```
           __
          /  \
        /      o
    __/ \__          .
  /  +    +  \         .
 |     w      |         .
 \___________/          .
```

CS 440/ECE 448 - 3 Hours
Spring 2015

Assignment 4: **Markov Decision Processes and Reinforcement Learning**

Instructor: Svetlana Lazebnik

Harrison Kiang - hkiang2, Jonathan Park - jgpark2, Gabriella Quirini - gquirin2

Part 1.1: Grid World

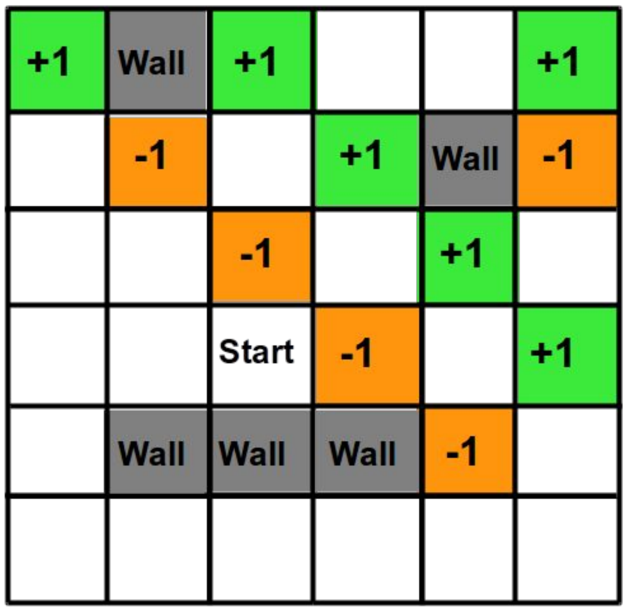


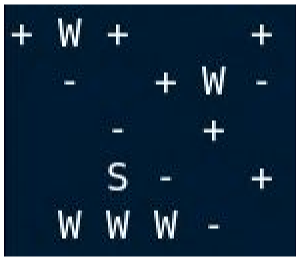
Figure 1: The provided grid for Part 1.1

Notes for the Reader

- The agent can move in any cell that is not a wall
- Each cell that is not blank or a wall has a fixed reward of either +1 or -1
- Each state s is a cell in the grid
- Each action a is a move either up, right, down, or left
- s - the current state
- s' - a potential next state (the next cell)
- a - an action taken from s
- a' - an action taken from s'
- $A(s)$ - the set of actions that can be made from state s
- $P(s'|s, a)$ - the transition model
- $\pi(s)$ - the action that an agent takes in any given state according to the transition model
- $R(s)$ - the reward function (fixed at -0.04)
- $Q(s, a)$ - the action-utility function that tells us the value of doing action a in state s
- $Q(s, a')$ - the action-utility function that tells us the value of doing action a' in state s
- $Q(s', a')$ - the action-utility function that tells us the value of doing action a' in state s'
- $U(s)$ - the utility of state s
- $U(s')$ - the utility of state s'
- t - the time step
- α - the learning rate. $\alpha = \frac{60}{59+t}$
- γ - the discount factor (fixed at 0.99)
- R^+ - the optimistic reward estimate
- N_e - a fixed number; the number of times that the agent has taken action a from state s

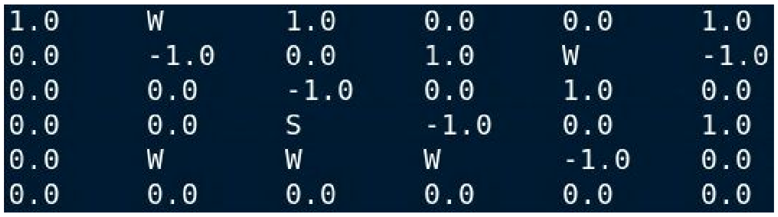
- Each cell is assigned a value based on the grid above, either 0, +1, or -1. The utilities for both value iteration and TD learning are initially set to these same values. Each iteration of the respective action-utility function $Q([arg1],[arg2])$ called by these cells will update the utilities for value iteration and TD learning.
- $Q(s',a')$ examines the utility of the action taken by the agent after its next move. For example, if s is [3,2] and it moves up, s' will be [2,2], and a' will be the best move from [2,2]. The utility from action a' is the value that is evaluated to be $Q(s',a')$.
- Moving out of bounds or into a wall will force the agent to stay in place.

Grid Parsing
We took the picture of the grid and transformed into a textfile for the application to parse.



```
+ W +     +
  -   + W -
    -   +
    S -   +
  W W W -
```

Named grid.txt

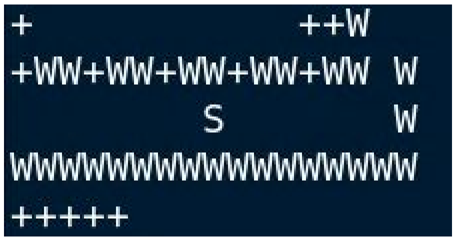


```
1.0     W       1.0     0.0     0.0     1.0
0.0     -1.0    0.0     1.0     W       -1.0
0.0     0.0     -1.0    0.0     1.0     0.0
0.0     0.0     S       -1.0    0.0     1.0
0.0     W       W       W       -1.0    0.0
0.0     0.0     0.0     0.0     0.0     0.0
```

(Grid.txt parsed with values)

Other grid files can be made and passed into the application as an argument.

For example, this is a **tricky** maze from **Assignment 1**: We used this for extra credit portion in the upcoming report.



```
+           ++W
+WW+WW+WW+WW+WW W
        S       W
WWWWWWWWWWWWWWWWW
+++++
```

Part 1.1.1
Value Iteration:

We chose value iteration to calculate the optimal policy (the best direction to try to take for each grid cell) for the Markov Decision Process under the given reward function and the fixed values for each cell marked +1/-1.

