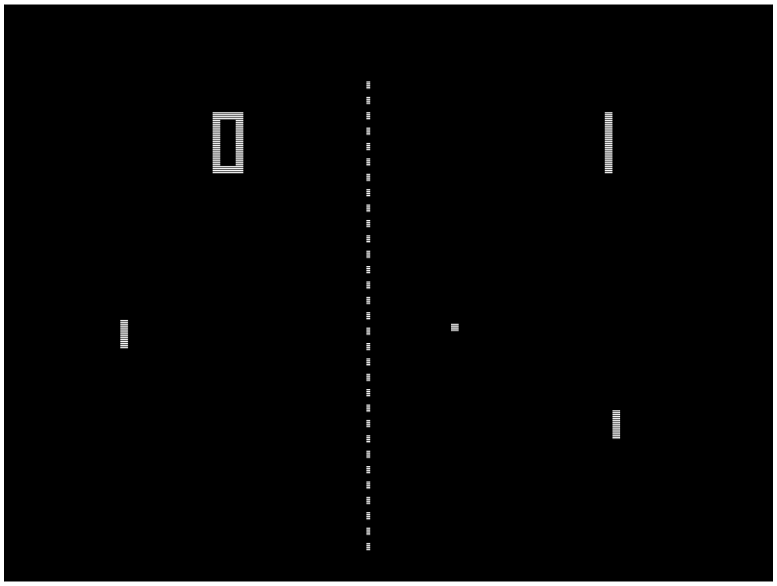


ASSIGNMENT 4

REINFORCEMENT LEARNING AND PERCEPTRONS – 4 CREDIT

Q-Learning Pong Player
& Digit Classification

ABSTRACT

In the first part of this assignment, Q-Learning is implemented for a pong game AI player. The learning parameters are explored and discussed. The trained AI player performs significantly better than untrained AI, hard-coded auto-player and human player. Interactive GUI animations are shown for better visualization. In the second part, Perceptron and K-nearest neighbor methods are applied in digit classification problem. Parameters are discussed and the classification accuracy is compared with Naïve Bayes classifier.

Qixin Zhu

CS 440 / ECE 448

Table of Contents

1-Q-Learning Pong Player

1.1 Single-Player Pong

1.1.1 Implementation

In the defined Markov Decision Process (MDP), a state has 5 features: ball's X location, ball's Y location, ball's X velocity, ball's Y velocity and the paddle's location. The corresponding Q-state of this pong game will have 1 extra dimension as the paddle's action – either move up or down. To speed up the implementation, a 7-dimension matrix is used to store all the above information. The 1 extra dimension in the matrix is used to store the reward of a given state and the number of trials of every action.

1.1.2 Parameters

Learning rate $\alpha = \frac{30}{30 + \mathrm{N(s,a)}}$ in which N(s,a) is the number of times the action ***a*** has been tried at state ***s***. 30 is a constant found through trials, whose physical meaning is how fast the learning rate drops to below 50% (when N(s,a) = 30).

Discount factor $\gamma = 0.8$. The original math was: assuming, at a hard scenario, the ball's speed reaches 0.1, so it takes 20 steps to bounce. 0.8^20 is about 1%, low enough to forget the reward from the last bounce. Through later trials, no better performance can be reached with either a higher or a lower discount factor.

Exploration function $= Q(s,a) + \frac{1}{0.01+N(s,a)}$. 0.01 is there to avoid "divided by 0" problem. Thus, seeing action ***a*** at state ***s*** 0 times results in a definite trial. **The exploration is then turned off at testing phase**, which improves the performance by choosing the best strategy found so far.

1.1.3 Game Results

Using the parameters found in section 1.2, the AI player can **rebound the ball 13.88 times** on average of 5000 test games, after playing **100,000 training games**, which only takes 8847ms. The performance improvement of the Q-Learning AI player can be observed in the following 3 animations (click the link for YouTube videos):

- Part_1_1_10 (https://youtu.be/0_cvhSdY3yg): Single Q-Learning player playing against the wall after 10 trainings
- Part_1_1_1K (https://youtu.be/sIIu_oALTPE): Single Q-Learning player playing against the wall after 1000 trainings
- Part_1_1_100K (https://youtu.be/0V17bNkKScY): Single Q-Learning player playing against the wall after 100K trainings

After 10 trainings, the AI player can mostly catch the first bounce the ball. However, almost all states are never seen to the AI, so it chooses to explore all possible actions by taking the "move-up" option before "move-down", resulting in wandering around the top of the game board.

After about 1000 trainings, the AI player is exploring the “move-down” option for most of the unseen states, resulting in wandering around the bottom of the game board. It is also able to rebound more balls.

