CS 440 Assignment 4 Markov Decision Processes, Reinforcement Learning, and Classification

Weijie Huo (4 credits), netid: whuo3

Shang Zhang (4 credits), netid: szhan110

Part 1: MDPs and Reinforcement Learning

1.1 (for everybody): Grid World MDP

Value Iteration Implementation: Basically, Iterate through every square of the board. For each square, compute the bellman equation (Calculate the expect utility for every action in a state, then choose the largest utilities of actions as current policy and update the utility of the current state). Keep iterating until the difference between new utility and old utility is less than 0.0001 for every square/states.

Policy Iteration Implementation: Basically, Iterate through every square of the board. For each square, compute the bellman equation (Find the best action by the max utility around, then calculate the expect utility for the given action, then update the current state and policy). It will keep iterating until there is no policies changed for at least 5 rounds.

Value Iteration with terminal convergence takes **23** iterations:

Utilities of all states:

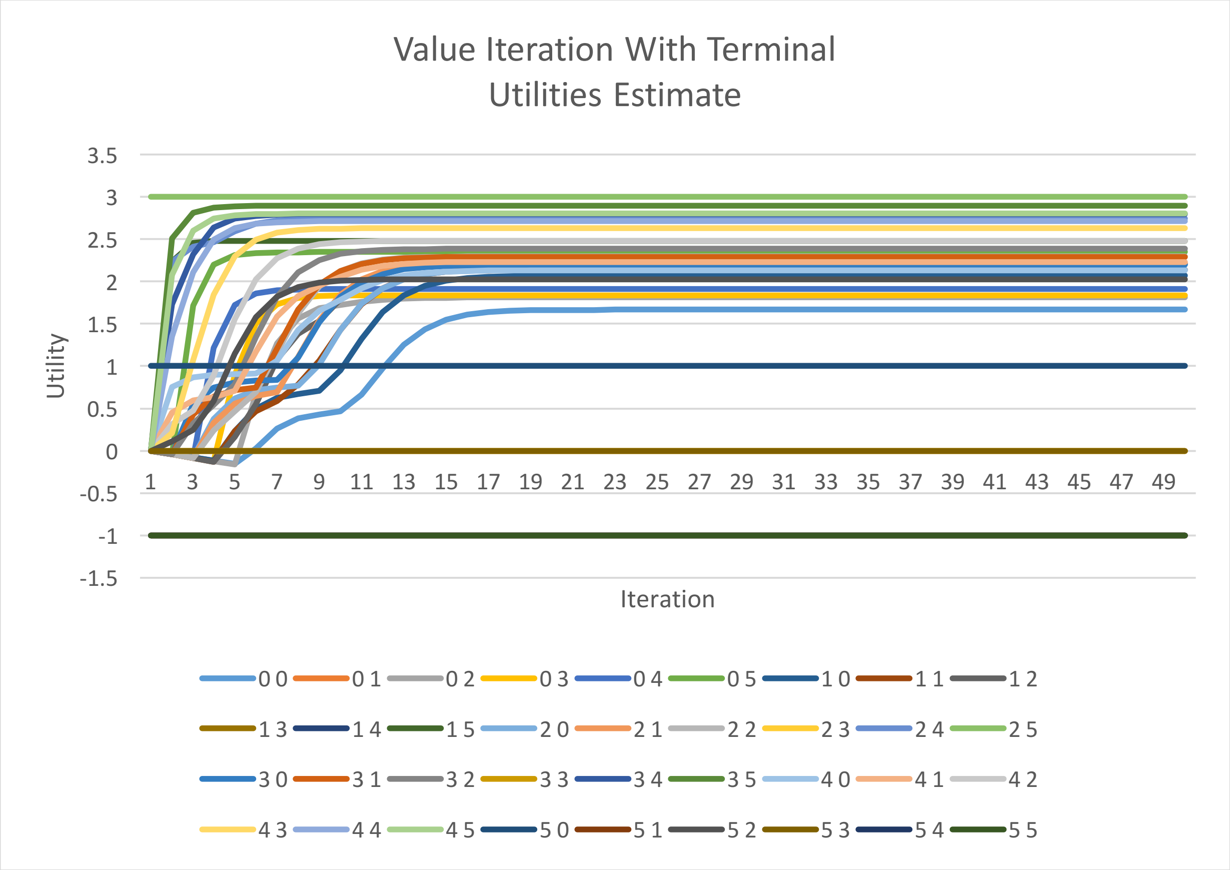
1.666	-1	1.812	1.836	1.91	2.348
2.071	2.141	2.21	W	-1	2.483
2.139	2.218	2.297	W	2.744	3
2.197	2.291	2.387	W	2.797	2.9
2.132	2.231	2.479	2.629	2.713	2.803
1	-1	2.025	W	-1	-1

Optimal policy:

↓	-1	↓	→	→	↓
→	↓	↓	W	-1	↓
→	→	↓	W	→	3
→	→	↓	W	→	↑

→	→	→	→	→	↑
1	-1	↑	W	-1	-1

Utility Estimate:



Value Iteration without terminal

If we set the converge threshold to be no changes of policy in consecutive 5 iterations. It converges in **16 iterations**

If we set the converge threshold to be that the difference between old utility and new utility of all states is less than 0.1, it takes about 280 iterations to converge:

Utilities of all states:

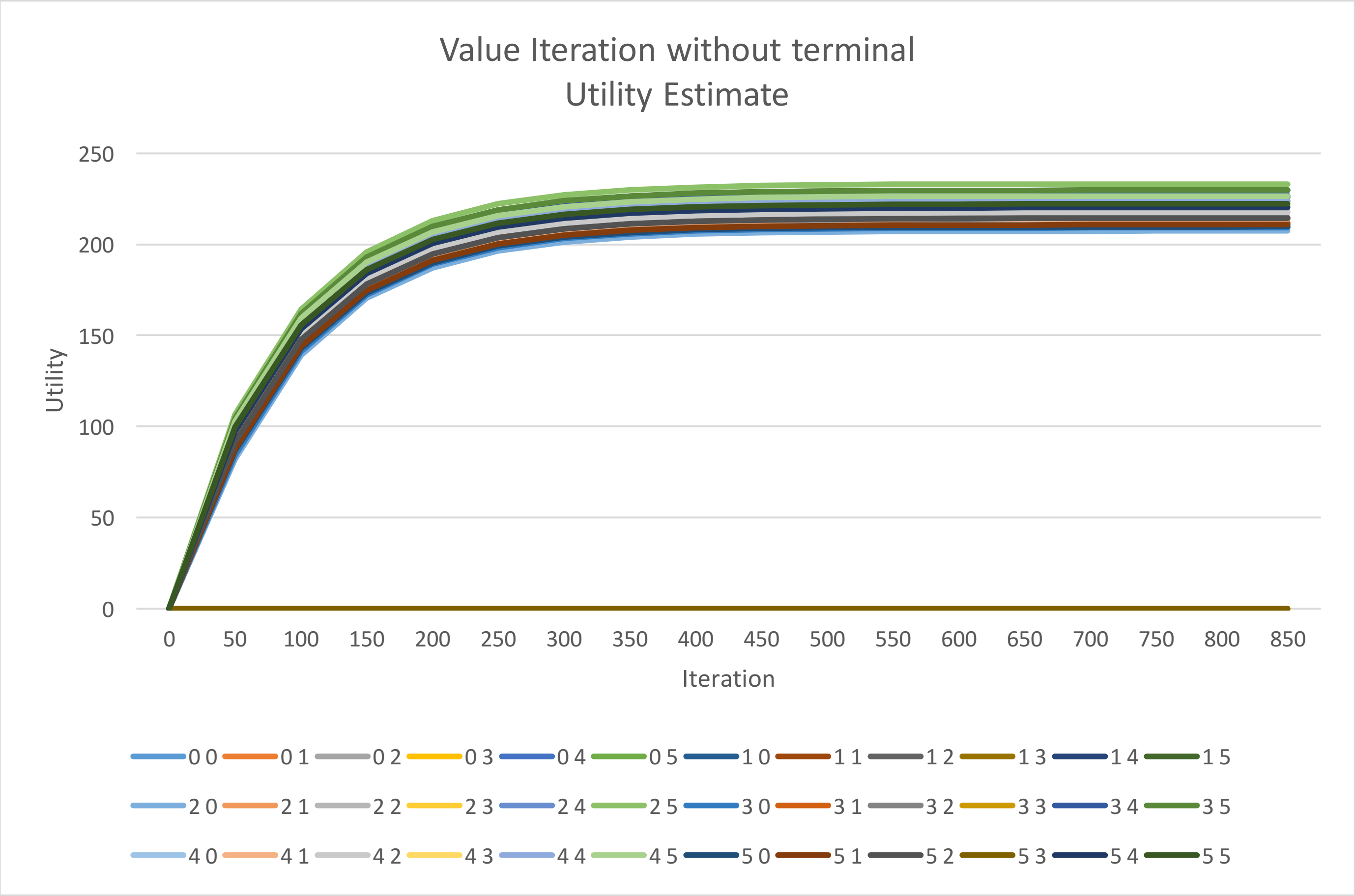
210.7	213.6	217.8	220.9	223.8	226.4

209.1	211.8	214.6	W	225.7	229.6
207.3	209.5	211.7	W	229.4	233.1
209	211.7	214.4	W	226.8	229.8
210.9	214	217.5	221.1	223.9	226.5
209.7	210.8	214.3	W	220.1	222.2

Optimal policy:

→	→	→	→	→	↓
→	→	↑	W	→	↓
→	↑	↑	W	→	→
→	→	↓	W	→	↑
→	→	→	→	↑	↑
↑	→	↑	W	↑	↑

Utility Estimate:



Policy Iteration with terminal convergence takes 20 iterations:

Utilities of all states:

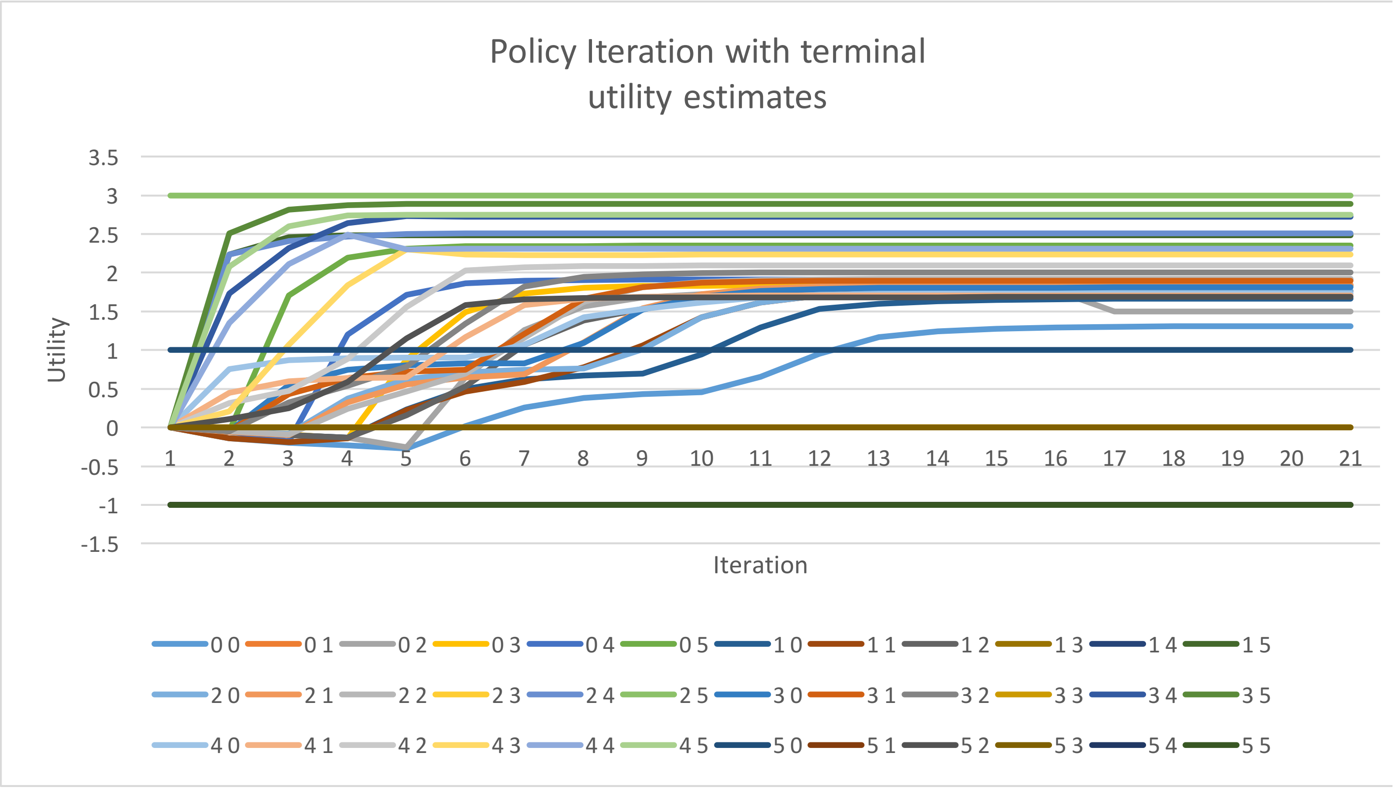
1.31	-1.00	1.50	1.84	1.91	2.35
1.66	1.77	1.84	W	-1.00	2.48
1.76	1.84	1.92	W	2.51	3.00
1.81	1.90	2.00	W	2.73	2.89
1.73	1.71	2.09	2.23	2.31	2.75

1.00	-1.00	1.69	W	-1.00	-1.00

Optimal policy:

↓	-1	↓	→	→	↓
→	↓	↓	W	-1	↓
→	→	↓	W	→	3
→	→	↓	W	→	↑
↑	→	→	→	→	↑
1	-1	↑	W	-1	-1

Utility Estimate:



Policy Iteration without terminal convergence

If we set the converge threshold to be no changes of policy in consecutive 5 iterations. It converges in **17 iterations**

If we set the converge threshold to be that the difference between old utility and new utility of all states is less than 0.1, it takes about 307 iterations to converge:

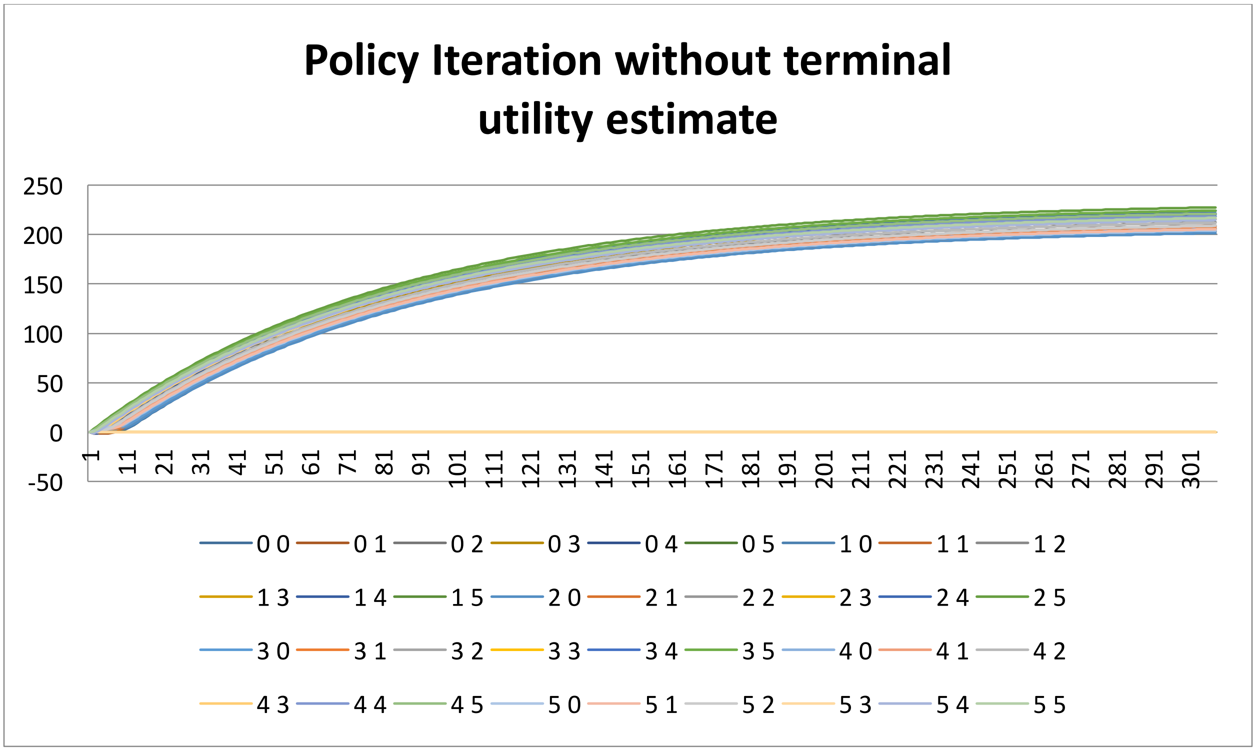
Utilities of all states:

21.53	24.68	29.11	32.55	35.9	38.98
22.05	25.02	28.04	W	38.36	42.5
22.12	24.69	27.29	W	42.42	46.19
24.68	27.6	30.55	W	41.73	44.89
27.19	30.37	33.9	37.48	40.82	43.65
27.68	28.73	32.66	W	38.82	41.28

Optimal policy:

→	→	→	→	→	↓
→	→	↑	W	→	↓
→	↓	↓	W	→	→
→	→	↓	W	→	↑
→	→	→	→	→	↑
→	→	↑	W	→	↑

Utility Estimate:



1.2 (for everybody): Grid World Reinforcement Learning

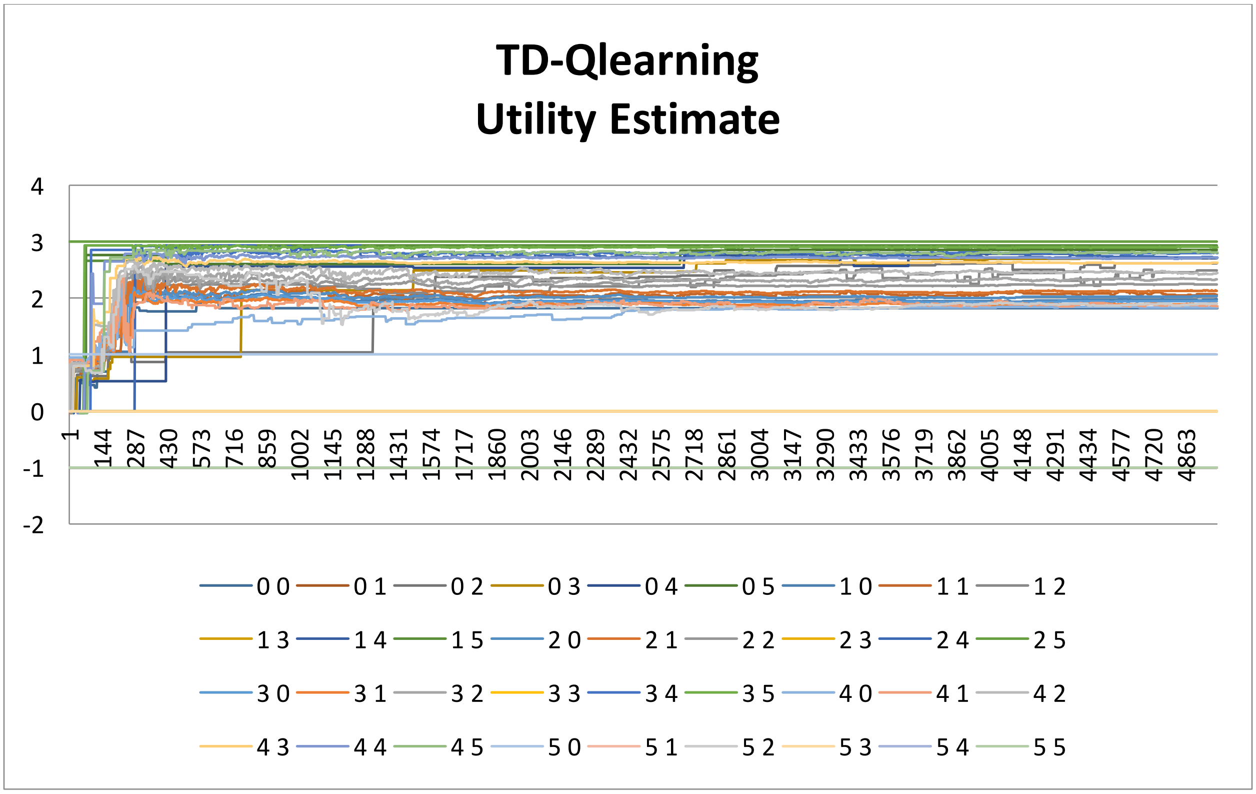
TD Q-learning:

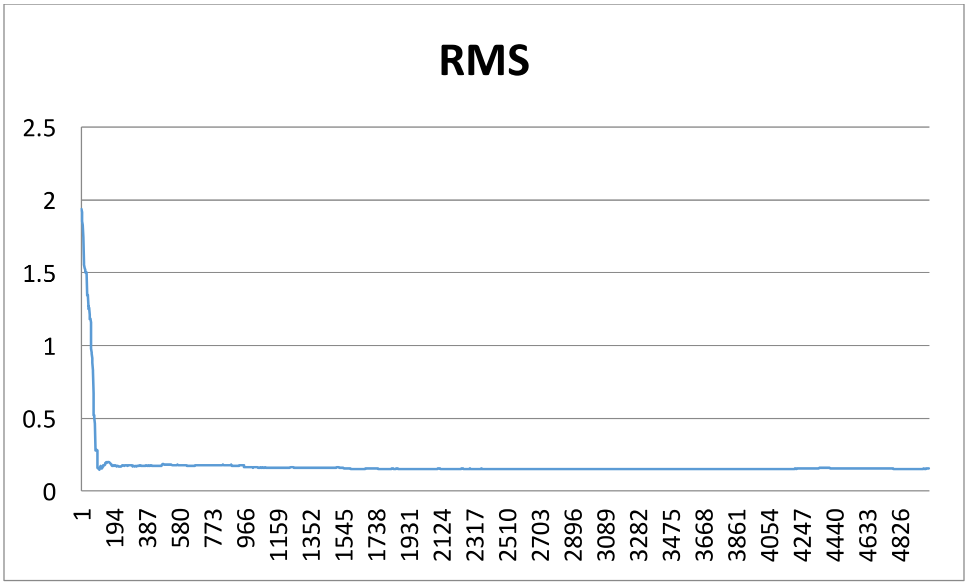
Reinforcement learning implementation: Maintain a Q values (Double[4]) in every state, each element in the array represents Q(s, a). Agent start from start, and then traverse the maze with the following rule: 1. Get action, if there are some actions has not been tried for more than Ne times, it will randomly generate a number to compare with other Q value. Return the action with max value. 2. Take action, update the Q value correspond to the action following the steps of TD learning. 3. If agent reach terminal, set the agent back to start to restart. 4. Redo the above steps for 5000 times.

Parameters: Ne(Optimistic reward estimate) = 40, number of trials: 5000,

$\alpha = 60 / (59 + t)$

t := how many times the specific action being executed in the specific state


TD-Qlearning
Utility Estimate
4
3
2
1
0
-1
-2
1
144
287
430
573
716
859
1002
1145
1288
1431
1574
1717
1860
2003
2146
2289
2432
2575
2718
2861
3004
3147
3290
3433
3576
3719
3862
4005
4148
4291
4434
4577
4720
4863
0 0
0 1
0 2
0 3
0 4
0 5
1 0
1 1
1 2
1 3
1 4
1 5
2 0
2 1
2 2
2 3
2 4
2 5
3 0
3 1
3 2
3 3
3 4
3 5
4 0
4 1
4 2
4 3
4 4
4 5
5 0
5 1
5 2
5 3
5 4
5 5


RMS
2.5
2
1.5
1
0.5
0
1
194
387
580
773
966
1159
1352
1545
1738
1931
2124
2317
2510
2703
2896
3089
3282
3475
3668
3861
4054
4247
4440
4633
4826

1.3 (for 4-unit students): Pizza Delivery

Implementation of Pizza Delivery: The algorithm of Pizza delivery is basically the same as 1.2. The difference is that instead of having 4 Q values in a square, now we have 4 states in a square. Therefore, there are 16 Q values. Each Q value correspond to the state and action. Besides, this is a non terminal game. For each step, update the Q value according to the state and action.

The Parameter: Ne(Optimistic reward estimate) = 40, number of trials: 5000,

$\alpha = 60 / (59 + t)$

t := how many times the specific action being executed in the specific state

No Pizza, No Ingredient

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	R	D	R	D	D	D	D	D	D	L	L	L	L	W
W	R	R	R	R	D	D	D	L	D	L	L	L	L	W
W	R	U	U	W	D	D	D	D	D	W	U	U	L	W
W	W	W	W	W	D	R	R	R	D	W	W	W	W	W
W	D	D	L	D	D	L	U	R	D	W	D	D	D	W
W	R	D	L	L	L	R	R	D	R	R	R	L	L	W
W	U	U	U	L	U	L	U	R	R	R	R	U	L	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

No Pizza, yes Ingredient

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	R	R	R	R	R	R	R	R	R	L	L	L	L	W
W	R	U	U	R	U	L	R	U	U	L	L	L	L	W
W	R	U	U	W	R	R	U	R	U	W	R	U	L	W
W	W	W	W	W	D	U	R	D	U	W	W	W	W	W
W	R	D	L	D	R	R	U	R	U	W	D	L	D	W
W	R	R	D	D	L	D	R	U	U	L	L	L	D	W
W	R	R	R	R	R	L	L	L	L	L	U	L	L	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

Yes Pizza, No Ingredient

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	D	D	L	D	L	L	L	L	L	D	D	D	D	W
W	R	U	L	L	L	L	L	R	R	R	R	L	L	W
W	U	U	U	W	U	U	U	U	U	W	U	U	L	W

W	W	W	W	W	U	U	U	R	U	W	W	W	W	W
W	R	D	L	L	U	L	R	R	U	W	D	D	L	W
W	R	R	L	L	D	L	L	L	U	L	R	L	L	W
W	U	U	L	L	L	L	U	U	U	L	U	U	L	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

Yes Pizza, Yes Ingredient

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	R	D	L	L	L	L	R	D	D	D	D	D	D	W
W	R	R	L	L	L	L	R	R	R	R	R	U	L	W
W	U	U	U	W	U	U	R	R	U	W	U	U	U	W
W	W	W	W	W	U	R	R	U	L	W	W	W	W	W
W	R	R	R	R	U	U	U	R	U	W	D	L	U	W
W	R	U	R	U	L	L	U	L	L	L	L	D	L	W
W	U	R	R	R	D	U	U	L	U	L	L	L	R	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

Extra Credit:

We add a Home state, for each each move of the agent, the negative reward will be multiplied by a number, the number will be increased for every move of agent. Whenever the agent come home, the number will be recover to 1. In the following example, the home state is marked as H. For every table show as below, whenever a agent want to go through the middle, it will always try to bypass the 'H' square.