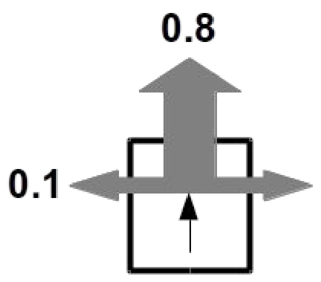


Figure 2: The transition model with up as the intended direction

The transition model, as shown above, shows that the agent has 80% to travel forward in its intended direction, but 10% in travelling to each of the perpendicular direction of the intended direction.

The reward function, $R(s)$ is simply -0.04 for all states, and is applied at each iteration of the utility calculation.

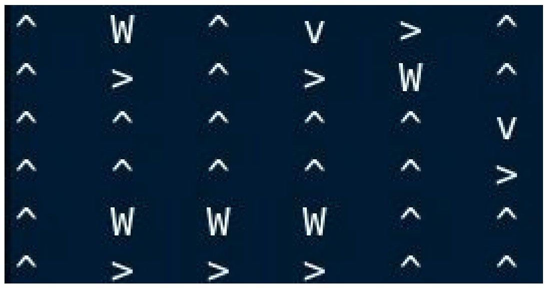
We iterate through the entire grid by each cell and find the intended direction of that state starting by going through the adjacent states of the starting state. Since there is no terminal state, we iterate indefinitely until we hit an iteration threshold or a convergence threshold.
We can calculate the Utility of the next iteration of a state with the following equation:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

Figure 3: The Bellman Equation used in value iteration

The next iteration of utility is equal to the reward function (-0.04) plus the total expected utility of state s' after taking the best action that could be taken from state s, scaled by the discount factor to make sooner value better than later values.

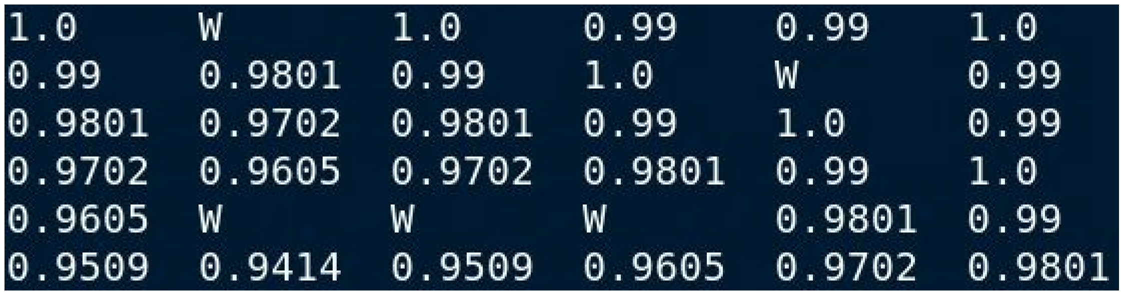
Here is the optimal policy after 10 iterations and a **convergence threshold of 0.01**:



```
^   W   ^   v   >   ^
^   >   ^   >   W   ^
^   ^   ^   ^   ^   v
^   ^   ^   ^   ^   >
^   W   W   W   ^   ^
^   >   >   >   ^   ^
```

Figure 4: The intended directions of non-wall cells calculated by value iteration

The following are the utilities of each state on the grid:



```
1.0     W       1.0     0.99    0.99    1.0
0.99    0.9801  0.99    1.0     W       0.99
0.9801  0.9702  0.9801  0.99    1.0     0.99
0.9702  0.9605  0.9702  0.9801  0.99    1.0
0.9605  W       W       W       0.9801  0.99
0.9509  0.9414  0.9509  0.9605  0.9702  0.9801
```

Figure 5: The utilities of each non-wall cell calculated by value iteration
or:

Listed in (col, row) format

```
( 0 , 0 ):  1.0
( 0 , 1 ):  0.99
( 0 , 2 ):  0.9801
( 0 , 3 ):  0.970299
( 0 , 4 ):  0.96059601
( 0 , 5 ):  0.9509900499
( 1 , 1 ):  0.9801
( 1 , 2 ):  0.970299
( 1 , 3 ):  0.96059601
( 1 , 5 ):  0.941480149401
( 2 , 0 ):  1.0
( 2 , 1 ):  0.99
( 2 , 2 ):  0.9801
( 2 , 3 ):  0.970299
( 2 , 5 ):  0.9509900499
( 3 , 0 ):  0.99
( 3 , 1 ):  1.0
( 3 , 2 ):  0.99
( 3 , 3 ):  0.9801
( 3 , 5 ):  0.96059601
( 4 , 0 ):  0.99
( 4 , 2 ):  1.0
( 4 , 3 ):  0.99
( 4 , 4 ):  0.9801
( 4 , 5 ):  0.970299
( 5 , 0 ):  1.0
( 5 , 1 ):  0.99
( 5 , 2 ):  0.99
( 5 , 3 ):  1.0
( 5 , 4 ):  0.99
( 5 , 5 ):  0.9801
```

Figure 6: The utilities of each non-wall cell calculated by value iteration formatted according to the reference utilities

Part 1.1.2

TD Q-learning:

Unlike value iteration and policy iteration, reinforcement learning implies that the agent does not know what the transition model and the reward function are. TD Q-learning is a type of model-free reinforcement learning wherein the agent learns how to act without explicitly learning the transition probabilities (transition model).