After 100,000 trainings, the AI player has seen enough states and tried reasonable amount of exploration. Exploitation is taking over the action choice, so the AI acts optimally according to its knowledge. It is able to rebound most of the balls only except when the ball speeds up too high to give enough time for paddle movement. **The average number of bounces over 5000 test games is 13.88.**

1.1.4 Adjust Discretization Settings

Discretization settings are adjusted through trails, but no significant performance improvement is observed. The largest state space is taken to have**: a 20 x 20 grid game board, 4 ranges of the ball’s X velocity** ({ (-∞, -0.06), [-0.06, 0), [0, 0.06), [0.06, ∞) }), **5 range2 of the ball’s Y velocity** ({ (-∞, -0.06), [-0.06, -0.015), [-0.015, 0.015), [0.015, 0.06), [0.06, ∞) }) **and 20 paddle locations**. The size of this state space is 20^2*4*5*20 = 160,000, which is roughly 16 times of the settings in section 1.1.3 (12^2*2*3*12 = 10368). To make the average number of times seeing each state the same as in section 1.1.3, 1600K training games is used in this new discretization setting. **The AI player can rebound the ball 14.18 times** on average of 5000 test games. This is only slightly better than section 1.1.3, despite the fine discretization, 16 times of training games and 302,695ms of training time.

The conclusion is that **the original discretization settings are good enough** for the AI player to have a good performance. Further discretization does not result in much performance improvement. It may be possible to have a better performance by solving this problem using continuous environment instead discrete environment.

1.2 Two-Player Pong

1.2.1 Implementation

Not much changes in implementation is needed for the Q-Learning AI player. The hard-coded auto-player traces the ball and tries to place the paddle’s center at the same y-coordinate as the ball, except the moving speed is only half of the AI player.

One potential change in the state space is to take the auto-player’s location into account, which may increase the size of the state space by 12~20 times. However, in this game, the player has no control about which direction the ball may bounce, since the Y velocity is set randomly at each bounce. Therefore, it is not possible for the AI player to develop a strategy to bounce the ball toward the further side of the opponent’s location. In the end, **the auto-player’s location is not added into the state space**.

A **side effect** of the two-player game is that **the training speed is faster** than one-player playing against the wall. This is because the auto-player may miss the ball more often and end the game, resulting in 3509ms training time for 100K games, less than half of section 1.1. **This side effect may be negative for the AI player**, because it learns less information for the same amount of trainings due to less overall rebounds

1.2.2 Game results

The AI player wins 90.9% of the test games against the hard-coded auto-player after 100K trainings. A few sample games is shown in the animation below:

- Part_1_2_10 (https://youtu.be/VqVnSQZMNlI): Q-Learning player playing against the hard-coded player after 10 trainings
- Part_1_2_100K (https://youtu.be/xuNDVS8lgA0): Q-Learning player playing against the hard-coded player after 100K trainings

It is obvious that the hard-coded auto-player has a better performance after only 10 training. But the auto-player has no improvements from the trainings, while **the AI player keeps learning and dominates the game after 100K trainings**. It can be observed that the trained AI player is able to anticipate the ball's rebound on the wall and move accordingly, whereas the hard-coded player simply traces the center of the ball toward the wall and do not have enough time to move in the opposite direction after the rebound.

1.3 Interactive Game (Extra Credit)

An interactive game is created to enable the user controlling one paddle through keyboard and playing against a trained AI player. For the training phase of the AI player, it still plays against the wall and learns to rebound the ball as much as possible without knowing the actions of a future human player. (**Note: to play the game, please uncomment function Part_1_3() in Main.java, press 's' to start the game** after the GUI showing up after 100K trainings.)

Apparently, **the performance of a human player is affected by the speed of the game**. Faster game speed favors the AI player since the human player will not have enough time to move the paddle. To have a competitive performance, the following game animation have a **game speed about half of before** (in section 1.1 and 1.2). Even with such a low speed, **the human player (me) still cannot win the AI** due to mistakes in anticipating the rebound location of the ball.

- Part_1_3_Interactive (https://youtu.be/HcJpuBKXltc): Q-Learning player playing against the human player (author) after 100K trainings

1.4 Gravitational Pong (Extra Credit)

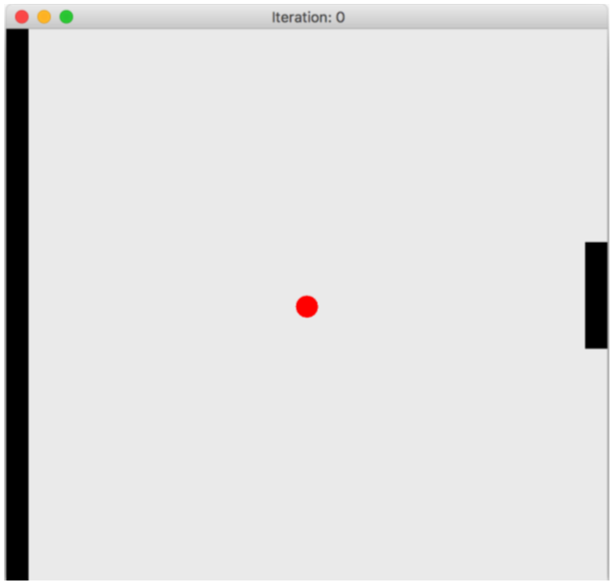
An extra rule is added to the game to see how the AI player adjusts to the changes in the environment. There is **a gravity force applied on the ball**, constantly changing the Y velocity 0.002 downward at every time step, resulting in a curved path of the ball, which will **make the game harder**. As you can see in the following animation, **the AI player can rebound the ball 6.69 times after 1000K trainings**, without a more complex state space representation.