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	→	→	→	↓	↓	↓	↓	←	↓	←	←	←	←	W
W	→	→	↑	→	↓	↓	↓	↓	←	←	↑	↑	←	W
W	↑	↑	↑	W	↓	↓	↓	←	←	W	↑	↑	↑	W
W	W	W	W	W	↓	→	H	←	↑	W	W	W	W	W
W	→	↓	←	←	←	←	↑	↑	↓	W	→	↓	→	W
W	→	→	←	↑	←	↓	↓	↓	←	→	↑	←	←	W
W	↑	↑	←	←	←	←	←	←	←	→	→	→	↑	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

W	↓	→	→	→	↓	→	→	↓	→	←	←	←	←	W
W	→	↓	→	→	→	↑	→	→	↑	↑	↑	←	←	W
W	→	→	↑	W	↑	→	↓	→	↑	W	↑	↑	→	W
W	W	W	W	W	→	→	H	↑	↑	W	W	W	W	W
W	→	→	→	→	→	↑	↑	↑	↑	W	↓	↓	←	W
W	→	→	→	→	↓	↓	↑	↑	←	←	←	↑	→	W
W	→	↑	→	→	→	→	↑	↑	←	←	←	←	↓	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	→	↓	←	←	←	←	←	→	→	→	→	↓	↓	W
W	→	→	←	←	←	←	↑	↑	→	→	→	↑	←	W
W	→	↑	↑	W	↑	↑	→	→	↑	W	↑	↑	←	W
W	W	W	W	W	↑	→	H	←	↑	W	W	W	W	W
W	←	←	→	→	→	↑	↑	↑	↑	W	↓	→	←	W
W	→	↑	↑	↑	↑	↑	↑	↑	↑	←	←	←	→	W
W	→	↓	←	↓	→	→	↑	←	↑	↑	↑	↑	←	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
W	↑	→	↑	←	←	→	→	→	→	→	→	↓	←	W
W	↑	↑	↑	↓	←	↑	→	↑	↑	→	→	←	↓	W
W	→	↓	↑	W	→	↑	↑	←	↑	W	→	↑	↑	W
W	W	W	W	W	→	→	↑	←	←	W	W	W	W	W
W	→	→	→	→	↑	↑	↑	↑	←	W	↓	↓	↓	W
W	→	↑	↓	→	→	↑	↑	↑	←	←	←	←	←	W
W	→	↑	→	→	→	→	↑	←	←	←	←	↑	↑	W
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

Part 2.1 (for everybody): Digit Classification with Perceptron

In this part, a perceptron was built to classify the images, the value of the pixel which is the feature was taken as 1 if it is foreground and 0 if it is background, weight vectors were generated for each digit class, in the beginning, the components of the weight vectors were all set to zero, the outputs of each digit class were calculated based on the sum of all the features times their corresponding weights, and the one with highest output was used as the result as the classification. And the results are listed below:

Training curve:

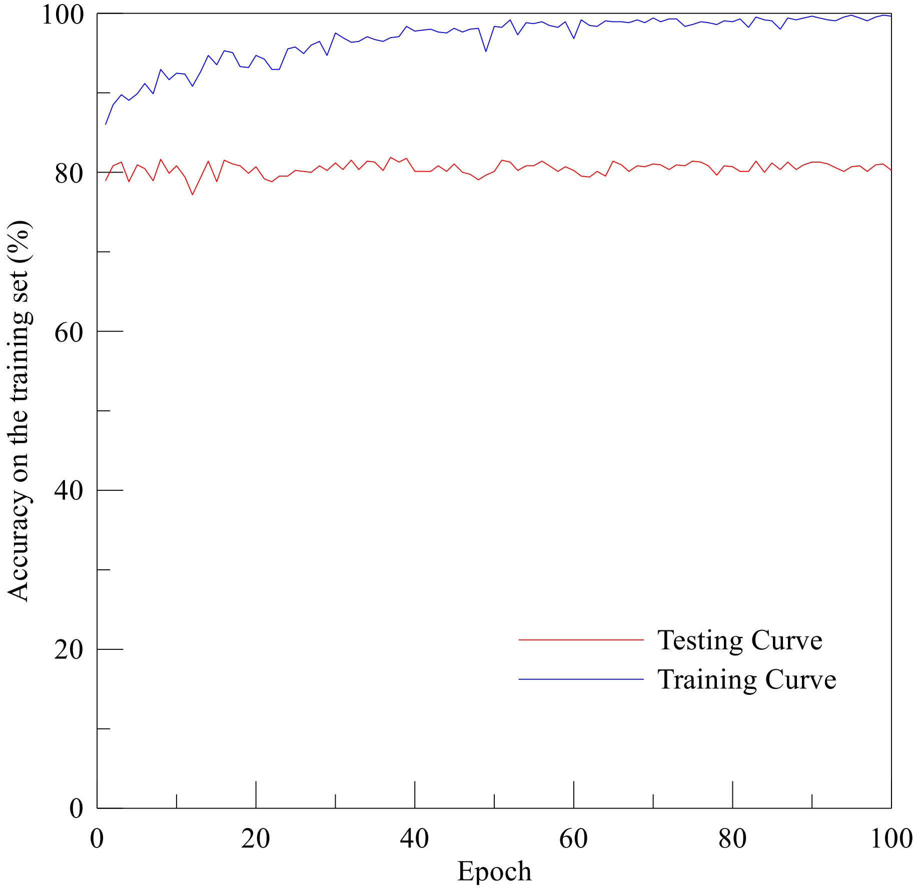


Figure 1 Training curve

Maximum overall accuracy on the test set is 81.7% corresponding to epoch 8.

Confusion matrix:

	0	1	2	3	4	5	6	7	8	9
0	86.67%	.00%	3.33%	.00%	.00%	3.33%	1.11%	.00%	4.44%	1.11%
1	.00%	99.07%	.00%	.00%	.00%	.00%	.93%	.00%	.00%	.00%
2	.00%	1.94%	81.55%	1.94%	.00%	.97%	5.83%	1.94%	4.85%	.97%
3	.00%	.00%	2.00%	66.00%	.00%	18.00%	3.00%	6.00%	3.00%	2.00%
4	.00%	.93%	1.87%	.00%	86.92%	.00%	2.80%	.00%	.00%	7.48%
5	2.17%	.00%	1.09%	3.26%	.00%	81.52%	1.09%	2.17%	7.61%	1.09%

6	1.10%	1.10%	2.20%	.00%	2.20%	1.10%	91.21%	1.10%	.00%	.00%
7	.94%	4.72%	4.72%	.00%	.00%	.00%	.00%	74.53%	.94%	14.15%
8	.00%	3.88%	8.74%	1.94%	.97%	10.68%	4.85%	1.94%	65.05%	1.94%
9	.00%	.00%	.00%	1.00%	6.00%	2.00%	.00%	6.00%	.00%	85.00%

The following parameters were tuned, and the discoveries are discussed below:

Learning rate decay function: A decay function alpha=1000/(1000+t) was used in the code, where t is the epoch, other decay functions were also tried in the code, for example alpha=1/(1+t), an overall accuracy of 62.3% was obtained, so it's worse than the former one.

Bias vs. no bias: A bias of 1 was used in the code, other values like zero were also tried, but same accuracies were obtained, this is mainly because for perceptron for multiple classes, the class with the highest output was taken as the results, so a bias was added to all digit classes, and it does not affect the order of the outputs of the digit classes.

Initialization of weights (zeros vs. random): the weight vectors were all set to zero in the code, randomly generated weight vectors were also tried, when generating randomly between 0-10, the accuracies obtained varies every time the code was ran, generally speaking, the accuracies were about 4% lower than all being set to zero case. When generating randomly between 0-100, the accuracies were about 10% lower than all being set to zero case. When generating randomly between 0-1000, the accuracies were about 40% lower than all being set to zero case. So in summary, all the weight vectors being set to zeros is superb, which makes sense because it is irrational to set weight to pixels at the beginning without knowing anything about the images.