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

Figure 7: A general exploration function

The formula for TD Q-learning instead uses an exploration function that is defined as $f(Q(s,a'),N(s,a'))$, where $n = N(s,a')$ represents the number of times we've taken action a' from state s, and $u = Q(s,a')$ is the action-utility function that tells us the value of doing action a' in state s (see "Notes for the Reader" for help interpreting variables).

$$a = \arg\max_{a'} f\big(Q(s,a'),N(s,a')\big)$$

Figure 8: Formula to pick an action a for a given state s

The action for a state s is picked from a maximum of potential actions a'. If $n = N(s,a') > N_e$, in other words, if the agent has taken a certain action a enough times where it is confident that the operation u will give the correct utility, it will pick the action that maximizes $u = Q(s,a')$. Otherwise, it will return R^+, which is in our case hard-coded to +1.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\big)$$

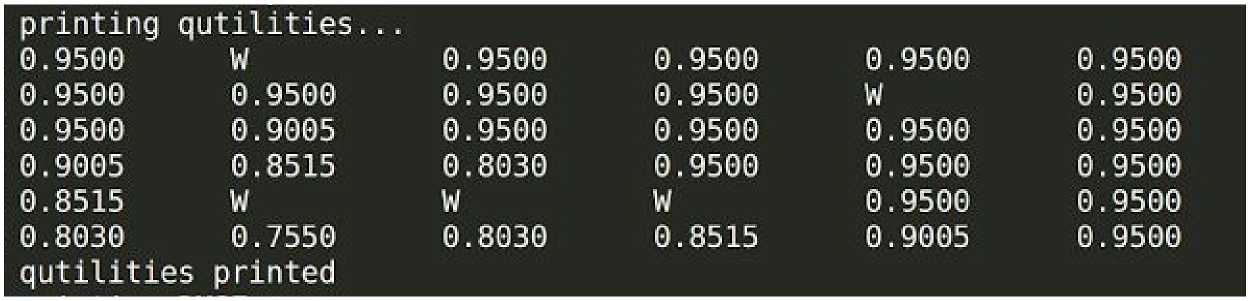
Figure 9: The Temporal Difference Learning Equation

$$Q_{t+1}(s_t,a_t) = \underbrace{Q_t(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t,a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t,a_t)}_{\text{old value}} \right)$$

Figure 10: The Temporal Difference Learning Equation as a function of time step t

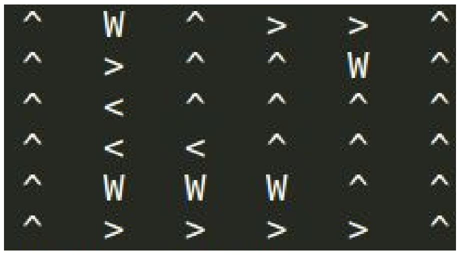
The temporal difference learning equation updates the utility of a cell based on its value from the previous time step $Q_t(s, a)$, the learning rate α, the reward function $R(s)$, the discount factor γ, and the estimate of the optimal future value characterized by the action-utility function that tells us the value of doing action a' in state s' which is $Q(s', a')$. Each successive iteration of TD learning creates a lesser and lesser contribution towards the ultimate utility due to the learning rate α.

Much like value iteration, we iterate through the entire grid by each cell and find the intended direction of that state according to the action a' that corresponds to the maximum contribution at a given time step. Each call to TD learning takes constant time, as the algorithm analyzes a maximum of 16 cells, each referring to the action a' that maximizes $Q(s', a')$ in state s', the state corresponding to the intended direction (see "Notes for the Reader" above part 1.1).



```
printing qutilities...
0.9500      W           0.9500      0.9500      0.9500      0.9500
0.9500      0.9500      0.9500      0.9500      W           0.9500
0.9500      0.9005      0.9500      0.9500      0.9500      0.9500
0.9005      0.8515      0.8030      0.9500      0.9500      0.9500
0.8515      W           W           W           0.9500      0.9500
0.8030      0.7550      0.8030      0.8515      0.9005      0.9500
qutilities printed
```

Figure 11: The resulting utilities from TD Learning after 10,000 iterations



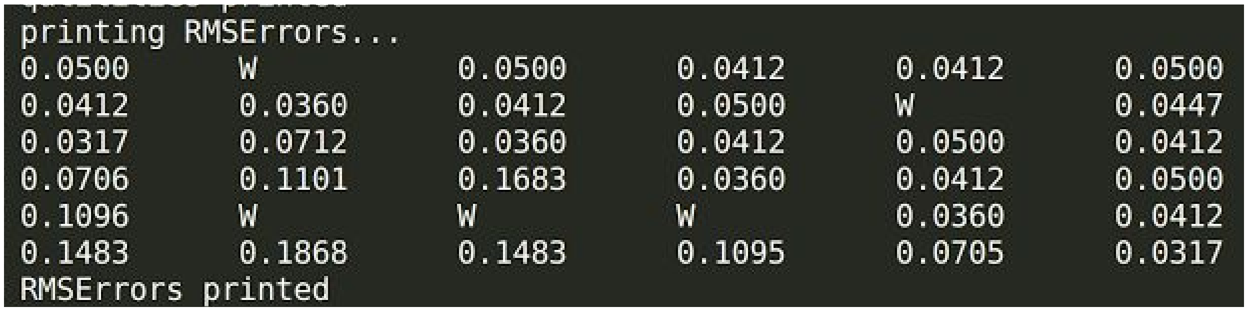
```
^   W   ^   >   >   ^
^   >   ^   ^   W   ^
^   <   ^   ^   ^   ^
^   <   <   ^   ^   ^
^   W   W   W   ^   ^
^   >   >   >   >   ^
```

Figure 12: The intended directions calculated by TD Learning

$$RMSE(U', U) = \sqrt{\frac{1}{N}\sum_{s}\left(U'(s) - U(s)\right)^2}$$

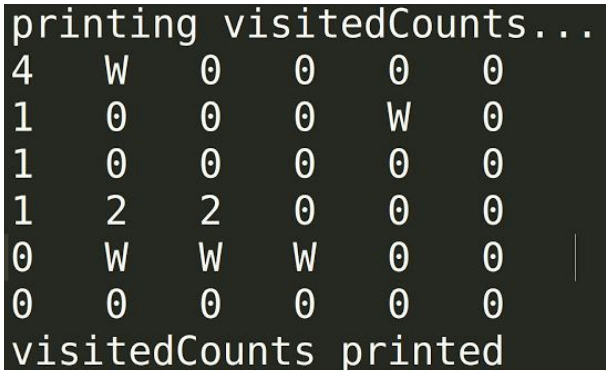
Figure 13: The formula for calculating root mean squared error when comparing the utility U' from TD learning with the "true" utility U obtained by value iteration

