Assignment 4: Reinforcement Learning and Deep Learning

Yu Xiao Zhang (yxzhang2)
Ankith Subramanya (ankiths2)
Arnav Sarin (arnav2)

Part 1

First thing we do in act is to transform the representation of the game from environmental space to state space. In environmental space we have the location of the snake head, the location of the snake body parts and the location of the food. Using some conditional statements, we check the location of the snake and its relation with the food and walls to create a state. Then we check if we are training or testing.

If we are in training, we first update the Q-table. Since updating the Q-table is based on the result of the previous action, we skip this part if it’s the first move in a game. To update the table we calculate the reward of the last state-action pair by using the current state. If the current point is more than the previous point, we must have gained a point so our reward is 1. If we are dead in the current state, then that means the last state-action pair lead to a death so our reward is -1. If none of those happen then our reward is -0.1 which encourages our agent to win as fast as possible. Next we calculate alpha using the given formula. C/(C+N(s,a)). Next we update the Q-table by plugging in the reward, alpha and the given gamma in the given formula, $Q(s,a) \leftarrow Q(s,a)+\alpha(R(s)+\gamma \max_{a'} Q(s',a')-Q(s,a))$. Next we calculate the next action by taking the argmax of the f function that we are given. This f function takes in the Q-table, the N-table and a given Ne (optimistic reward estimate). In the first move, the N-table is filled with zeros. Then, if we are not dead, we update the N-table by adding one to the state-action pair and saving the current state, action and points for access in the next move. Finally, we reset the previous state, action and points if we are dead to signify a new game.

If we are in testing, we simply return argmax over the actions of the Q-table element with our current state. This means we take the action that will maximize the expected reward.

Part 1.2

The training parameters we believed are the best after trial and error were Ne=20, and fixed alpha=0.3. The training convergence time is 10000 games in 26.3 seconds and average score in 1000 test games is 26.45 points.

Part 1.3

I added a new state that determined if the food was on the diagonal of the snake head. It was 5 values, 1 if the snake head was on the +x +y diagonal, 2 for +x -y, 3 for -x +y 4 for -x -y and 0 for not on a diagonal. We had an average of 24.58 points for 1000 games. These changes are reasonable because if the food is on a diagonal, the snake can sometimes be more efficient by moving diagonally which can minimize trapping it self. The positive effects are that it more often avoids directly trapping itself by moving diagonally but negative effects are that it leaves a diagonal tail which can lead to blocking paths later on.

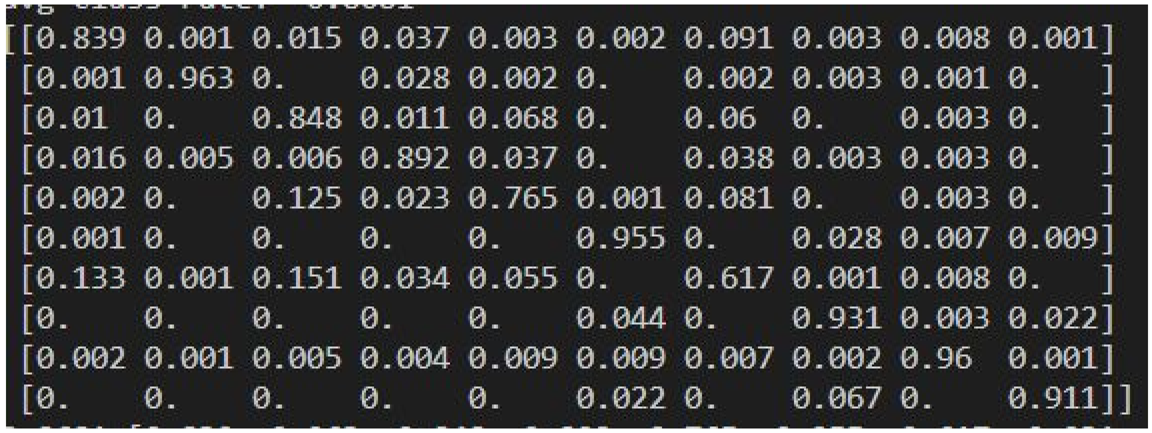
Part 2

1. In the first iteration of our code, we used nested for loops to implement affine_forward, affine_backward, relu_forward and relu_backward. In order to make this more optimal, we used matrix multiplication (np.matmul) in affine_forward and affine_backward, np.maximum in relu_forward and np.where in relu_backward.

   Additionally, we tried using numpy functions as much as possible for most of our math operations   (Math.exp -> np.exp). Since numpy is written in C, this ensures that our code is optimal.

