CS440 Assignment 4 Report

Xuan Wang Shuo Feng Yuchen Qian
xwang182 sfeng15 yqian14

Part 1.1 Grid World MDP

In this part, we use the transition model according to the lecture slides.
As a general part, we use a 6*6 vector as our data structure and each objects in the vector is a node(our own data structure). A node contains its utility, the optimal direction, the boolean value of if it could go left, right, down, up or if it’s a wall. Also, initial reward is set for each terminal node(corresponding value on the map) and non-terminal(-0.04).

Here is our pseudocode:

Here's our data structure:

```
 vector< vector > grid; // hold the grid map. Every element in it is a Node class.
 class Node {
         bool wall; // true if it's a wall
         bool left ,right, up, down; // can go to left,right,up,down from this point
         int visited, inlist; // for bfs, if it's visited,has been put into list.
         int x,y; //save the coordinate
         float reward_initial; //constant, initialized with the rewards from map
         float utility;
         char optimal_direction;
};
```

Algorthm:

First initialize all the grid and the Nodes in it with proper information.

```
int main():
     int BFSunchanged=0;
     For Reward squares are treated as non-terminal states:
         when total 32 Nodes unchanged after BFS, stop loop
     For Reward squares are treated as terminal states:
         when total 25 Nodes unchanged after BFS, stop loop
    while(BFSunchanged != 32 (or25)):
          empty();    //reset the BFS queue, and Node.inlist, Node.visited attributes
          BFSunchanged=BFS();
    print out final grid with optimal direction
```

Run one time of BFS on the grid, return how many Node's utility doesn't change in this turn.

```
int BFS():
    int unchanged=0;
    while(BFS queue is not empty):
        Node current = pop up one Node from BFS queue;
        calculate Node current's new utility and optimal direction;
        put current's neighbours into BFS queue;
        For Reward squares are treated as non-terminal states:
            put current's neighbours into BFS queue regardless of it is a Reward square
        For Reward squares are treated as terminal states:
            put current's neighbours into BFS queue if it is not a Reward square
        unchanged = unchanged + one_node_calculation(current);
    return unchanged;
```

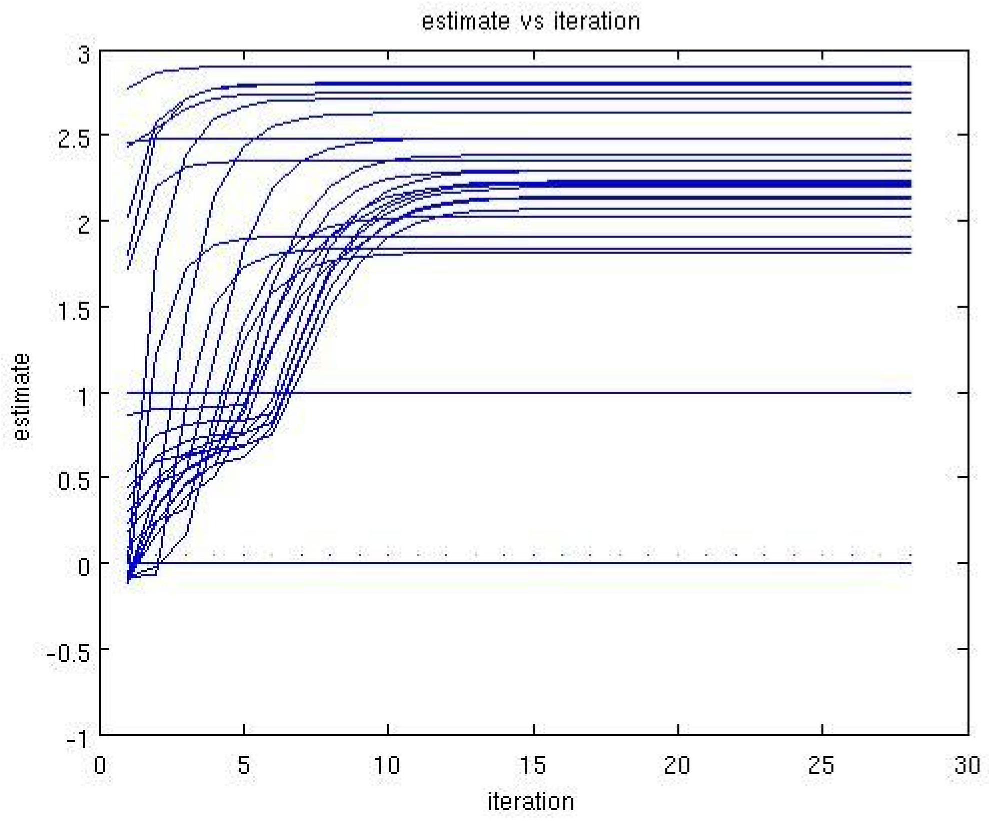
Return whether this Node's utility changed after calculation.

```
one_node_calculation(current node):
    get current node's neighbour Nodes' utility.
    If current can not go to left,  left_utility=current.utility. The other directions are the same.
    Compute the new utility based on four possible directions.
    left_compare=discount_factor*(0.8 * left_utility+0.1*up_utility+0.1*down_utility) - 0.04;
    right_compare, up_compare, down_compare are calculated in the same way;
    update current.utility and current.optimat_direction based on max of the four values;
    return 0/1 based on whether the utility changed;
```