The above formula was used to compare the root mean squared errors for each cell. The results are displayed below.



```
printing RMSErrors...
0.0500      W           0.0500      0.0412      0.0412      0.0500
0.0412      0.0360      0.0412      0.0500      W           0.0447
0.0317      0.0712      0.0360      0.0412      0.0500      0.0412
0.0706      0.1101      0.1683      0.0360      0.0412      0.0500
0.1096      W           W           W           0.0360      0.0412
0.1483      0.1868      0.1483      0.1095      0.0705      0.0317
RMSErrors printed
```

Figure 14: The resulting RMS error values from TD Learning after 10,000 iterations of the TD-learning function



```
printing visitedCounts...
4   W   0   0   0   0
1   0   0   0   W   0
1   0   0   0   0   0
1   2   2   0   0   0
0   W   W   W   0   0
0   0   0   0   0   0
visitedCounts printed
```

Figure 15: Ideal path from start after 10 moves in addition to start placement (counts as move) with TD learning

The above figure represents an ideal path for the agent starting at [3,2] using TD learning (see Figures 9 and 10). Notice that once the agent reaches [0,0], every successive move will grant the agent a utility of +1, due to the fact that [0,0] is adjacent to the borders of the grid and a wall, and that +1 is the fixed reward for [0,0]. +1 is also the optimistic reward estimate R^+.

Part 1.1: Extra Credit

This tricky maze has far more states (5x18) and a complex environment (many walls and narrow paths) than the default grid.

Convergence count at each state for value iteration of the **originial grid**:
0 , 336 , 1304 , 344 , 392 , 48 , 224 , 1404 , 624 , 1300 , 184 , 52 , 572 , 2408 , 9652 , 2000 , 2004 , 4 , 904 , 5828 , 4324 , 12164 , 1700 , 520 , 628 , 1420 , 9272 , 4304 , 9016 , 1864 , 656 , 1476 , 1676 , 8684 , 1984 , 4376

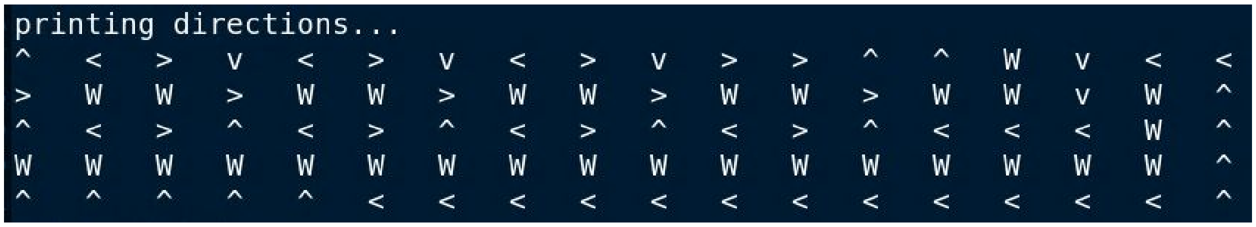
As opposed to the complex tricky grid:

0 , 72 , 80 , 72 , 0 , 88 , 160 , 192 , 108 , 36 , 256 , 304 , 184 , 192 , 72 , 240 , 288 , 384 , 344 , 72 , 552 , 320 , 456 , 232 , 72 , 224 , 544 , 232 , 256 , 72 , 440 , 144 , 456 , 352 , 176 , 472 , 352 , 472 , 272 , 104 , 248 , 496 , 240 , 224 , 152 , 384 , 144 , 456 , 336 , 48 , 472 , 344 , 472 , 176 , 56 , 240 , 496 , 240 , 192 , 8 , 376 , 256 , 456 , 288 , 8 , 12 , 180 , 456 , 168 , 0 , 12 , 224 , 224 , 168 , 0 , 64 , 64 , 224 , 56 , 0 , 8 , 64 , 56 , 0 , 0 , 8 , 0 , 0 , 0 , 0

With the same convergence threshold, but a much more complex grid, the algorithm converges much faster. This is because there are much more restrictions happening in many areas of the grid (where there are narrow 1-cell wide paths of straight lines).

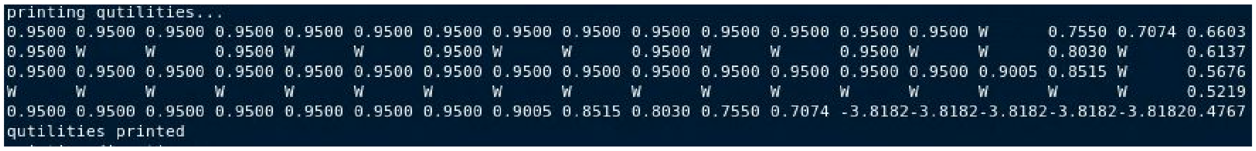
The utility of each state and policy generated by value iteration is as follows (utilities printed in a rable to save vertical space):

(0 , 0): 1.0	(5 , 0): 0.9801	(10 , 0): 0.9801	(15 , 2): 0.96059601
(0 , 1): 1.0	(5 , 2): 0.9801	(10 , 2): 0.9801	(15 , 4): 0.895338254259
(0 , 2): 0.99	(5 , 4): 0.99	(10 , 4): 0.941480149401	(16 , 0): 0.932065347907
(0 , 4): 1.0	(6 , 0): 0.99	(11 , 0): 0.99	(16 , 4): 0.886384871716
(1 , 0): 0.99	(6 , 1): 1.0	(11 , 2): 0.9801	(17 , 0): 0.922744694428
(1 , 2): 0.9801	(6 , 2): 0.99	(11 , 4): 0.932065347907	(17 , 1): 0.913517247484
(1 , 4): 1.0	(6 , 4): 0.9801	(12 , 0): 1.0	(17 , 2): 0.904382075009
(2 , 0): 0.9801	(7 , 0): 0.9801	(12 , 1): 1.0	(17 , 3): 0.895338254259
(2 , 2): 0.9801	(7 , 2): 0.9801	(12 , 2): 0.99	(17 , 4): 0.886384871716
(2 , 4): 1.0	(7 , 4): 0.970299	(12 , 4): 0.922744694428	
(3 , 0): 0.99	(8 , 0): 0.9801	(13 , 0): 1.0	
(3 , 1): 1.0	(8 , 2): 0.9801	(13 , 2): 0.9801	
(3 , 2): 0.99	(8 , 4): 0.96059601	(13 , 4): 0.913517247484	
(3 , 4): 1.0	(9 , 0): 0.99	(14 , 2): 0.970299	
(4 , 0): 0.9801	(9 , 1): 1.0	(14 , 4): 0.904382075009	
(4 , 2): 0.9801	(9 , 2): 0.99	(15 , 0): 0.941480149401	
(4 , 4): 1.0	(9 , 4): 0.9509900499	(15 , 1): 0.9509900499	