- Part_1_4_Gravity (https://youtu.be/IxqOG-WizqQ): Q-Learning player playing a gravitational pong game against the wall after 1000K trainings

Since the challenge of this gravitational ball mainly comes from the Y velocity, finer discretization in Y velocity should and indeed improve the performance. By **dividing the ball's Y velocity into 5 ranges** ({ (-∞, -0.05), [-0.05, -0.015), [-0.015, 0.015), [0.015, 0.05), [0.06, ∞) }) instead of the original 3 ranges, **the AI player is able to reach 7.56 average rebounds** without changing any other parameters.

1.5 Graphic User Interface (Extra Credit)

This section corresponds to the 1st Extra Credit instruction. A simple GUI is created through Java swing package to visualize the game play. The pong ball is represented by a red dot and the wall or players are represented by black boards. The GUI is illustrated as below, and can be understood in all animations in section 1.1 to 1.4.



1.6 Game Animation (Extra Credit)

This section corresponds to the 3rd Extra Credit instruction. All game animations are shown in section 1.1 to 1.4 and are listed below for summarization. Every animation is linked to the corresponding YouTube video and the **original .mov file is included in source code zip file**.

- Part_1_1_10 (https://youtu.be/0_cvhSdY3yg): Single Q-Learning player playing against the wall after 10 trainings
- Part_1_1_1K (https://youtu.be/sIIu_oALTPE): Single Q-Learning player playing against the wall after 1000 trainings
- Part_1_1_100K (https://youtu.be/0V17bNkKScY): Single Q-Learning player playing against the wall after 100K trainings
- Part_1_2_10 (https://youtu.be/VqVnSQZMNlI): Q-Learning player playing against the hard-coded player after 10 trainings
- Part_1_2_100K (https://youtu.be/xuNDVS8lgA0): Q-Learning player playing against the hard-coded player after 100K trainings
- Part_1_3_Interactive (https://youtu.be/HcJpuBKXltc): Q-Learning player playing against the human player (author) after 100K trainings
- Part_1_4_Gravity (https://youtu.be/IxqOG-WizqQ): Q-Learning player playing a gravitational pong game against the wall after 1000K trainings

2-Digit Classification

2.1 Perceptron

2.1.1 Parameter Tuning

Learning rate $\alpha = \frac{5}{5+\mathrm{t}}$, in which ***t*** is the number of times trained on each data, aka epochs. A large constant gives a slower decrease in learning rate, in which case more epochs are needed for the weights to converge. Learning rate with small constant reaches convergence quickly, but may stop at suboptimal results. Constant 5 is found through trials, considering both **converging speed and classification accuracy**.

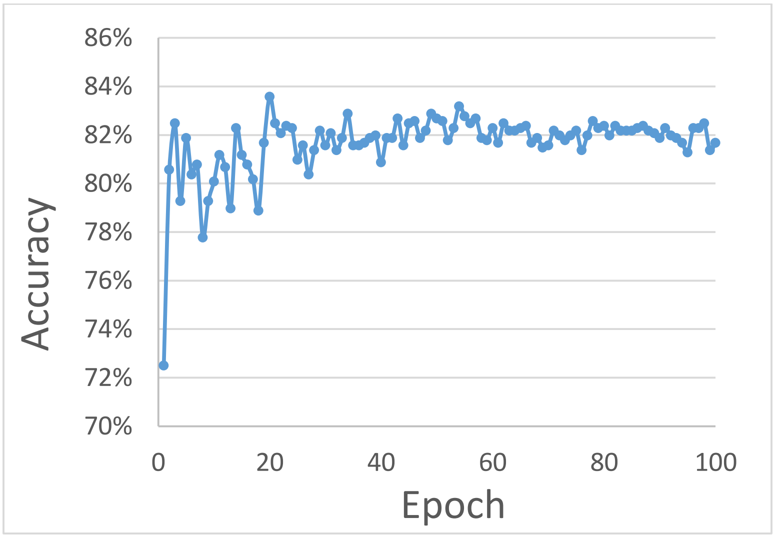
Bias factor is turned on, but does not affect the overall classification accuracy much in trials. The potential explanation is that all digit pictures have some pixels to be foreground, so the situation where all features are 0 is not possible. However, **the bias factor still slightly improves the performance** by reducing the bias on other features (pixels).

Initial weights are set to zeros instead of random values. It is found through trials that random initialization rarely improves the overall classification accuracy. A possible explanation is that when weights are updated, only the ones corresponding to foreground pixels are changed, so the problem of updating all weights every time is naturally avoided. Thus, random initialization is not needed in this case.

Ordering of training examples are set to random instead of fixed. It is found that the training ordering has almost no effect on the final classification accuracy. It can be imaged that the weight vector may oscillate forever when seeing some training data in the same order every time, if learning rate is not properly decreasing.

2.1.2 Training Curve

As shown in the curve below, **the classification accuracy oscillates with not enough training**. When epoch grows, the learning rate decreases and all weight values converges. The accuracy-epoch curve becomes smoother with less oscillation.



2.1.3 Confusion Matrix

Based on the best **epoch = 20** from section 2.1.2, the **overall classification accuracy is 83.58%** and the corresponding confusion matrix is shown below.