Ordering of training examples (fixed vs. random): the order used in the final version of the code was from start to the end of the training set. Random order was also tried, the accuracies obtained from this tweak were sometimes higher and sometime lower, it seems doesn't have obvious advantage against following the natural order of the training set.

Number of epochs: the number of cycles of the training cases was used as the epoch in the code, and best accuracy was obtained when epoch is equal to 8. The relationship between the overall accuracy on the test set and the number of epochs is shown in Figure 1.

The table below shows the comparison between the results obtained from Naive Bayes and perceptron:

Naive Bayes	Single pixels	77.10%
	2*2 patches	85.40%
Perceptron	Single pixels	81.70%

For classifiers using single pixels, the accuracy obtained from perceptron is higher than that obtained from Naïve Bayes method. While the classifier using Naïve Bayes method with 2*2 patches got the highest accuracy.

Part 2.2 (for 4-unit students): Digit Classification with Nearest Neighbor

In this part, a classifier was established based on the nearest neighbor method, the distance function was taken as:

$$\mathrm{D} = \sum_{i}^{28}\sum_{j}^{28} |\, Test\ image(i,j) - Training\ image(i,j) \,|$$

Where: Test image(i, j) = 0 if the (i, j) pixel is blank, and 1 if it is not blank in test image

Training image(i, j) = 0 if the (i, j) pixel is blank, and 1 if it is not blank in training image

Distances based on the function above were calculated with all the training images when identifying one test image, the top k training images with the least distance were selected, and the classification result for the test image was decided by voting by the k nearest neighbors.

The overall accuracy on the test set as a function of k is shown below:

k	Accuracy (%)
1	90.2%
10	88.6%
50	84.2%
100	79.80%
200	74.3%
500	62.4%
800	54.40%
1000	51.4%
1500	45.5%
2000	37.8%
2500	31.90%
3000	25.6%

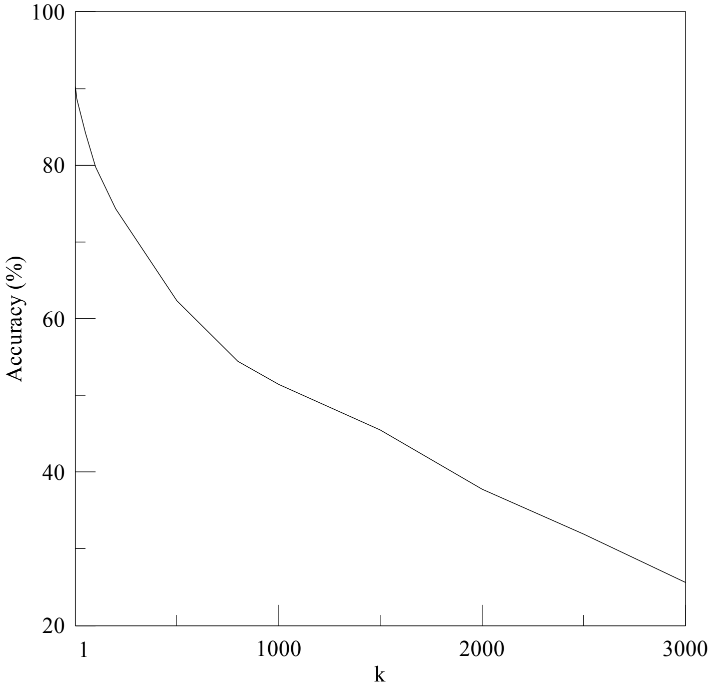


Figure 2 Relationship between k and accuracy

When using k=1, the obtained accuracy is the maximum, the confusion matrix obtained from k=1 is shown below:

	0	1	2	3	4	5	6	7	8	9
0	98.89%	.00%	.00%	.00%	.00%	.00%	1.11%	.00%	.00%	.00%
1	.00%	100.00%	.00%	.00%	.00%	.00%	.00%	.00%	.00%	.00%
2	.97%	1.94%	92.23%	1.94%	.97%	.00%	.97%	.97%	.00%	.00%
3	.00%	.00%	1.00%	82.00%	.00%	7.00%	.00%	3.00%	6.00%	1.00%
4	.00%	1.87%	.00%	.00%	87.85%	.00%	2.80%	.00%	.00%	7.48%
5	2.17%	.00%	.00%	4.35%	.00%	86.96%	3.26%	1.09%	1.09%	1.09%
6	.00%	1.10%	.00%	.00%	.00%	1.10%	97.80%	.00%	.00%	.00%
7	.94%	6.60%	1.89%	.94%	.94%	.00%	.00%	83.02%	.00%	5.66%
8	.97%	2.91%	2.91%	1.94%	1.94%	.97%	.00%	1.94%	84.47%	1.94%
9	.00%	.00%	.00%	3.00%	4.00%	2.00%	.00%	1.00%	.00%	90.00%

The running time of the code with different k values are listed below:

k	Time(s)
1	64.059
10	64.489
50	26.603
100	25.948
200	27.662
500	27.718

800	26.028
1000	28.376
1500	39.369
2000	31.391
2500	35.162
3000	38.199

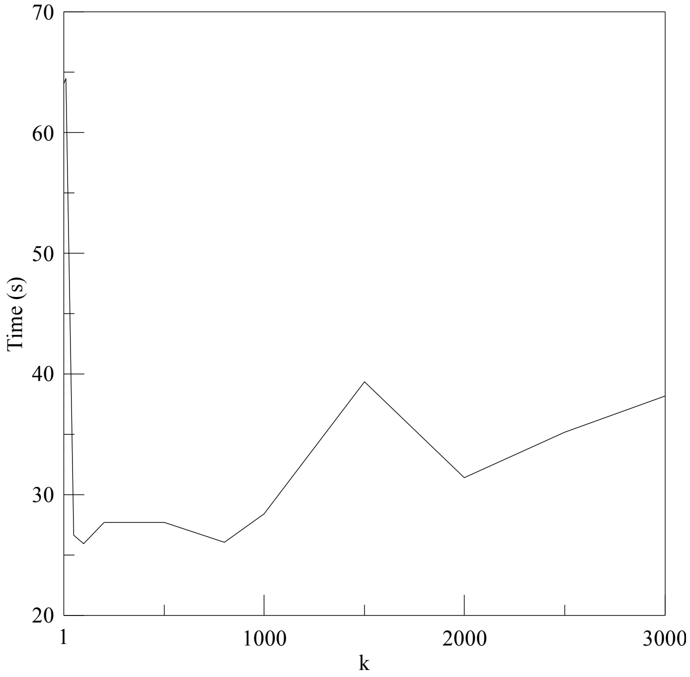


Figure 3 Relationship between k and running time

There is no obvious relationship between k and the running time simply based on the result shown above.