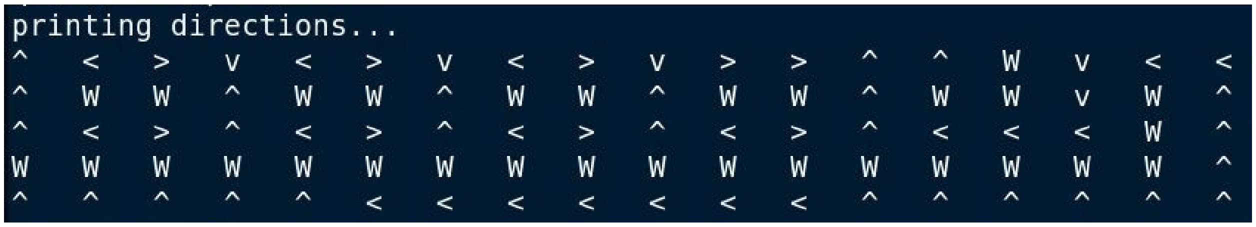
```
printing directions...
^   <   >   v   <   >   v   <   >   v   >   >   ^   ^   W   v   <   <
>   W   W   >   W   W   >   W   W   >   W   W   >   W   W   v   W   ^
^   <   >   ^   <   >   ^   <   >   ^   <   >   ^   <   <   <   W   ^
W   W   W   W   W   W   W   W   W   W   W   W   W   W   W   W   W   ^
^   ^   ^   ^   ^   <   <   <   <   <   <   <   <   <   <   <   <   ^
```

TD Learning Q’s:



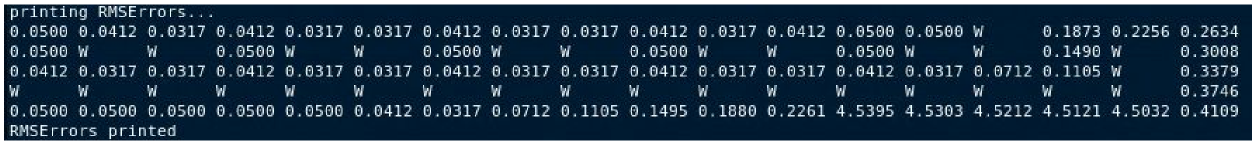
```
printing qutilities...
0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 W      0.7550 0.7074 0.6603
0.9500 W      W      0.9500 W      W      0.9500 W      W      0.9500 W      W      0.9500 W      W      0.8030 W      0.6137
0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9005 0.8515 W      0.5676
W      W      W      W      W      W      W      W      W      W      W      W      W      W      W      W      W      0.5219
0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9500 0.9005 0.8515 0.8030 0.7550 0.7074 -3.8182-3.8182-3.8182-3.8182-3.81820.4767
qutilities printed
```

TD Learning Intended directions:



```
printing directions...
^   <   >   v   <   >   v   <   >   v   >   >   ^   ^   W   v   <   <
^   W   W   ^   W   W   ^   W   W   ^   W   W   ^   W   W   v   W   ^
^   <   >   ^   <   >   ^   <   >   ^   <   >   ^   <   <   <   W   ^
W   W   W   W   W   W   W   W   W   W   W   W   W   W   W   W   W   ^
^   ^   ^   ^   ^   <   <   <   <   <   <   <   ^   ^   ^   ^   ^   ^
```

TD Learning RMSE Errors:

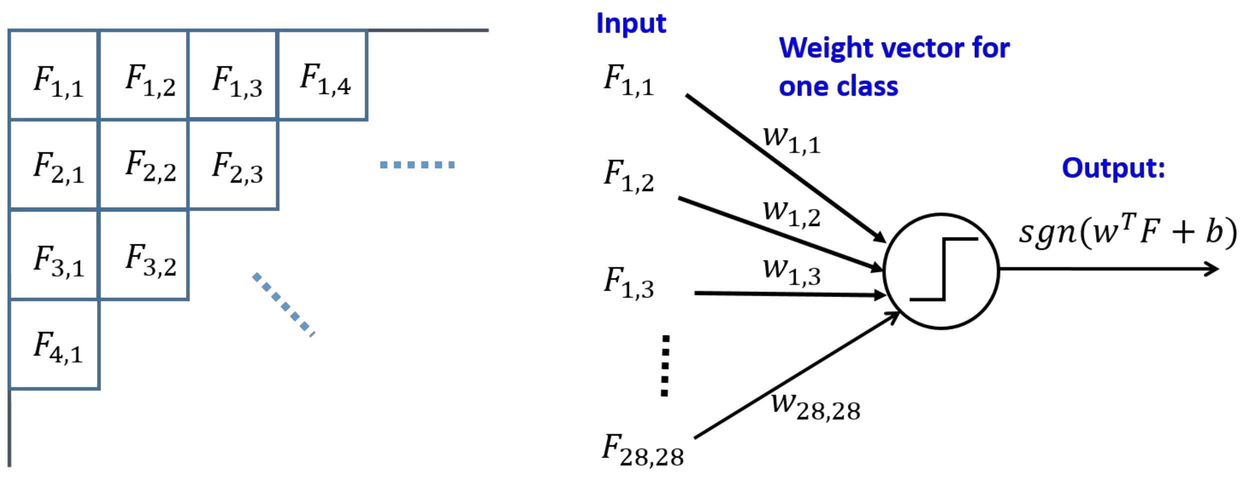


```
printing RMSErrors...
0.0500 0.0412 0.0317 0.0412 0.0317 0.0317 0.0412 0.0317 0.0317 0.0412 0.0317 0.0412 0.0500 0.0500 W      0.1873 0.2256 0.2634
0.0500 W      W      0.0500 W      W      0.0500 W      W      0.0500 W      W      0.0500 W      W      0.1490 W      0.3008
0.0412 0.0317 0.0317 0.0412 0.0317 0.0317 0.0412 0.0317 0.0317 0.0412 0.0317 0.0317 0.0412 0.0317 0.0712 0.1105 W      0.3379
W      W      W      W      W      W      W      W      W      W      W      W      W      W      W      W      W      0.3746
0.0500 0.0500 0.0500 0.0500 0.0500 0.0412 0.0317 0.0712 0.1105 0.1495 0.1880 0.2261 4.5395 4.5303 4.5212 4.5121 4.5032 0.4109
RMSErrors printed
```