Digit	Accuracy	Confusion Matrix									
		0	1	2	3	4	5	6	7	8	9
0	93.17%	**93.17%**	0.00%	1.14%	0.00%	0.00%	2.30%	2.26%	1.12%	1.11%	0.03%
1	98.13%	0.00%	**98.13%**	0.93%	0.00%	0.93%	0.00%	0.93%	0.00%	0.00%	0.00%
2	90.19%	0.00%	0.98%	**90.19%**	0.98%	0.97%	0.99%	1.98%	1.96%	2.92%	0.01%
3	68.65%	0.00%	0.00%	3.03%	**68.65%**	0.00%	16.16%	3.05%	7.07%	3.01%	0.03%
4	90.54%	0.00%	0.00%	0.95%	0.93%	**90.54%**	0.00%	2.84%	0.01%	1.87%	3.80%
5	82.41%	2.17%	0.00%	1.10%	4.41%	1.11%	**82.41%**	1.12%	3.30%	5.46%	0.02%
6	86.70%	1.10%	1.11%	2.21%	0.00%	3.33%	2.22%	**86.70%**	2.21%	2.22%	0.00%
7	79.97%	0.00%	1.90%	6.68%	1.90%	2.86%	0.94%	0.00%	**79.97%**	0.00%	6.70%
8	75.37%	0.98%	1.96%	5.93%	3.91%	1.96%	5.90%	1.00%	1.96%	**75.37%**	2.01%
9	77.87%	0.00%	0.00%	0.00%	2.01%	8.06%	4.03%	0.00%	8.03%	1.01%	**77.87%**

Comparing to the accuracy with Naïve Bayes Classifier in Assignment 3 (75.5%), **Perceptron has an overall better performance**. Perceptron has higher accuracy for almost all digits, except 3 and 9, which tends to be misclassified into 5 and 4, which is the same trend as in Naïve Bayes Classifier.

2.2 K-Nearest Neighbor

2.2.1 Similarity Function

(1) p-norm:

All p-norm will give the same result as 1-norm (aka Manhattan Distance). The reason is that each entry in the vector representation of a digit's image is either 1 or 0. So the difference of an entry between two images is always a binary value. Since 1 raise to the power of n is still 1 for any value of n, the following equation will always be true, in which x_i and y_i denotes the ith entry of the vector for image X and Y.

$$\sum_i (x_i - y_i)^1 = \sum_i (x_i - y_i)^n\,, \qquad x_i, y_i \in \{0, 1\}$$

As a result, the n-norm distance between two vectors is the same as $\sqrt[n]{Manhattan\ Distance}$. The images' similarity ranking based on vector distances will not change no matter what n is, only the absolute value changes.

In conclusion**, the reciprocal of Manhattan Distance is chosen as one option of similarity measure**, other p-norm methods will not be tried.

(2) Dot product:

Dot product of two vectors can be used as the similarity measure as below, in which x_i and y_i denotes the ith entry of the vector for image X and Y. The shortage of this method is that it has no normalization, which bias the digits with more foreground pixels.

$$Similarity = d_x \cdot d_y = \sum_i x_i * y_i$$

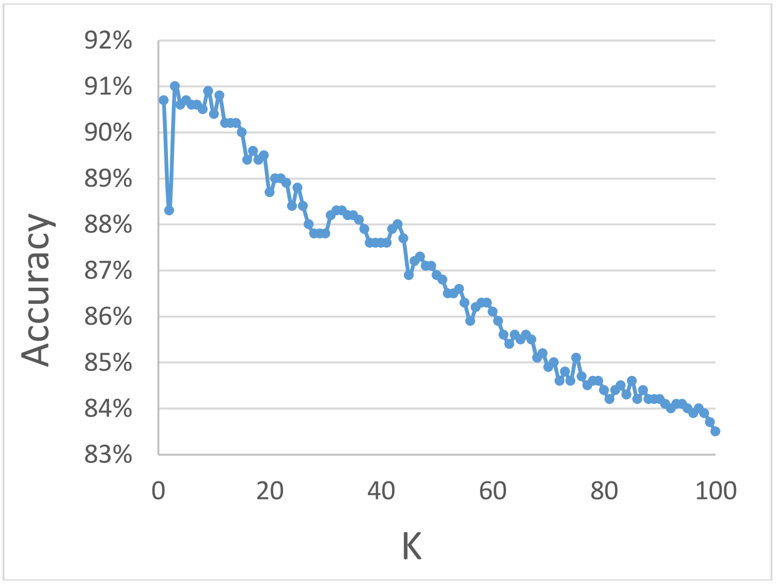
(3) Cosine Similarity:

Cosine similarity is the normalized dot product, where the vector dot product is divided by the norm of two vectors. This measure has a better overall classification performance than the above 2, possibly due to the normalization.

$$Similarity = \frac{d_x \cdot d_y}{|d_x| \cdot |d_y|}$$

2.2.2 Accuracy-K Curve

The classification results are shown for the Cosine Similarity who has the best overall performance. Manhattan distance method has close but slightly worse accuracy (~89%), but dot product method has much worse performance (~75%) then the other 2.



The classification accuracy is close for a reasonable range of K, but then decreases when K becomes too large. One potential explanation is that the current similarity measure only tells how many pixels are different between two images, without taking the pixel location into account. Then any two images with N different pixels are considered to be equally dissimilar. So when k is large, a lot images of various digits becomes valid "neighbor".