Here is the result:

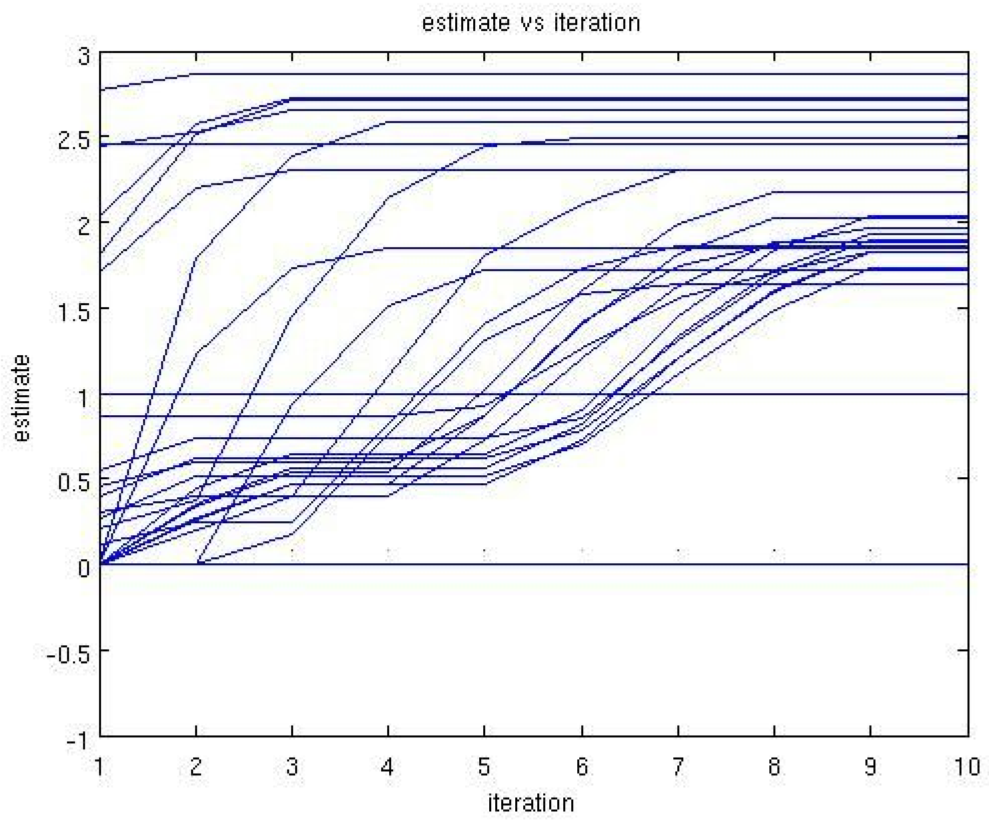
Terminal case

```
1.666   -1      1.812   1.836   1.91    2.348
2.071   2.141   2.21    0.00    -1      2.483
2.139   2.218   2.297   0.00    2.74    4 3
2.197   2.291   2.387   0.00    2.797   2.9
2.132   2.231   2.479   2.629   2.713   2.803
1       -1      2.025   0.00    -1      -1
v  .  >  >  >  v
v  v  v  .  .  v
>> v  .  v  .
>> v  .  >  ^
^^  >> ^  ^
. .  ^  .  .  .
```


estimate vs iteration
estimate
iteration
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
5
10
15
20
25
30