Part 2.1.1: Digit/Text Classification with Perceptrons

We used a multi-class (non-differentiable) perceptron learning rule to the digit classification problem described in mp3. Each image is 28x28 and represents a digit, whose foreground values are 1 and background values are 0. A vector for each of the training images was created and put into a training list. There are 5000 training vectors of length 784 in this training list, each corresponding to their respective training image.

Each test image was treated similarly, with foreground values 1 and background values 0. There are 1000 test vectors of length 784 in this test list, each corresponding to their respective test image.



Each epoch or iteration classifies the test vector corresponding to a test image to its appropriate class. Weights are either initially set to all 0 or to a number between 0 and 1 and inserted into a weight vector. The decision rule is as follows:

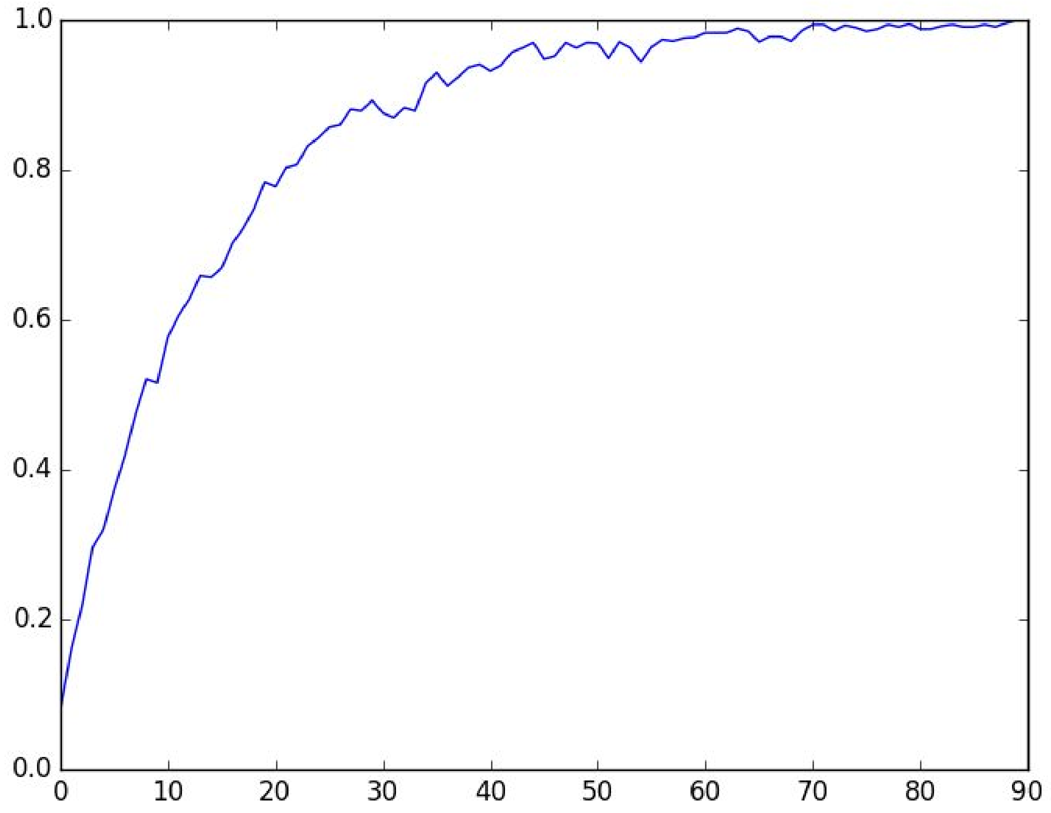
$c' = argmax_c \ (w_c \bullet x)$
where w_c is the weight vector for class c and

Mismatch: If an example from c gets misclassified as c' is the "best" class
$w_c = w_c + \alpha x$
$w_{c'} = w_{c'} - \alpha x$
where $\alpha = \frac{1000}{1000+t}$ fis the learning rate function for iteration t

Weights initially 0
Output Z (weights initially 0)
reached 7.9% accuracy at epoch 0
reached 57.7% accuracy at epoch 10
reached 77.8% accuracy at epoch 20
reached 87.6% accuracy at epoch 30
reached 93.2% accuracy at epoch 40
reached 96.9% accuracy at epoch 50
reached 98.3% accuracy at epoch 60
reached 99.4% accuracy at epoch 70
reached 98.8% accuracy at epoch 80

