Henry Solberg (R4) and Ayush Sinha (Q4)
CS 440 MP4
4/20/2019

**Part 1:**

**(1)**
As instructed, we implemented a temporal difference learning algorithm with exploration and Q-learning. At each time step, we update the Q table for the last action taken based on the reward model, and then choose a new action based on the current state. We keep track of how many times each action has been taken in each state, ensuring that we take each action a set number of times (exploration).

During the training phase, the agent prefers to go to the grid cell that hasn't been visited Ne times. If there are more than one possible cells that haven't been visited Ne times, it uses the priority right>left>down>up. If all possible next cells have been 'explored', then the agent chooses the cell with maximum Q value. In case of a tie, it again follows the priority right>left>down>up.

During testing phase, the Q table is not updated and the number of times a state is visited is also not recorded. The agent simply chooses the action with highest Q value. Again, ties are broken using the priority right>left>down>up.

If the snake dies, the state, action and points are reset.

**(2)**
Ne = 20, C = 80 and gamma = 0.5 work best.
First, we kept C and gamma constant and varied Ne. We observed that lowering Ne from the default value of 40 increased average points. Increasing Ne made the performance worse. We believe this happens because we're able to learn a good policy with lesser number of explorations and hence doing additional explorations will give a worse result than exploiting the good policy found already. Ne = 20 worked best for us.
We varied C and gamma similarly and found that we converge to a good policy faster for lower gamma and higher C (or higher alpha). C = 80 and gamma = 0.5 worked well for us.

With the modified parameters, we train for 15000 games and our time for convergence is 45.7 seconds. The average points for 1000 test games is 26.369. (This was the best of several attempts)

**(3)**
We modified the state configuration to account for states when any possible next action would lead the snake head into 'trap'. 'Trap' refers to a cell which is surrounded by snake's body on all 4 sides.
We checked if a cell adjoining the snake_head is not occupied by snake_body, then if this adjoining cell is surrounded by snake_body on all sides, then we make adjoining_body_x = 2, for x = left,right,top or bottom.

This change should help the snake avoid obvious ‘traps’. The average points for this modified state configuration case were 21.485.

The snake will avoid the traps. However, this extends the Q-table and hence the policy might not converge within 10000 training games. Also, the traps might not be frequent enough to affect the average points as well.

Hence, we trained the modified state with 30000 training examples. The average points recorded were 26.201. Notably, the average score continued to increase as 20,000+ training games were played, a phenomenon that was never observed in training with the unmodified state space, even when we “luckily” found a high-scoring agent on the test games.

For comparison, we trained with the unmodified state with 30,000 training examples and achieved an average score on test games of only 22.36. Therefore, we speculate that the additional trap avoidance did contribute to an increased average score.

**Part 2:**

1. We determined that affine_forward, affine_backward, relu_forwrad, and relu_backward were most critical to performance. We were sure to use numpy array operations such as sum, matmul, clip, and argwhere wherever possible in these functions. Our initial implementation of relu_backward did not use np.argwhere and we were able to improve runtime substantially by using np.argwhere to locate 0s in the “Z” matrix. This allowed us to achieve roughly 6s per epoch runtime on our personal computer.
2. Here are the results for 10, 30, and 50 epochs:

<u>Number of epochs: 10</u>

Minibatch_gd run time: 61s (6.1s per epoch)

Confusion matrix:

```
[[0.906 0.002 0.014 0.037 0.009 0.    0.022 0.    0.01  0.   ]
 [0.001 0.974 0.001 0.021 0.003 0.    0.    0.    0.    0.   ]
 [0.027 0.004 0.674 0.013 0.252 0.    0.025 0.    0.005 0.   ]
 [0.028 0.011 0.007 0.917 0.032 0.001 0.002 0.    0.002 0.   ]
 [0.001 0.003 0.033 0.032 0.915 0.    0.011 0.    0.005 0.   ]
 [0.001 0.    0.    0.002 0.    0.929 0.001 0.036 0.006 0.025]
 [0.224 0.005 0.099 0.05  0.26  0.001 0.345 0.    0.016 0.   ]
 [0.    0.    0.    0.    0.    0.022 0.    0.929 0.001 0.048]
 [0.004 0.001 0.002 0.005 0.01  0.014 0.007 0.002 0.953 0.002]
 [0.    0.    0.    0.001 0.    0.011 0.    0.036 0.    0.952]]
```

