we were able to achieve accuracies of 89.5% – but training that model took over 2 hours! Furthermore, it was very difficult to keep the naive bayes model from confusing 7 and 9 – because they are very similar. But NN does not see mto have a problem with such cases.

The biggest downside is that to apply K-NN we have to keep the entire training data-set on hand. This can make this infeasible for large datasets.

Here is a table for the comparison:

Method	Overall Accuracy
Navie Bayes	0.77
Navie Bayes(Group feature)	0.895
Perceptron	0.834
KNN	0.908

5 Extra Credit

5.1 Advanced features for perceptron

We have tested on the ternary feature for the digit image as specified in Assignment 3. The representation of ternary feature is converted to integer value as the following table shows:

Symbol	Value
'+'	1
' '	0
'#'	2

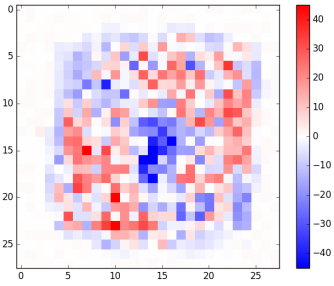
For the same setting as described in part 2.1, we tested the ternary feature on 100 epochs, the overall accuracy is 0.819, which is not better than binary feature case (accuracy = 0.834). However, we found out that the training curve didn't converge to 1 when using 100 epochs. Because we use ternary feature values, so it is normal that we need more epochs to train the classifier to converge. So, we found out that around it converges at around 120 epochs. Here is the result:

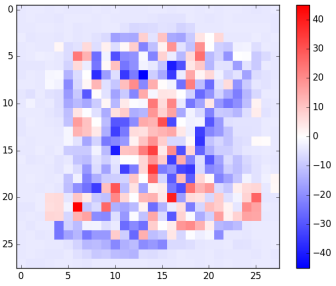
```
Ternary Feature:
Accuracy(epoch  = 120): 0.829
Accuracy(epoch = 100): 0.819

Binary Feature:
Accuracy(epoch = 120): 0.825
Accuracy(epoch = 100): 0.834
```

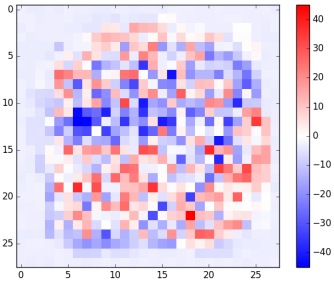
We also tried other setting of the values to see whether the performance could be better or not. But the result shows that there is not much difference among different values for the features. Therefore, for ternary feature case, if we use the same epoch (epoch = 120 in this case), ternary feature could improve just a little (0.829 vs 0.825) for the performance of the perceptron. However, if we compare the best accuracy of them, the best overall accuracy of ternary feature is not as good as that of the binary feature as specified.

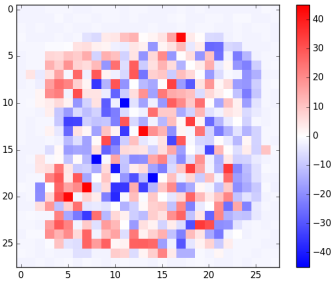
5.2 Visualization of weights

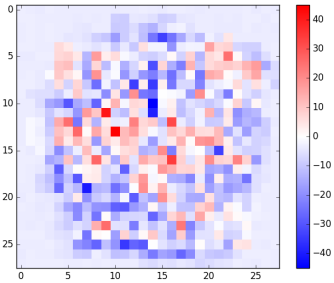
After the training of perceptron, we also plot the graph of weights to figure out the distributions of the discriminative weights of each digit label. These results are shown in Figure 15.

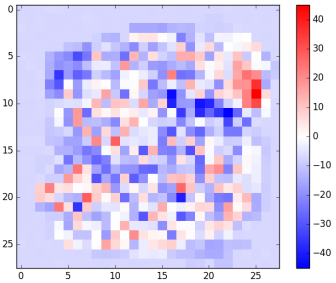
0: 

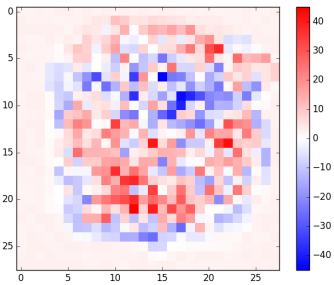
1: 

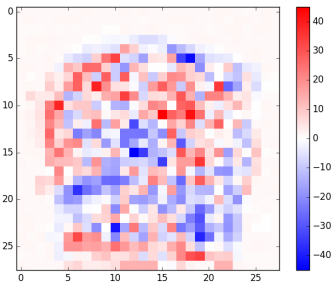
2: 