Overall accuracy: 98.8%

Curve for Z



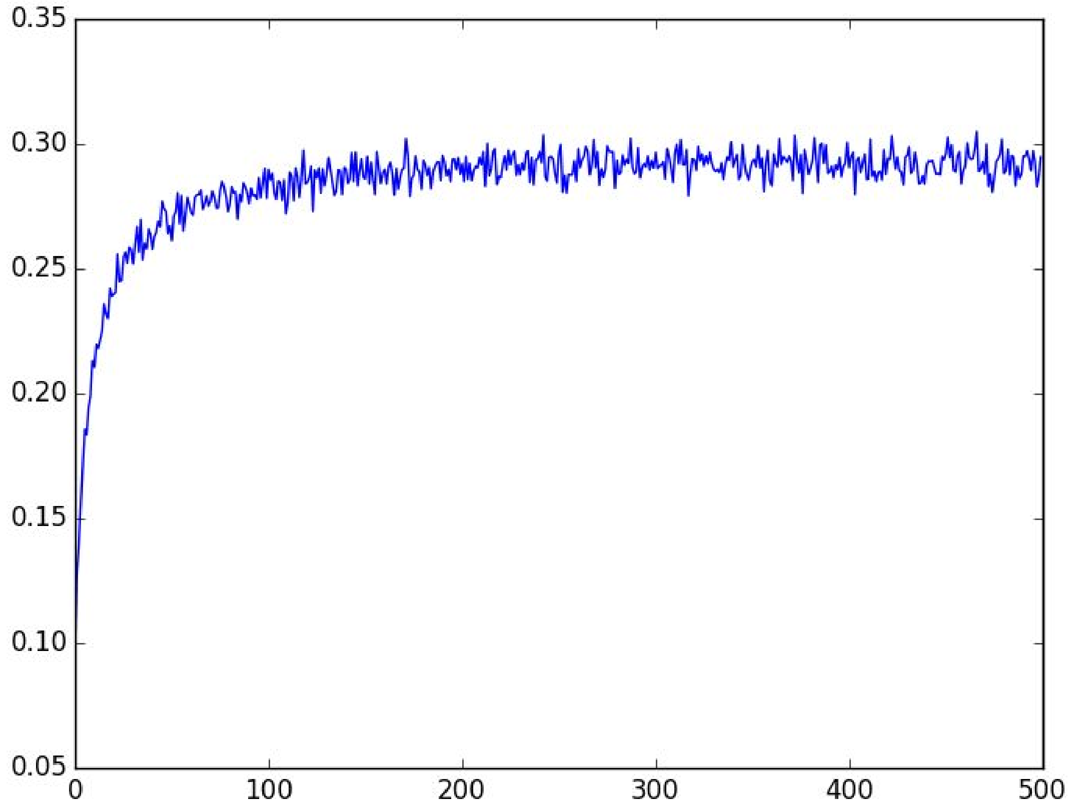
Confusion matrix for Z
0.0778, 0.1667, 0.0444, 0.0444, 0.1222, 0.0777, 0.1222, 0.1555, 0.1333, 0.0555
0.1481, 0.1204, 0.1111, 0.1019, 0.1204, 0.0833, 0.0648, 0.1111, 0.0833, 0.0555
0.1262, 0.0485, 0.1553, 0.0583, 0.1456, 0.0874, 0.0583, 0.1942, 0.0777, 0.0485
0.1600, 0.0899, 0.1799, 0.1100, 0.0899, 0.0400, 0.0500, 0.2000, 0.0500, 0.0299
0.0841, 0.0935, 0.1495, 0.1121, 0.0841, 0.0561, 0.0748, 0.1121, 0.1495, 0.0841
0.0652, 0.1413, 0.1522, 0.0870, 0.1630, 0.1087, 0.0543, 0.1739, 0.0435, 0.0109
0.0110, 0.1429, 0.1099, 0.1099, 0.0659, 0.1209, 0.0989, 0.2198, 0.0879, 0.0330
0.1038, 0.1132, 0.0755, 0.0755, 0.1132, 0.0755, 0.0849, 0.1792, 0.1321, 0.0472
0.1553, 0.1845, 0.0777, 0.0194, 0.0971, 0.0777, 0.1068, 0.1553, 0.0680, 0.0583
0.0600, 0.0700, 0.1700, 0.0700, 0.1100, 0.1000, 0.0800, 0.1900, 0.0700, 0.0800

Weights initially random
Output R (weights initially random)
reached 9.58% accuracy at epoch 0
reached 21.06% accuracy at epoch 10
reached 23.98% accuracy at epoch 20

```
reached 25.2% accuracy at epoch 30
reached 25.78% accuracy at epoch 40
reached 26.12% accuracy at epoch 50
reached 27.24% accuracy at epoch 60
reached 27.68% accuracy at epoch 70
reached 27.6% accuracy at epoch 80
reached 27.6% accuracy at epoch 90
reached 28.98% accuracy at epoch 100
reached 27.64% accuracy at epoch 110
reached 28.46% accuracy at epoch 120
reached 28.74% accuracy at epoch 130
reached 29.0% accuracy at epoch 140
reached 28.6% accuracy at epoch 150
reached 28.98% accuracy at epoch 160
reached 29.02% accuracy at epoch 170
reached 29.34% accuracy at epoch 180
reached 29.54% accuracy at epoch 190
reached 29.2% accuracy at epoch 200
reached 29.16% accuracy at epoch 210
reached 28.64% accuracy at epoch 220
reached 28.76% accuracy at epoch 230
reached 29.68% accuracy at epoch 240
reached 29.6% accuracy at epoch 250
reached 29.8% accuracy at epoch 260
reached 29.68% accuracy at epoch 270
reached 29.34% accuracy at epoch 280
reached 28.88% accuracy at epoch 290
reached 29.66% accuracy at epoch 300
reached 29.64% accuracy at epoch 310
reached 29.36% accuracy at epoch 320
reached 29.0% accuracy at epoch 330
reached 29.08% accuracy at epoch 340
reached 29.3% accuracy at epoch 350
reached 28.32% accuracy at epoch 360
reached 29.3% accuracy at epoch 370
reached 29.04% accuracy at epoch 380
reached 28.9% accuracy at epoch 390
reached 29.06% accuracy at epoch 400
reached 28.6% accuracy at epoch 410
reached 29.42% accuracy at epoch 420
reached 29.54% accuracy at epoch 430
reached 29.12% accuracy at epoch 440
reached 29.64% accuracy at epoch 450
reached 29.12% accuracy at epoch 460
reached 28.8% accuracy at epoch 470
reached 28.82% accuracy at epoch 480
reached 29.52% accuracy at epoch 490
```