Non-Terminal case

```
1.34     -1       1.64     1.72     1.85   2.3
1.73     1.82     1.88    0.00     -1       2.46
1.82     1.93     2.02    0.00    2.66      3
1.89     2.03     2.17    0.00   2.72   2.86
1.82     1.96      2.3     2.49   2.59    2.71
1           -1       1.86    0.00     -1       -1
v v >  > >  v
v v v  .  >  v
> v v  .   v  >
> > v  .  >   ^
^  ^  > >  ^  ^
^   ^ ^  .   ^  ^
```



Part 1.2 Grid World Reinforcement Learning

In this part, we implemented TD Q-learning as mentioned in the lecture.
We used the same data structure as in the part 1.1, but we added several properties for the node object. We also set our exploration function. When a node has been visited less than N times(we manually choose our N), then we set the utility to a specific value(manually choose). Otherwise we use another value. We did a lot of test to determine these values.

Here's our data structure:

```
vector< vector<Node> > grid;     hold the maze. Every element in it is a Node class.
class Node {
            bool wall;                          // true if it's a wall
            bool left,right,up,down;       // can go to left,right,up,down from this point
            float utility;
            char optimal_direction;
            int x,y;                              //save the coordinate
            float reward_initial;

            float leftUtility,rightUtility,upUtility,downUtility; //the four directions'  sub utility
            int leftN, rightN, upN,downN; // record the time go left from this Node

      float maxUtility(){
            return Max(leftUtility, rightUtility, upUtility, downUtility);
      }
};
int maxStep = 10000;    //max step
int position_x = 3;
int position_y = 1;
```

save the current node or next node going to visit. (3,1) is the start point

Algorithm:
First initialize all the grid and the Nodes in it with proper information.

```
int main():
   int count=0;
   while(count < maxStep){ // loop for maxStep times, means it go maxStep steps
         count=count+1;
         one_step_direction();
    }
    print Grid with direction();
```

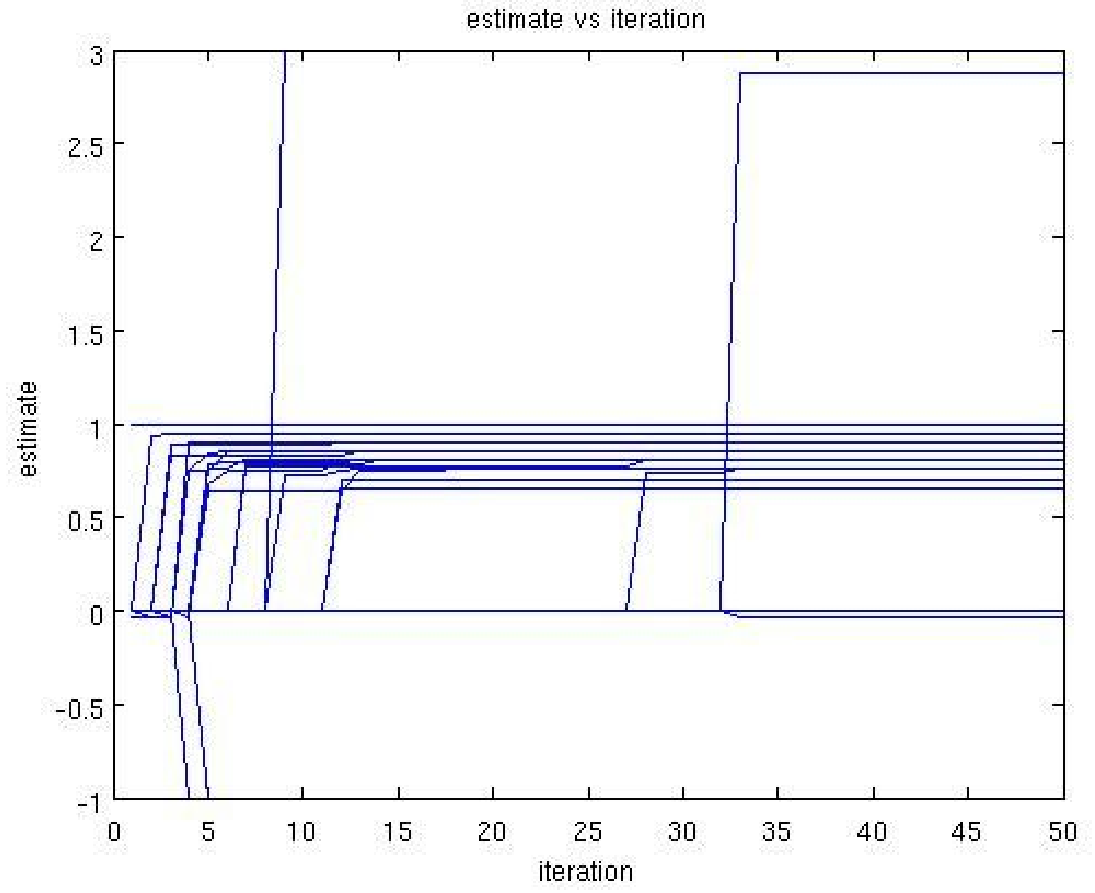
```
based on the subUtilities, this function will get the direction going to
one_step_direction():
    int x=position_x, y=position_y
    if (x,y) is a reward square:
        go back to start point (3,1),
        position_x=3, position_y=1
        return;
    float max_direction=getMax(grid[x][y]'s leftUtility,rightUtility,upUtility,downUtility)
    get direction based on max_direction
    follow_direction(x,y,direction)

based on the direction, this function will update the subUtilities and subNs
follow_direction(x,y,direction):
    int next_x, next_y;
    x,y is the current position, next_x,next_y is the next point position.
    get the next point we are going to based on the direction and current direction
    if direction is up but it can not go up at (x,y),then it hold on next_x=x, next_y=y
         (other direction is the same)

    update grid[x][y]'s one subUtility and subN (leftN/rightNâ¦)
     if the direction is ââup":  (if it is other direction, do it in the same way)
           grid[x][y].upN = grid[x][y].upN+1;
          grid[x][y].upUtility = grid[x][y].upUtility +
         60/(59+grid[x][y].upUtility)*Ã(grid[x][y].reward_initial+discount_factor*grid[next_x][next_y].
         maxUtility()-grid[x][y].upUtility);
          the above calculation is based on lecture formula
          grid[x][y].reward_initial is +3, +1,-1 based on the map. discount_factor is 0.99
    update position_x,position_y, let them hold the next point's coordinate
    position_x=next_y, postion_y=next_y
```