Average classification rate: 0.8494

<u>Number of epochs: 30</u>

Minibatch_gd run time: 194.6s (6.5s per epoch)

Confusion matrix:

```
[[0.842 0.001 0.008 0.024 0.005 0.    0.111 0.    0.009 0.   ]
 [0.001 0.97  0.002 0.023 0.002 0.    0.001 0.    0.001 0.   ]
 [0.021 0.001 0.773 0.014 0.123 0.    0.062 0.    0.006 0.   ]
 [0.027 0.008 0.005 0.913 0.026 0.001 0.017 0.    0.003 0.   ]
 [0.001 0.001 0.045 0.043 0.873 0.001 0.033 0.    0.003 0.   ]
 [0.002 0.    0.    0.    0.    0.932 0.    0.057 0.003 0.006]
 [0.119 0.001 0.059 0.03  0.097 0.    0.68  0.    0.014 0.   ]
 [0.    0.    0.    0.    0.    0.007 0.    0.99  0.003 0.   ]
 [0.006 0.    0.    0.002 0.003 0.007 0.011 0.004 0.966 0.001]
 [0.    0.    0.    0.    0.    0.02  0.001 0.219 0.    0.76 ]]
```

Average classification rate: 0.8699

Number of epochs: 50

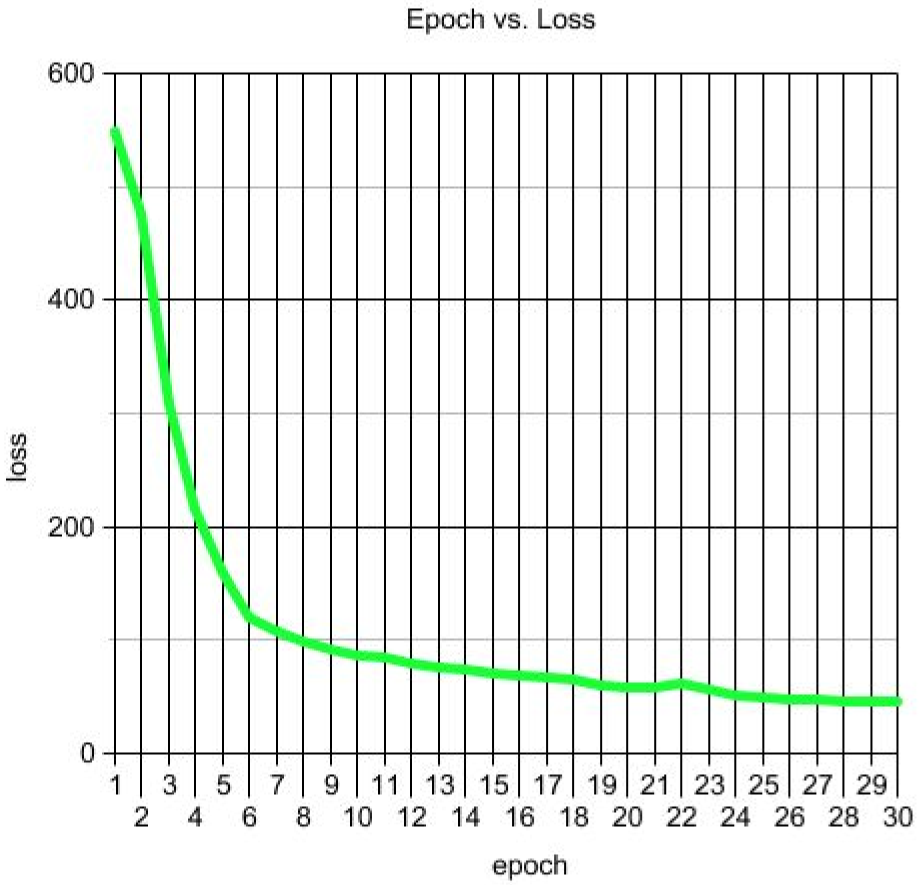
Minibatch_gd run time: 312s (6.2s per epoch)