Overall accuracy: 29.52%

Curve for R



Confusion matrix for R

```
 0.0777, 0.1111, 0.0888, 0.1555, 0.0222, 0.1000, 0.0333, 0.2666, 0.0555, 0.0888
 0.1851, 0.1018, 0.0462, 0.0370, 0.0277, 0.0833, 0.0277, 0.2777, 0.0925, 0.1203
 0.0776, 0.0873, 0.0776, 0.0970, 0.0388, 0.1165, 0.0388, 0.3398, 0.0291, 0.0970
 0.0800, 0.1700, 0.0899, 0.1000, 0.0200, 0.0500, 0.0700, 0.2600, 0.1000, 0.0599
 0.0934, 0.1214, 0.0373, 0.0934, 0.0280, 0.0747, 0.0467, 0.3644, 0.0373, 0.1028
 0.0652, 0.1086, 0.0652, 0.0760, 0.0217, 0.1304, 0.0652, 0.3478, 0.0217, 0.0978
 0.0769, 0.0769, 0.0879, 0.0989, 0.0109, 0.0989, 0.0109, 0.2417, 0.1098, 0.1868
 0.0377, 0.0943, 0.1603, 0.0660, 0.0094, 0.0754, 0.0566, 0.3301, 0.0660, 0.1037
 0.1165, 0.1553, 0.0485, 0.0776, 0.0388, 0.0679, 0.0485, 0.3398, 0.0388, 0.0679
 0.1200, 0.1200, 0.0800, 0.1300, 0.0100, 0.0700, 0.0899, 0.2200, 0.0500, 0.1100
```

Part 2.1.2: Digit/Text Classification with Perceptrons

Dictionaries are created for each category, where the entries' keys correspond to each unique word in the class, and the entries' values correspond to the number of times they appear each of these classes.

There is a vector created for each training document whose weights are in order of and directly correlate to the frequencies of the words in each training document. That is, the vectors' entries have the most common frequency first. The test documents are transformed into vectors of their own in a similar manner.

The decision rule is as follows (similar to above, but each class corresponds to one of 8 newsgroup categories instead of one of 10 digit classifications):

$c' = argmax_c \ (w_c \bullet x)$

where w_c is the weight vector for class c and

Mismatch: If an example from c gets misclassified as c' is the "best" class

$w_c = w_c + \alpha x$

$w_{c'} = w_{c'} - \alpha x$

where $\alpha = \frac{1000}{1000+t}$ fis the learning rate function for iteration t

The vectors are truncated to the minimum length of the two when taking the dot product. That is, if we were taking the dot product of a vector of length 30 and a vector of length 35, the second vector would be truncated to length of 30. This along with the ordering of the weights of these vectors by decreasing frequency would ensure the optimal dot product is used to make a decision. The results are as follows:

Weights initially 0

```
Output Z (weights initially 0)
reached 7.1% accuracy at epoch 0
reached 52.7% accuracy at epoch 10
reached 76.5% accuracy at epoch 20
reached 79.2% accuracy at epoch 30
reached 84.6% accuracy at epoch 40
reached 87.3% accuracy at epoch 50
reached 88.5% accuracy at epoch 60
reached 89.5% accuracy at epoch 70
reached 89.0% accuracy at epoch 80

Overall accuracy:89.0%
```

Weights initially random

Output R (weights initially random)

```
reached 8.43% accuracy at epoch 0
reached 20.96% accuracy at epoch 10
reached 21.87% accuracy at epoch 20
reached 25.2% accuracy at epoch 30
reached 23.21% accuracy at epoch 40
reached 22.61% accuracy at epoch 50
reached 24.52% accuracy at epoch 60
reached 24.92% accuracy at epoch 70
reached 24.9% accuracy at epoch 80
reached 24.9% accuracy at epoch 90
reached 25.79% accuracy at epoch 100
reached 25.88% accuracy at epoch 110
reached 26.62% accuracy at epoch 120
reached 25.97% accuracy at epoch 130
reached 26.1% accuracy at epoch 140
reached 26.3% accuracy at epoch 150
reached 26.09% accuracy at epoch 160
reached 26.12% accuracy at epoch 170
reached 26.61% accuracy at epoch 180
reached 27.39% accuracy at epoch 190
reached 26.7% accuracy at epoch 200
reached 26.25% accuracy at epoch 210
reached 26.89% accuracy at epoch 220
reached 26.89% accuracy at epoch 230
reached 26.90% accuracy at epoch 240
reached 27.3% accuracy at epoch 250
reached 26.9% accuracy at epoch 260
reached 26.72% accuracy at epoch 270
reached 26.41% accuracy at epoch 280
reached 26.00% accuracy at epoch 290
reached 26.70% accuracy at epoch 300
reached 26.64% accuracy at epoch 310
reached 26.6% accuracy at epoch 320
reached 27.0% accuracy at epoch 330
reached 27.08% accuracy at epoch 340
reached 27.3% accuracy at epoch 350
reached 26.32% accuracy at epoch 360
reached 26.3% accuracy at epoch 370
reached 26.04% accuracy at epoch 380
reached 26.9% accuracy at epoch 390
reached 26.06% accuracy at epoch 400
reached 25.8% accuracy at epoch 410
reached 26.88% accuracy at epoch 420
reached 26.59% accuracy at epoch 430
reached 26.81% accuracy at epoch 440
reached 26.68% accuracy at epoch 450

Overall accuracy: 26.68%
```

Individual Contribution:

Jonathan:	Part 1.1 V-I, Part1 extra credit, report
Harrison:	Part 1.1 V-I and TD Learning, report
Gabriella:	Part 2, report