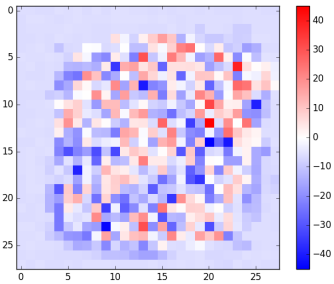
3: 

4: 

5: 

6: 

7: 

8: 

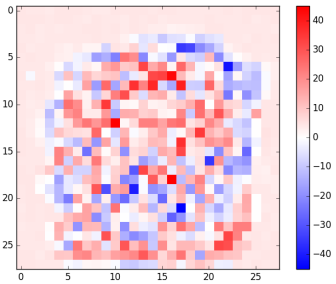
9: 

Figure 15: Perceptron Weights Visualization Digits 0-9

We found that the cell in red color indicates the the most important (discriminative) pixel feature for each digit. Because the value is much higher than others (40, 50...), which means we always add value to this individual value of weight once we misclassified the digit with others. So it is the most discriminative individual pixel for this digit class. In the contrary, the blue cell indicates that the most confused pixel for this digit and other digits. Because we will deduce the weight of the pixel if we misclassified this digit with other digit. So, the smaller the value of weight (e.g. -40, - 45), the more likely this pixel would be confused by the perceptron. It is reasonable because this phenomenon implies this individual pixel is less discriminative for the digit. And we always deduce the value of weight in this pixel if we misclassified the digit with others. So, that's the reason of why we have two different signs of values here. The sign of weights tell us the discriminative degree of the pixel for each digit weights.

5.3 Differentiable perceptron vs. Non-differentiable perceptron

For the differentiable perceptron, the update rule is different with the rule for the non-differentiable. For non-differentiable update rule, we only update the weights when there is a misclassification. The formula is: $w_i = w_i + \alpha * x$ and $w_j = w_j - \alpha * x$. But for the differentiable perceptron, we adopt a soft update for each weight. It depends on a sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

So the update for the weight becomes:

$$w = w + \alpha \cdot f(w \cdot x) \cdot (1 - f(w \cdot x)) \cdot x \tag{4}$$

So we also implemented differentiable perceptron according to the mentioned idea. And the result doesn't show a big difference compared with the non-differentiable. The overall accuracy is in this range (81%, 82%) after several tests on the differentiable perceptron. The result is shown as below:

```
Accuracy(non-differentiable) = 0.834
Accuracy(differentiable) = 0.82
```

The reason could be that we are dealing with linearly separable data instead of non-linear separable data. And differentiable could behave better over non-linear case compared with non-differentiable. Therefore, for this kind of pixel data, differentiable is not guaranteed to perform better compared with non-differentiable. In a nutshell, they have similar overall accuracy on this dataset.

5.4 Performance of other classifiers

We compared our results with other classifiers. We leveraged the widely used SciKit-Learn python package for the implementations of classifiers absed on the following techniques:

1. Linear and Radial based Support Vector Machine
2. Gaussian Process
3. Decision Tree
4. Random Forest
5. Multi Layer Perceptron Network (Hidden Layer = 100)
6. AdaBoost
7. Naive Bayes (From prior MP)
8. Quadratic Discriminant Anlaysis

The results are shown in Table 2.

The interesting thing here is that the classifiers which did best are two that we covered in class. Linear Function based Support Vector Machine and the Multi Layer Perceptron. The Multi Layer Perceptron performs significantly better than our single layer one. It achieves the best accuracy at 94.1%. The closest second is our Linear Support Vector Machine. The best performing classifier from our last MP had 89.5% accuracy.

All these results were based on using default configuration classifiers from SciKit-Learn kit [3].

Table 2: Comparison With Other Classifiers

Classifier	**Accuracy**	**Time (ms)**
Linear SVM	0.924	3364
RBF SVM	0.1155	15455
Gausian Process	0.1335	1157707
Decision tree	0.687	78
Random Forest	0.648	37
Neural Network	0.941	3826
AdaBoost	0.516	1149
QDA	0.263	647

6 Work Distribution

6.1 Joint Efforts

- Report

6.2 Yasser Shalabi

- Comparison to Classifiers EC
- Support (graphing, verification, some of results collection)

6.3 Chongxin Luo

- Part 1
- Part 1 EC

6.4 Haoran Wang

- Part 2
- Part 2 EC

References

[1] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[2] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[5] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[6] Wikipedia. K-nearest neighbors algorithm — wikipedia, the free encyclopedia, 2016. [Online; accessed 28-November-2016].

[7] Wikipedia. Multiclass classification — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-November-2016].

[8] Wikipedia. Perceptron — wikipedia, the free encyclopedia, 2016. [Online; accessed 28-November-2016].