The comparison between different methods is shown below:

Naive Bayes	Single pixels	77.10%
	2*2 patches	85.40%
Perceptron	Single pixels	81.70%
Nearest Neighbor	Single pixels	90.20%

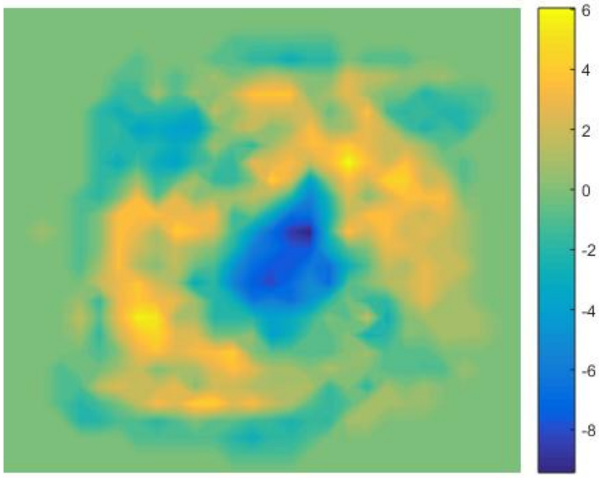
It can be seen that the accuracy obtained from nearest neighbor method is the highest.

Part 2 Extra Credit

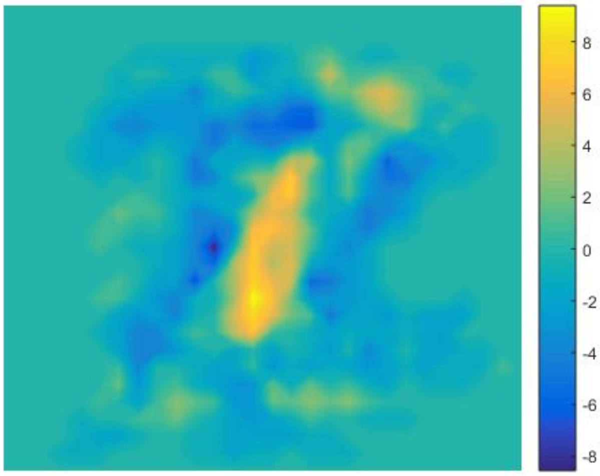
Visualization:

The visualizations of weight vectors of digit classes generated by the perceptron are shown below:

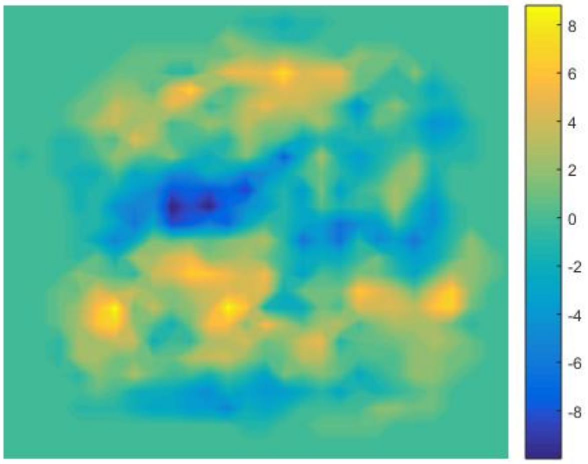
0:



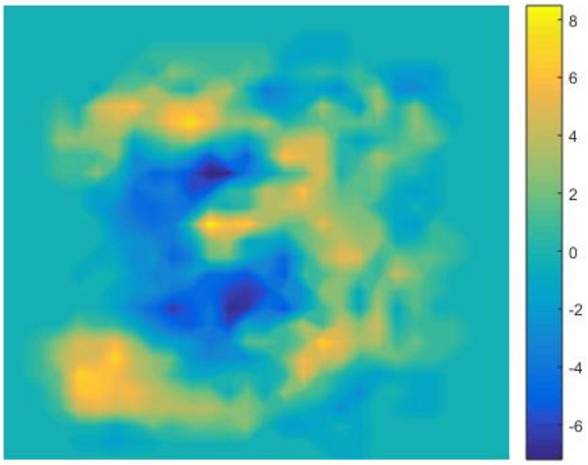
1:



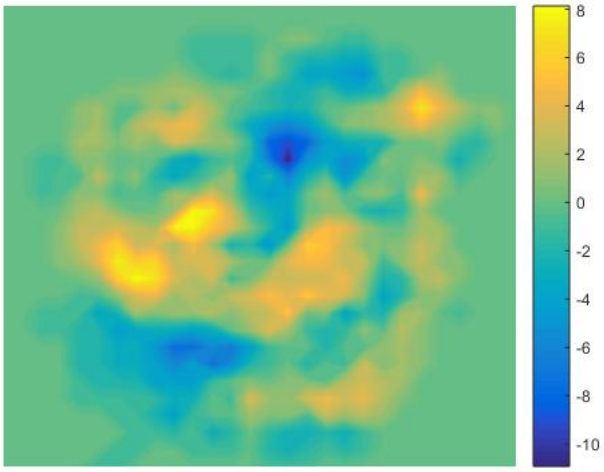
2:



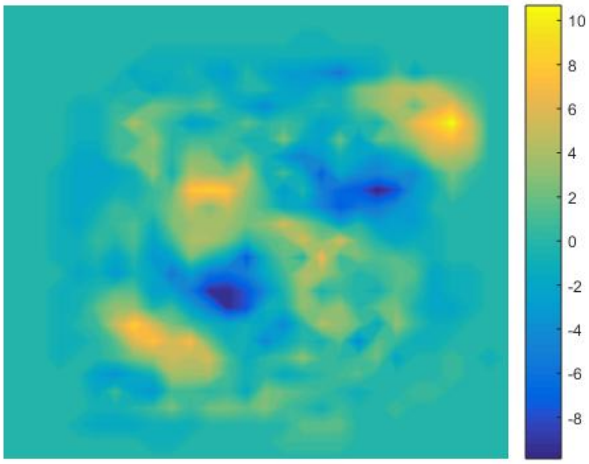
3:



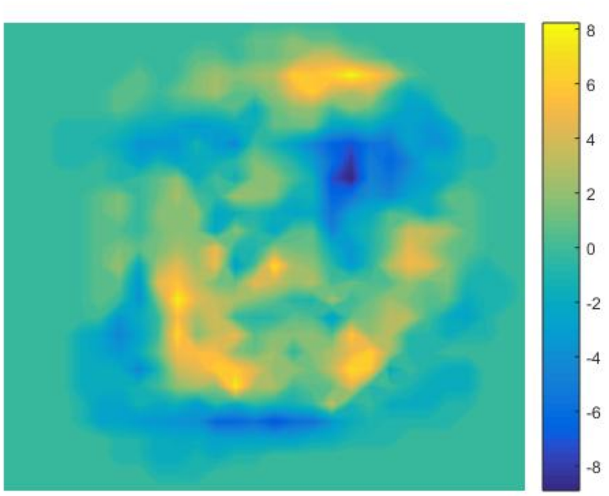
4:



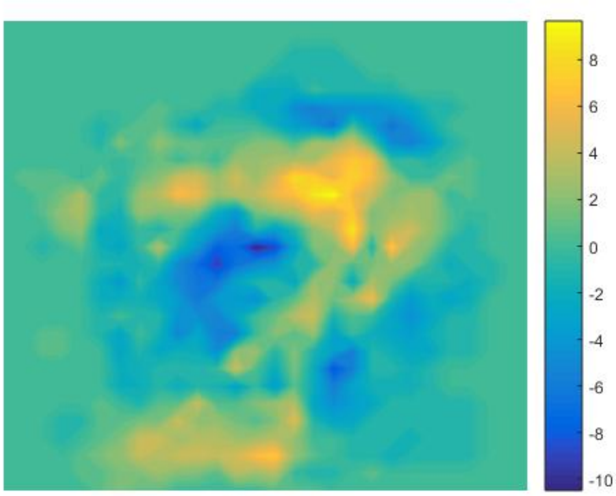
5:



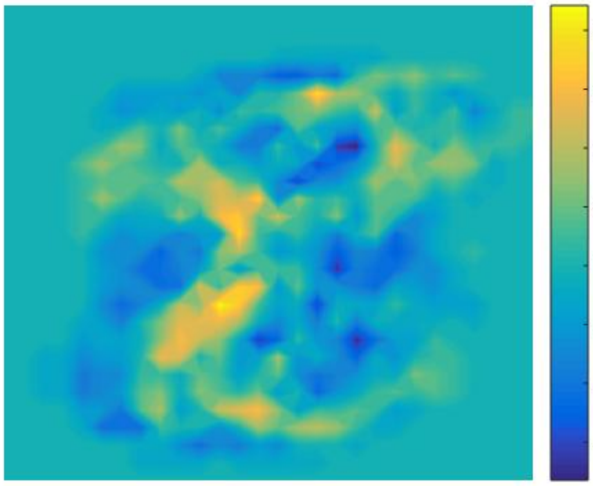
6:



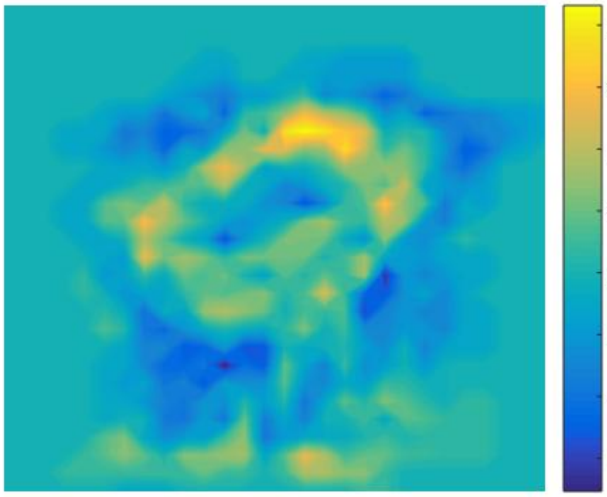
7:



8:



9:



In particular, the weight vectors indicate the location of the image where is unique to a certain digit, which means for that digit, features 1 are likely to occur in that location. If there's feature 1 in that area, the weight vectors will emphasis that feature in that location by multiplying a higher weight vector. For some other parts of the image of a certain class, feature 1 may be also occur, but that feature may also occur in other digits, so the perceptron will decrease the weight factor of that location for that class to reduce the chance of mistaken classification. For example, in the visualization for digit 9, the weight vectors are high in the upper central part, this means this feature is kind of unique to digit 9, if there's 1 occurred in that part, the perceptron is likely to classify the image as 9. While the negative weight vector region indicates rarely occurring of feature 1, again for digit 9, if feature 1 occurred in the lower left part in an image, the perceptron is not likely to classify it as 9.

Differentiable perceptron:

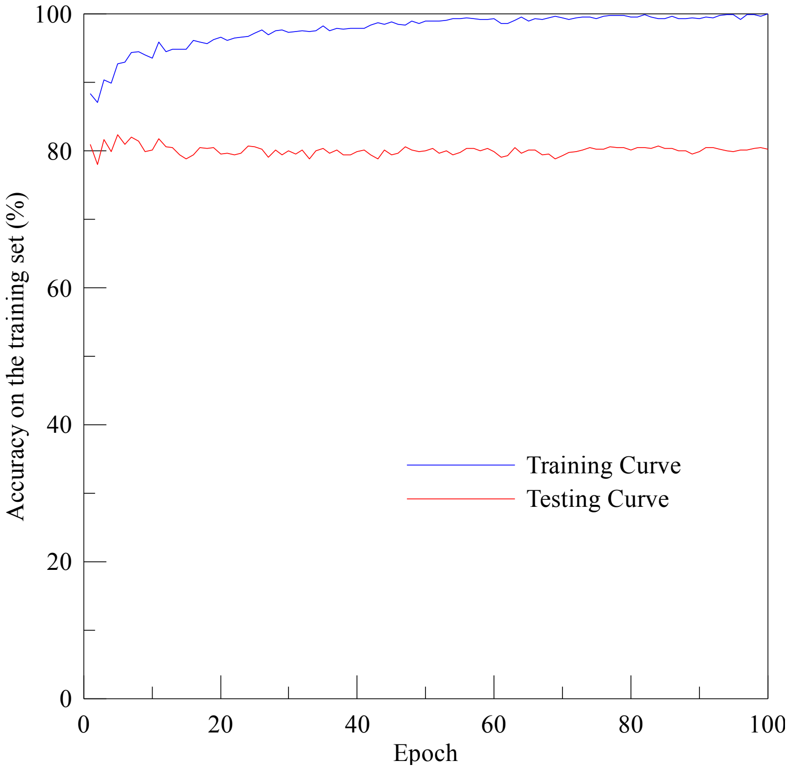
In this part, suppose example from class c gets misclassified as c', update the weights by gradient descent

Update for c: $w \leftarrow w + \alpha\sigma(w \cdot x)(1-\sigma(w \cdot x))x$

Update for c': $w \leftarrow w - \alpha\sigma(w \cdot x)(1-\sigma(w \cdot x))x$

When initializing all the weight vectors to zero, and going through the training set from the start to the end, an accuracy of 78.0% was obtained, which is lower than the non-differentiable case.

The training curve is shown below:



The confusion matrix is shown below:

	0	1	2	3	4	5	6	7	8	9
0	88.89%	.00%	1.11%	.00%	.00%	8.89%	1.11%	.00%	.00%	.00%
1	.00%	97.22%	1.85%	.00%	.00%	.00%	.93%	.00%	.00%	.00%
2	.00%	.00%	91.26%	.00%	.97%	.00%	3.88%	.97%	.00%	2.91%
3	.00%	.00%	10.00%	52.00%	.00%	28.00%	3.00%	6.00%	1.00%	.00%
4	.00%	.00%	2.80%	.00%	90.65%	.00%	2.80%	.93%	.00%	2.80%
5	3.26%	.00%	.00%	1.09%	1.09%	93.48%	.00%	1.09%	.00%	.00%
6	2.20%	1.10%	8.79%	.00%	3.30%	5.49%	78.02%	1.10%	.00%	.00%
7	.00%	3.77%	8.49%	.94%	.94%	1.89%	.00%	70.75%	.00%	13.21%
8	.97%	1.94%	16.50%	.97%	1.94%	31.07%	1.94%	2.91%	38.83%	2.91%
9	1.00%	.00%	1.00%	1.00%	8.00%	5.00%	.00%	4.00%	.00%	80.00%

INDIVIDUAL CONTRIBUTION

Weijie Huo: report , coding, design algorithms
Shang Zhang: coding ,report, design algorithms