Confusion matrix:

```
[[0.889 0.003 0.017 0.017 0.003 0.001 0.058 0.    0.012 0.   ]
 [0.003 0.972 0.004 0.016 0.003 0.    0.002 0.    0.    0.   ]
 [0.016 0.002 0.864 0.007 0.057 0.001 0.052 0.    0.001 0.   ]
 [0.031 0.006 0.013 0.915 0.015 0.001 0.019 0.    0.    0.   ]
 [0.004 0.    0.095 0.027 0.792 0.    0.076 0.    0.006 0.   ]
 [0.001 0.    0.001 0.    0.    0.955 0.002 0.028 0.002 0.011]
 [0.137 0.001 0.097 0.032 0.04  0.    0.683 0.    0.01  0.   ]
 [0.    0.    0.    0.    0.    0.012 0.    0.959 0.006 0.023]
 [0.006 0.001 0.008 0.004 0.003 0.004 0.006 0.001 0.966 0.001]
 [0.    0.    0.    0.001 0.    0.015 0.    0.032 0.    0.952]]
```

Average classification rate: 0.8947

3. (created using https://nces.ed.gov/nceskids/createagraph/)



4. We see that loss decays in a non-linear fashion, decreasing much more quickly at first and then leveling out. This is expected since as the neural network becomes a better and better approximation of the true mapping from image to class, the outputs of the last layer of the neural network will become closer to their best values. Since the changes made by back propagation are proportional to the difference between the actual output and the "best" output, we expect less drastic changes as we train our model. We also, of course, expect loss to decrease since we are changing weights to produce output that better matches ideal output.

5. (Extra credit):
Following the tutorial found here: https://www.tensorflow.org/tutorials/estimators/cnn , we created a convolutional neural network using tensorflow. A convolutional neural network takes advantage of the visual/spatial nature of the pixels of an image. Small groups of pixels, of course, simply represent spatial facts about the world in the same way that individual pixels do, so a convolutional neural network combines inputs from pixels with inputs from other nearby pixels in layers of its neural network.

Intuitively, it is unlikely that two adjacent pixels in one part of an image relate to two adjacent pixels in another part of an image in a very different way for each of the two pixels. It is much more likely that these two pairs of pixels interact similarly, and so a convolutional neural network takes advantage of that by parsing images into subregions before passing the input along to fully connected layers (like the ones we implemented in part 2).

Our convolutional neural network created following the tutorial on the tensorflow website had the following layers: convolution, pooling, convolution, pooling, a dense layer, and an output layer. It did not outperform our 4-dense-layer neural network created in part 2. We speculate that more dense layers are needed in order to increase the performance of the convolutional neural network, since more dense layers is the advantage the part 2 neural network has over this convolutional neural network.

Here is the confusion matrix for our cnn's evaluation on the test data:

```
[[828   3     8     65   3     5     78    0    9     1]
 [1     956   9     25   3     0     6     0    0     0]
 [10    1     716   13   150   2     104   0    4     0]
 [25    15    10    891  25    0      28   0    6     0]
 [1     1     66    33   823   1      68   0    7     0]
 [0     0     0     1    0     924   0     54   3     18]
 [165   1     100   47   108   0     567   0    11     1]
 [0     0     0     0    0     39    0     917  3     41]
 [1     1     5     8    5     6     18    5     950   1]
 [0     1     0     1    0     13    0     60   0      925]]
```

*(Values are out of 1000)*

Average accuracy: 0.8497

Accuracy per class: [.828, .956, .716, .891, .823, .924, .567, .917, .950, .925]

**Statement of contribution:**

Ayush focused on part 1, whereas Henry focused on part 2, although both wrote code for both parts. Ayush mainly wrote part 1 of the report, whereas Henry mostly wrote part 2. We submitted Ayush's code for part 1 and Henry's for part 2. Henry did the extra credit.