2.
i) ***10 epochs***

Confusion Matrix



```
[[0.839 0.001 0.015 0.037 0.003 0.002 0.091 0.003 0.008 0.001]
 [0.001 0.963 0.    0.028 0.002 0.    0.002 0.003 0.001 0.   ]
 [0.01  0.    0.848 0.011 0.068 0.    0.06  0.    0.003 0.   ]
 [0.016 0.005 0.006 0.892 0.037 0.    0.038 0.003 0.003 0.   ]
 [0.002 0.    0.125 0.023 0.765 0.001 0.081 0.    0.003 0.   ]
 [0.001 0.    0.    0.    0.    0.955 0.    0.028 0.007 0.009]
 [0.133 0.001 0.151 0.034 0.055 0.    0.617 0.001 0.008 0.   ]
 [0.    0.    0.    0.    0.    0.044 0.    0.931 0.003 0.022]
 [0.002 0.001 0.005 0.004 0.009 0.009 0.007 0.002 0.96  0.001]
 [0.    0.    0.    0.    0.    0.022 0.    0.067 0.    0.911]]
```

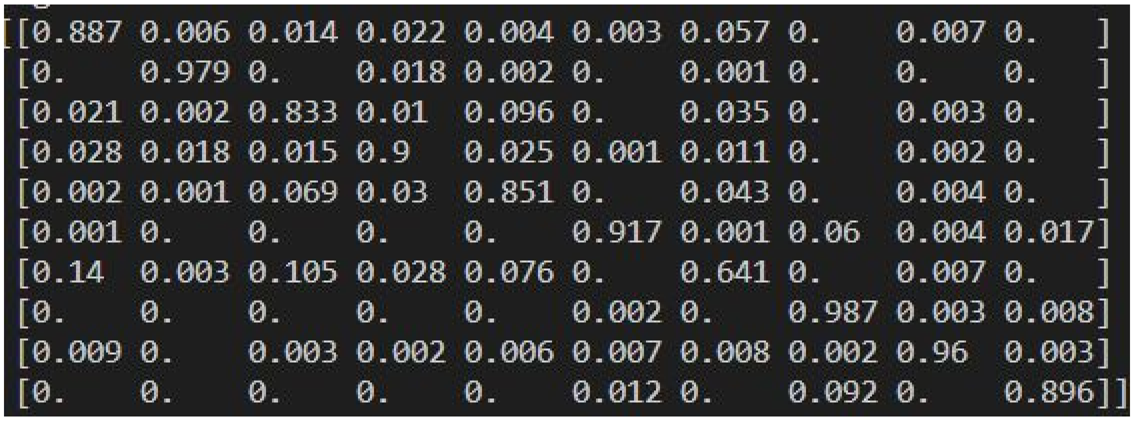
Average classification rate: 0.868

Classification rate per class: [0.839, 0.963, 0.848, 0.892, 0.765, 0.955, 0.617, 0.931, 0.96, 0.911]

Runtime: 5:20

ii) ***30 epochs***

Confusion Matrix



```
[[0.887 0.006 0.014 0.022 0.004 0.003 0.057 0.    0.007 0.   ]
 [0.    0.979 0.    0.018 0.002 0.    0.001 0.    0.    0.   ]
 [0.021 0.002 0.833 0.01  0.096 0.    0.035 0.    0.003 0.   ]
 [0.028 0.018 0.015 0.9   0.025 0.001 0.011 0.    0.002 0.   ]
 [0.002 0.001 0.069 0.03  0.851 0.    0.043 0.    0.004 0.   ]
 [0.001 0.    0.    0.    0.    0.917 0.001 0.06  0.004 0.017]
 [0.14  0.003 0.105 0.028 0.076 0.    0.641 0.    0.007 0.   ]
 [0.    0.    0.    0.    0.    0.002 0.    0.987 0.003 0.008]
 [0.009 0.    0.003 0.002 0.006 0.007 0.008 0.002 0.96  0.003]
 [0.    0.    0.    0.    0.    0.012 0.    0.092 0.    0.896]]
```