since position_x,position_y is global variable, one_step_direction() next time will use the new coordinate to compute the new direction. In this way, from the loop in main function, this two functions one_step_direction() and follow_direction(x,y,direction) call each other and go to every point.

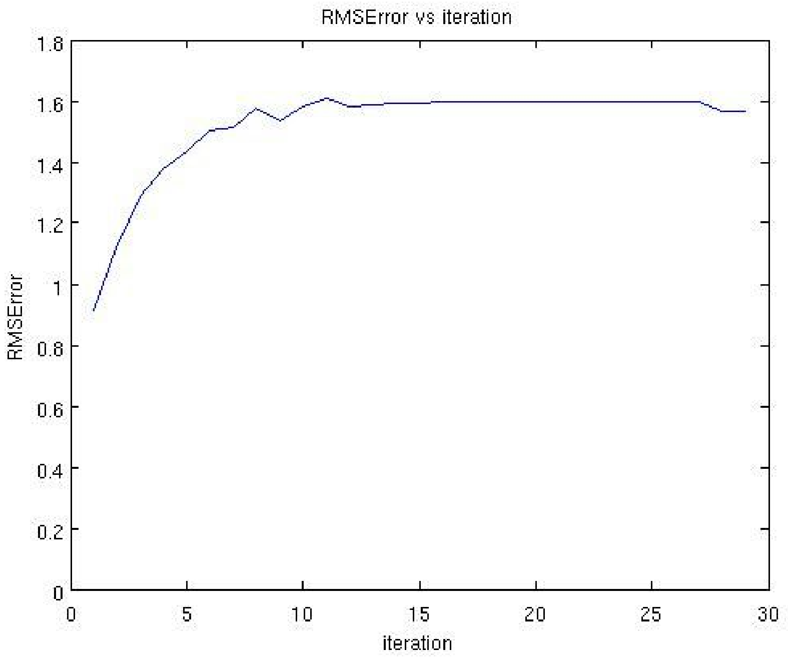
Here is our result:

direction:

```
v < > > > v
> > ^ . < v
> > v . > <
> > v . > ^
> > > > > ^
< < ^ . < <
```



Here is the rms error:



Part 2.1 Digit classification with perceptrons

In this part, we applied the multiclass perceptron learning rule as mentioned in the lecture to the digit classification problem from assignment 3.
We use our own data structure in this part.