2.2.3 Confusion Matrix

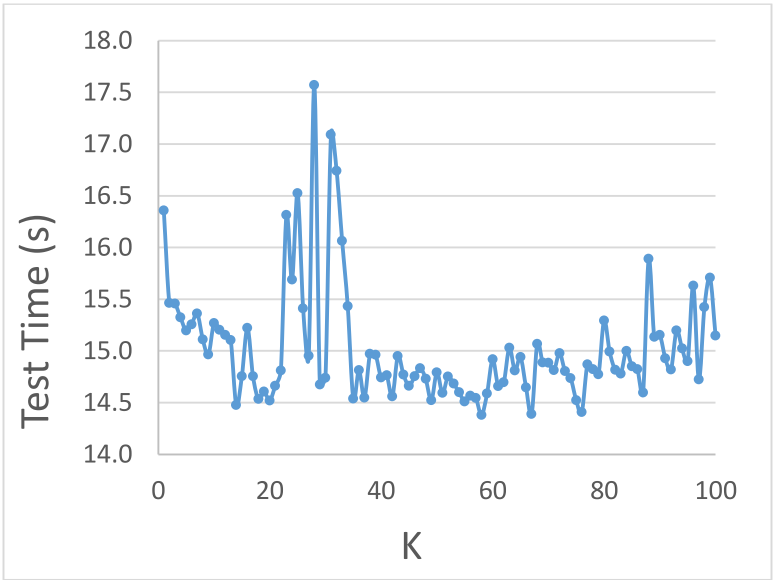
K=11 has 90.8% classification accuracy and the confusion matrix is shown in the following table. As it can be observed, the **K-nearest neighbor has a better performance than both Naïve Bayes and Perceptron** in solving this digit classification problem.

Digit	Accuracy	Confusion Matrix									
		0	1	2	3	4	5	6	7	8	9
0	100.00%	**100.00%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
1	99.07%	0.00%	**99.07%**	0.00%	0.00%	0.00%	0.00%	0.93%	0.00%	0.00%	0.00%
2	92.23%	0.97%	0.97%	**92.23%**	1.94%	0.00%	0.00%	0.97%	0.97%	1.94%	0.00%
3	94.00%	0.00%	0.00%	0.00%	**94.00%**	0.00%	0.00%	0.00%	3.00%	1.00%	2.00%
4	85.05%	0.93%	0.93%	0.00%	0.00%	**85.05%**	0.00%	3.74%	0.00%	0.00%	9.35%
5	79.35%	4.35%	0.00%	0.00%	5.43%	0.00%	**79.35%**	2.17%	0.00%	6.52%	2.17%
6	96.70%	2.20%	1.10%	0.00%	0.00%	0.00%	0.00%	**96.70%**	0.00%	0.00%	0.00%
7	81.13%	0.94%	3.77%	1.89%	0.00%	0.00%	0.00%	0.00%	**81.13%**	0.00%	12.26%
8	89.32%	1.94%	0.97%	0.97%	4.85%	0.00%	0.00%	0.00%	1.94%	**89.32%**	0.00%
9	92.00%	1.00%	0.00%	0.00%	2.00%	2.00%	1.00%	0.00%	0.00%	2.00%	**92.00%**

2.2.4 Running Time

The running time for **K-nearest method is longer than both Naïve Bayes and Perceptron**, because it **goes through all training data** to find the top K nearest ones for each testing data.

Let ***N*** be the number of training data. Let ***d*** be the number of pixels in each row/column of one image. By using a priority queue to record the top ***K*** nearest neighbors, the **theoretical worst case running time is O(N*d^2*log(K))**. In practice, the worst case is not often since the priority queue will not be updated in each loop once the K-nearest neighbors have been found. Thus, the expected increase in running time is **not observed** in the following time-K plot.



It is also possible to find the top K nearest neighbors in O(N*d^2) time, which is independent from K. When visit all training data and computing the similarity/distance to the testing data, a heap of training data can be created within linear time (heapify). To get the top K nearest neighbors, simply extract K data from the heap, which will take O(K*log(N)) time. The total time for comparison, heapify and extraction will be O(N*d^2+N+K*log(N)), and will be dominated by O(N*d^2) as long as K is small relative to N.

2.3 Advanced Features (Extra Credit)

Using the same parameter settings as section 2.1, **grouped pixels are treated as one feature in Perceptron** classifier. Same disjoint and/or overlap group sizes are tried as in Part 1.2 Assignment 3, and results are summarized as below.