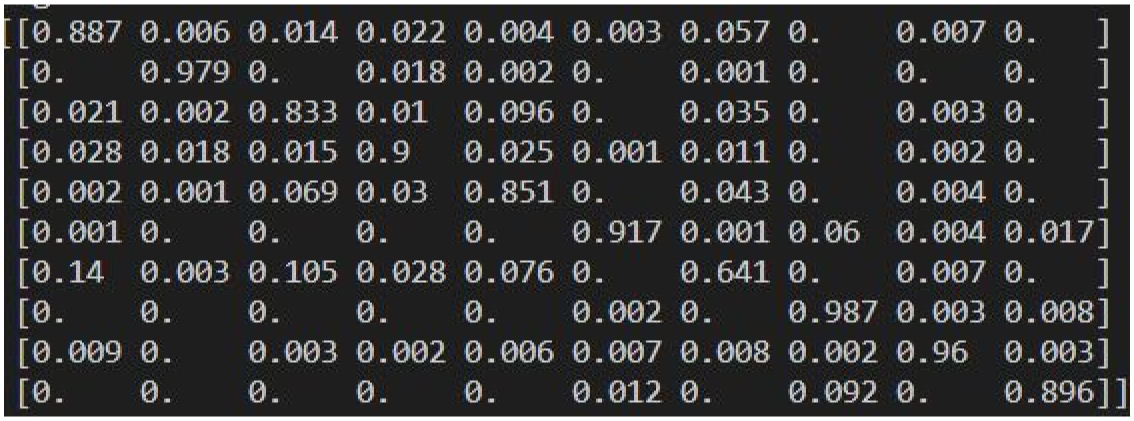
Average classification rate: 0.885

Classification rate per class: [0.887, 0.979, 0.833, 0.9, 0.851, 0.917, 0.641, 0.987, 0.96, 0.896]

Runtime: 15:46

iii) ***30 epochs***

Confusion Matrix



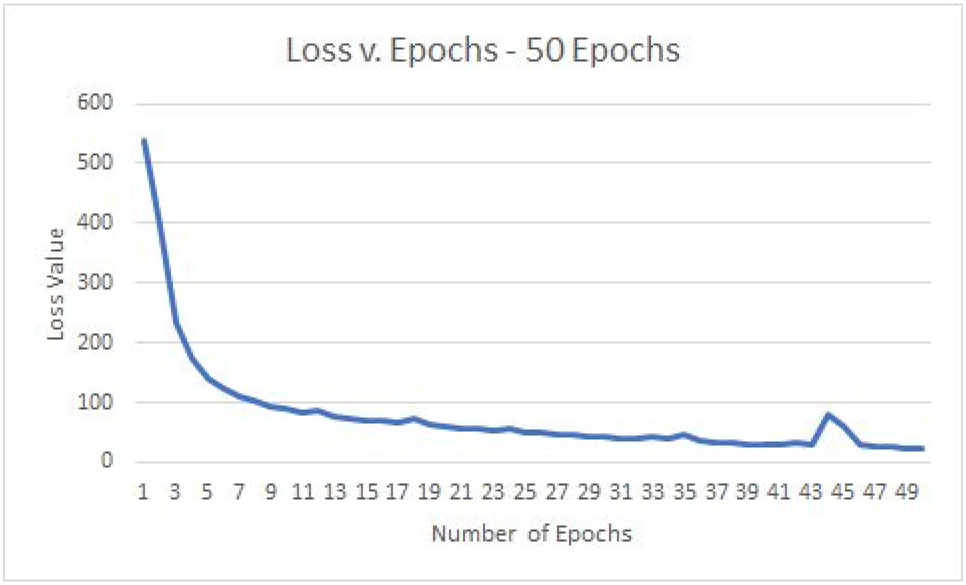
```
[[0.887 0.006 0.014 0.022 0.004 0.003 0.057 0.    0.007 0.   ]
 [0.    0.979 0.    0.018 0.002 0.    0.001 0.    0.    0.   ]
 [0.021 0.002 0.833 0.01  0.096 0.    0.035 0.    0.003 0.   ]
 [0.028 0.018 0.015 0.9   0.025 0.001 0.011 0.    0.002 0.   ]
 [0.002 0.001 0.069 0.03  0.851 0.    0.043 0.    0.004 0.   ]
 [0.001 0.    0.    0.    0.    0.917 0.001 0.06  0.004 0.017]
 [0.14  0.003 0.105 0.028 0.076 0.    0.641 0.    0.007 0.   ]
 [0.    0.    0.    0.    0.    0.002 0.    0.987 0.003 0.008]
 [0.009 0.    0.003 0.002 0.006 0.007 0.008 0.002 0.96  0.003]
 [0.    0.    0.    0.    0.    0.012 0.    0.092 0.    0.896]]
```

Average classification rate: 0.876

Classification rate per class: [0.881, 0.966, 0.825, 0.892, 0.896, 0.94, 0.572, 0.873, 0.949, 0.972]

Runtime 35:34:

3.



4. As you can see from the graph there are bumps where the loss goes up a little bit. This is expected. This behavior occurs because we have a constant learning rate, as the model converges, the learning rate will occasionally over shoot. A better implementation would be having a learning rate that decays as the model converges similar to part 1 of this MP.

Statement of Contribution
Yu Xiao worked on part 1. Yu Xiao, Ankith and Arnav worked on part 2. Everyone worked on the report together.