Here's our data structure:

```
int imageArray[28][28]            //used to save one picture, every element is 1 or 0 based on the pixel
float weightArray[10][28][28]    //used to save all the weight in the classification.
int t=0                                      // used to save the step(used in the step function)
```

Algorithm:

First we initialize all the variable and structures.

Training:

```
read every picture in file:
      save in imageArray[28][28];
      int label= get the corresponding label from the label file
      t=t+1;
      compute the hidden layer value saved in hidden_layber[10]
      for(int t=0;t<10;t++)
             hidden_layber[t] = hidden_layber[t] + imageArray[i][j] * weightArray[t][i][j];
       get the estimate result
       int estimate=getMax(hidden_layber);
       compare with training label to adjust the weight when estimate is incorrect
        if(estimate==label) return;
        if(estimate!=label){
           weightArray[label][i][j] = weightArray[label][i][j] + imageArray[i][j] Ã1000/(1000+t);
           weightArray[not_label][i][j] = weightArray[not_label][i][j] - imageArray[i][j] Ã1000/(1000+t);
         }
```

Testing:
Now we get the trained weight.
Do the same task as in training, stop when get the estimate answer at
int estimate=getMax(hidden_layber);

Finally calculate the confusion matrix and the overall accuracy

Here is our result:

After first run:

confusion_matrix in %:

```
67.78   0     5.556      0        0   22.22    3.333   0    1.111      0
0     94.44  1.852   0.9259   0  0.9259 0.9259  0   0.9259    0
0        0      84.47   1.942   0      0       7.767   0   3.883   1.942
0        0         0         90      0      5         0       2      3          0
0        0      3.738      0    71.96   0      3.738    0   5.607 14.95
0        0         0      9.783 1.087 85.87     0       0   3.261      0
0      2.198  3.297       0    2.198 8.791 78.02    0    5.495    0
0      3.774 11.32  0.9434 0.9434 0.9434 0  67.92 0.9434 13.21
0         0    2.913    10.68      0        9.709 0  0.9709 72.82 2.913
0         0       2          6         1           5    0       0        1       85
```

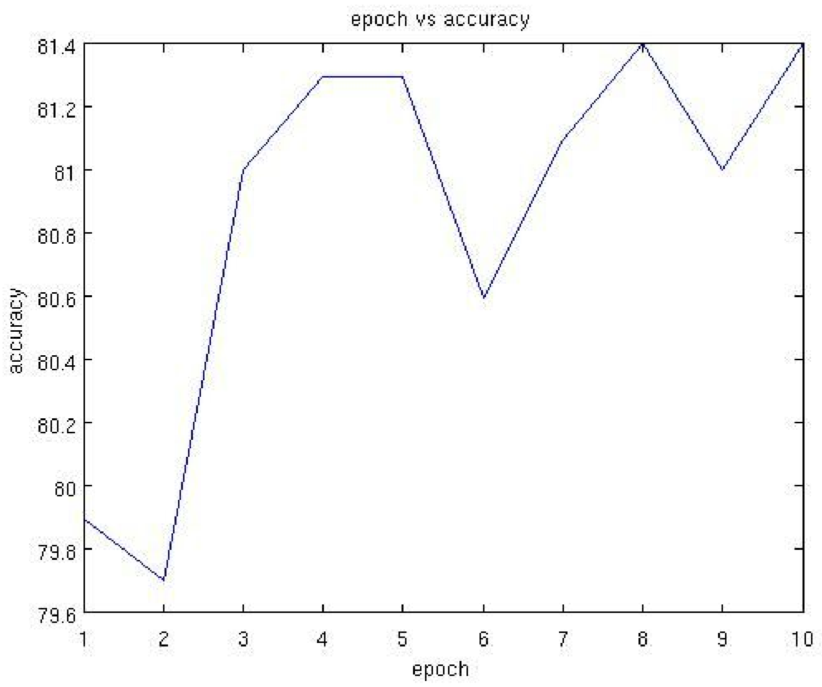
accuracy:

```
   0          1          2          3      4          5           6         7          8         9
67.78% 94.44% 84.47% 90% 71.96% 85.87% 78.02% 67.92% 72.82% 85%
```

overall accuracy:

79.9%, which is better than the naive bayes accuracy(70.5%)

After 10 epoches, here is the graph of epoch vs accuracy:


epoch vs accuracy
accuracy
epoch