Feature Set	Overall Accuracy	Digit Accuracy										Time (ms)	
		0	1	2	3	4	5	6	7	8	9	Training	Testing
Disjoint 2*2	87.6%	96.7%	97.2%	82.5%	86.0%	90.7%	81.5%	89.0%	86.8%	80.6%	85.0%	5820	29
Disjoint 2*4	87.1%	91.1%	96.3%	82.5%	91.0%	94.4%	80.4%	90.1%	80.2%	78.6%	86.0%	4092	22
Disjoint 4*2	88.3%	93.3%	97.2%	90.3%	88.0%	89.7%	90.2%	86.8%	84.0%	81.6%	82.0%	5149	25
Disjoint 4*4	84.8%	86.7%	96.3%	83.5%	91.0%	89.7%	70.7%	87.9%	81.1%	77.7%	82.0%	4866	26
Overlap 2*2	90.3%	96.7%	97.2%	88.4%	89.0%	91.6%	88.0%	91.2%	86.8%	87.4%	87.0%	24121	121
Overlap 2*4	89.5%	95.6%	99.1%	82.5%	89.0%	92.5%	90.2%	92.3%	84.0%	82.5%	88.0%	60081	265
Overlap 4*2	90.4%	95.6%	98.2%	93.2%	84.0%	90.7%	94.6%	94.5%	86.8%	81.6%	86.0%	66521	306
Overlap 4*4	88.7%	94.4%	97.2%	86.4%	87.0%	87.9%	85.9%	92.3%	86.8%	80.6%	89.0%	171681	861
Overlap 2*3	89.9%	94.4%	98.2%	86.4%	88.0%	91.6%	90.2%	92.3%	82.1%	87.4%	89.0%	34076	158
Overlap 3*2	90.1%	95.6%	96.3%	90.3%	91.0%	90.7%	87.0%	92.3%	83.0%	83.5%	92.0%	41570	196
Overlap 3*3	90.1%	94.4%	98.2%	90.3%	82.0%	91.6%	92.4%	89.0%	81.1%	90.3%	92.0%	80006	360

As shown in the table, **the overall accuracy is higher** for grouped pixels than single pixels. But when using grouped pixels as features, applying Perceptron classifier gives close results to Naïve Bayes classifier. Similarly, overlapped pixel group methods have slightly better performance, by also takes more time in training and testing.

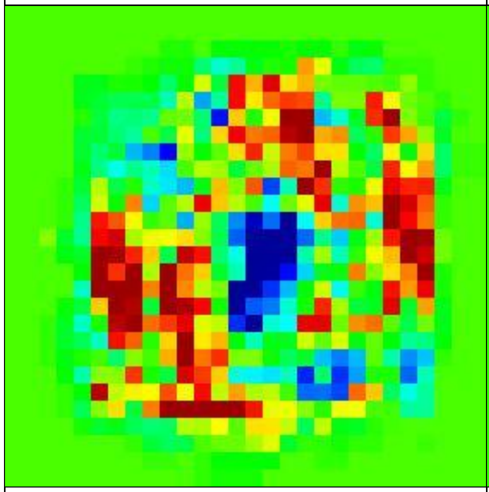
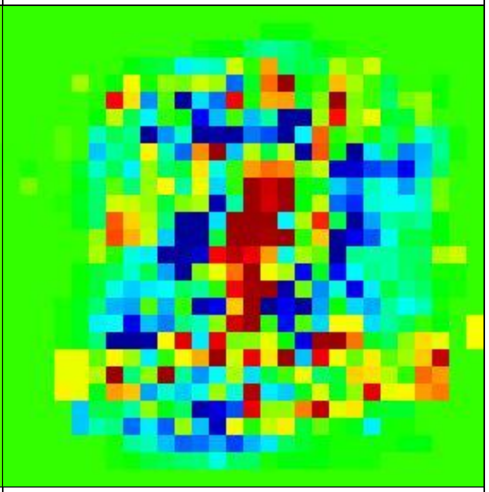
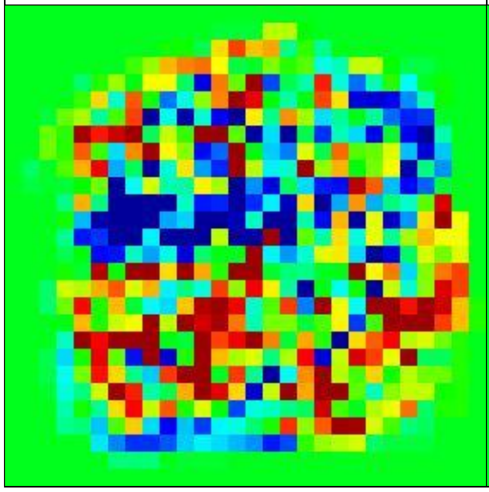
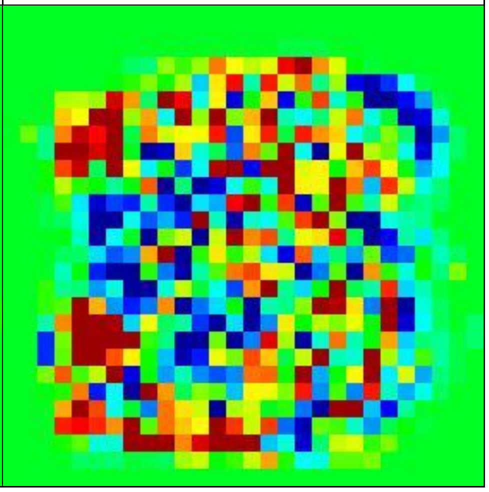
2.4 Perceptron Weights Visualization (Extra Credit)

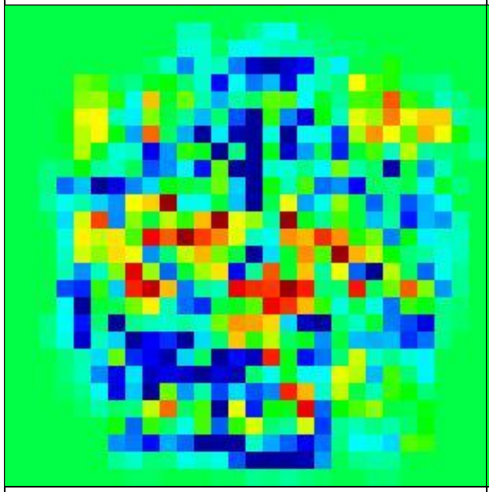
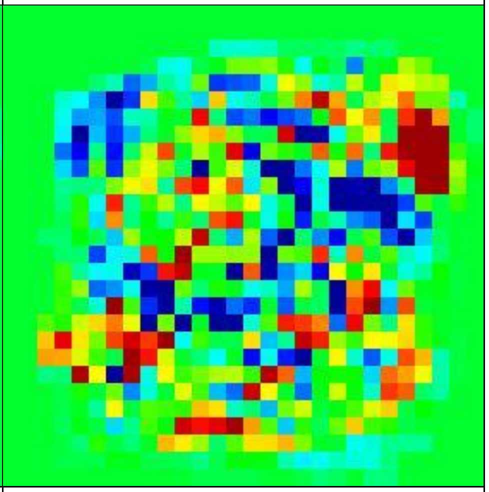
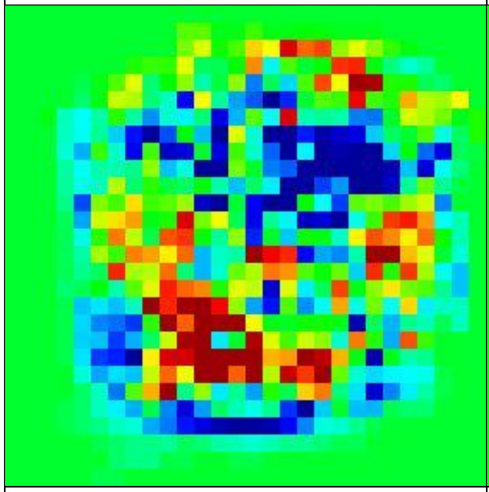
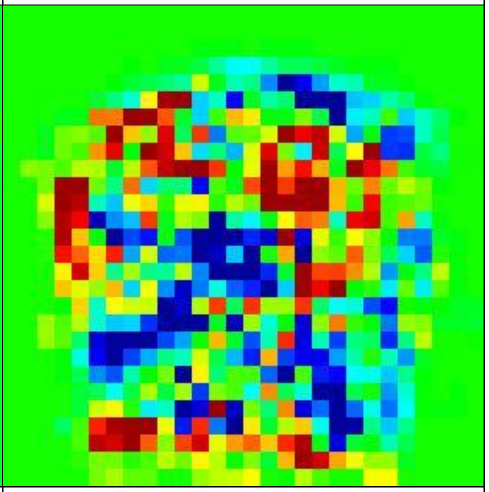
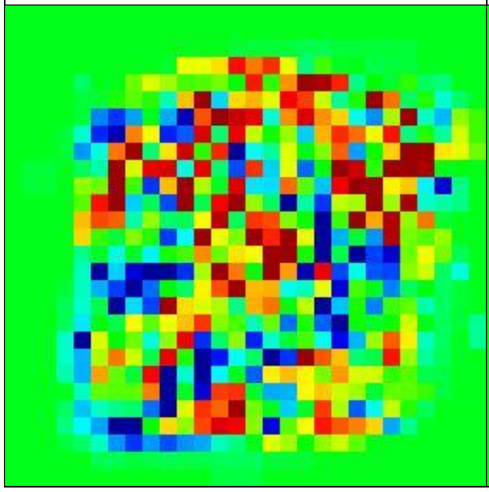
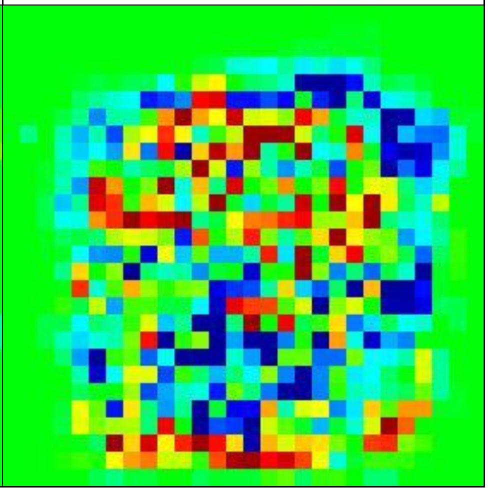
Similar to visualization of likelihood and odds ratio, the perceptron weights of each pixel are visualized as a color image for every digit, as shown in the following table. **The images are not ideal as the expected blurred digit plot, but there are some trends worth mentioning**:

(1) Blue pixel means negative weight and red pixel means positive weight. If a foreground pixel in the testing image occurs at a blue pixel in digit ‘0’ weight plot, it reduces the possibility that the testing image is digit ‘0’.

(2) Blocks of blue/red pixels are good indication of areas that give hints to classification results. For example, the center of image ‘0’ is a blue block. If a testing image has foreground pixels near the center, it is very likely that the image is not digit ‘0’.

(3) Since each Perceptron distinguish one digit from all other digits, only pixels with high possibility differentiating this digit from all others will be shown as red and/or blue. But if a pixel is a good hint for several digits, it will not be a good hint for one certain digit. For example, the weight plot for digit ‘8’ is very chaotic, because most area of the shape ‘8’ is shared with other digits.

0	**1**
	
2	**3**
	

4	**5**
	
6	**7**
	
8	**9**
	

2.5 Differentiable Perceptron (Extra Credit)

2.5.1 Implementation

Two major implementation changes are made to modify the original non-differentiable perceptron into a differentiable perceptron:

(1) The output of classification decision is processed by the Sigmoid function instead of the simple dot product of two vectors.

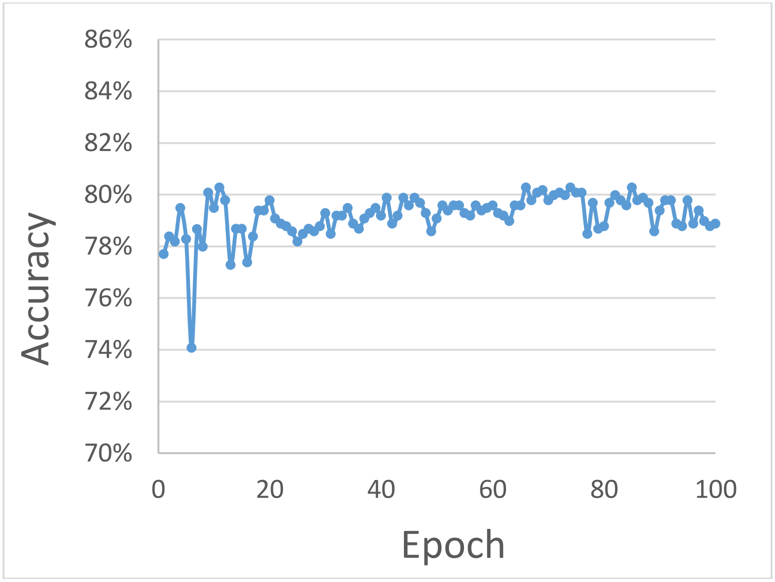
$$output = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \frac{1}{1 + e^{-(\boldsymbol{w} \cdot \boldsymbol{x} + b)}}$$

(2) The updates to the weight vector take the gradient descent into account, illustrated as below.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\big(y - f(\boldsymbol{x})\big)\sigma(\boldsymbol{w} \cdot \boldsymbol{x})\big(1 - \sigma(\boldsymbol{w} \cdot \boldsymbol{x})\big)\boldsymbol{x}$$

2.5.2 Training Curve

Same parameter settings in section 2.1.2 are used here to compare the convergence speed and quality between the Differentiable Perceptron (DP) and Non-Differentiable Perceptron (NDP). As shown in the following curve, the results from **DP converges much faster than NDP** without any extra help from the learning rate α (same for both methods). However, the converged classification accuracy is not necessarily better than NDP, in this case, it's about 2% lower. (Note: the vertical axis is set to be the same range as in section 2.1.2 for better visual comparison.)



2.5.3 Confusion Matrix

As stated in section 2.5.2, the maximum overall classification **accuracy is 80.28%** and occurs at **epoch = 74**, but is **not improved** by implementing the Differentiable Perceptron.

Digit	Accuracy	Confusion Matrix									
		0	1	2	3	4	5	6	7	8	9
0	91.00%	**91.00%**	0.00%	1.12%	0.00%	0.00%	2.25%	2.25%	1.12%	1.14%	2.25%
1	99.05%	0.00%	**99.05%**	0.94%	0.00%	0.00%	0.00%	0.93%	0.00%	0.01%	0.00%
2	79.41%	0.98%	2.94%	**79.41%**	5.88%	2.94%	0.98%	1.96%	0.98%	1.96%	2.94%
3	78.79%	0.00%	1.01%	2.02%	**78.79%**	0.00%	8.08%	2.02%	6.06%	3.03%	0.00%
4	81.14%	0.00%	0.94%	3.78%	1.89%	**81.14%**	0.93%	2.83%	2.83%	0.00%	6.59%
5	80.23%	4.38%	0.00%	0.00%	5.49%	1.10%	**80.23%**	2.20%	1.10%	3.30%	3.30%
6	81.11%	0.00%	1.11%	4.44%	0.00%	5.56%	7.78%	**81.11%**	0.00%	1.11%	0.00%
7	79.03%	0.00%	0.95%	4.76%	2.86%	0.95%	0.95%	0.95%	**79.03%**	1.90%	8.59%
8	64.72%	0.98%	1.96%	7.84%	8.82%	0.00%	7.84%	1.96%	1.96%	**64.72%**	4.89%
9	75.75%	0.00%	0.00%	1.01%	1.01%	11.12%	4.04%	1.01%	6.06%	1.01%	**75.75%**

3-References

[1] https://courses.edx.org/courses/BerkeleyX/CS188x_1/1T2013/info

[2] https://en.wikipedia.org/wiki/Q-learning

[3] https://en.wikipedia.org/wiki/Perceptron

[4] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[5] https://en.wikipedia.org/wiki/Cosine_similarity

[6] http://stackoverflow.com/questions/4956593/optimal-algorithm-for-returning-top-k-values-from-an-array-of-length-n