MP4 Report

- Daoji Zhang (daojiz2) daojiz2@illinois.edu (Part 1.1, Part 1.2, Part1.3 Extra Credit)
- Siti Zhang (sitiz2) sitiz2@illinois.edu (Part 1.1, Part 1.2, Part 1.3 Extra Credit)
- Andre He (songtao2) songtao2@illinois.edu (Part 2.1)

Part 1 Q-Learning (Pong)

Part 1.1 Single-Player Pong

We used the same parameters for both Q-Learning and SARSA agents and simulated for 100k games before it learns a good policy.

The learning rate function: $\alpha(N)=C/(C+N(s,a)),C=40$

The discount factor: $\gamma = 0.7$

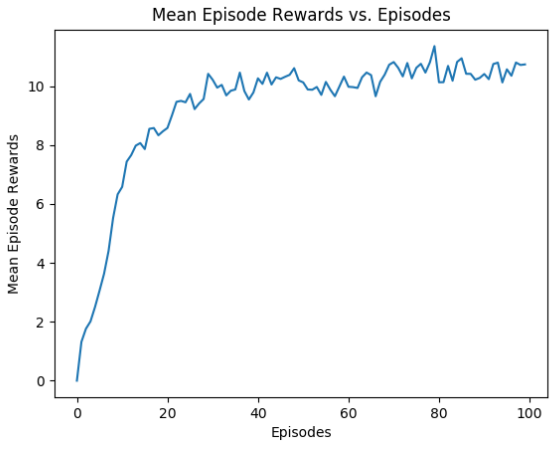
The exploration function: epsilon-greedy strategy, epsilon value = 0.1.

We set up different combination parameters and observe the final result.The above choice of parameters seems to achieve good results in my simulation. The learning rate determines to what extent newly acquired information overrides old information.A value of 0 makes the agent learn nothing, while a value of 1 makes the agent consider only the most recent information. Also, we used the learning rate decay function. With a larger C, α seems to decay slowly so the average bounce times converged more quickly. The discount factor determines the importance of future rewards. A value of 0 means only considering immediate rewards, while a value approaching

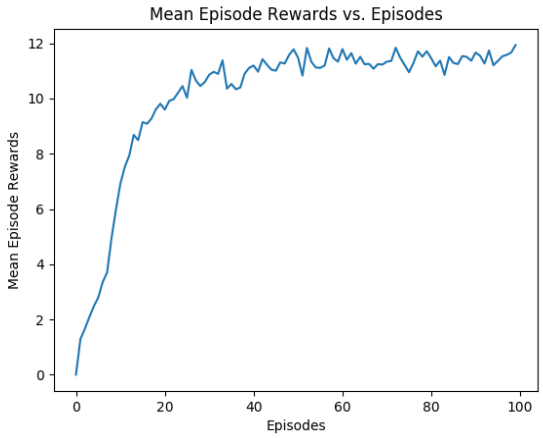
1tends to choose a long-term high reward. If we decrease the discount factor from 1 to almost 0, we can found that the agent may stuck at lower bounces on average. And the exploration strategy is epsilon-greedy,which means pick the current best option ("greedy") most of the time, but pick a random option with a small (epsilon) probability sometimes. If the epsilon value increases, the simulation tends to explore too much instead of learning. So the learning speed is slower and the average bounces times in the same period decreases.

On 200 test games, the average number of times per game that the ball bounces off your paddle is 14.69 (Q-learning), 16.535 (SARSA).Overall, these two agents have similar performances. Based on my code, the training time for a Q-learning agent seems longer than that of a SARSA agent.And for the average bounces times of SARSA seems a little higher than Q learning.

The Mean Episode Rewards vs. Episodes plot for Q-Learning agent.



The Mean Episode Rewards vs. Episodes plot for SARSA agent.



The difference is that SARSA is on policy while Q Learning is off policy.In Q learning, you update the estimate from the maximum estimate of possible next actions, regardless of which action you took. Whilst in SARSA, you update estimates based on and take the same action.

Q Learning:

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

That's not necessarily the action we will take next time...

SARSA:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$
$$s \leftarrow s'$$

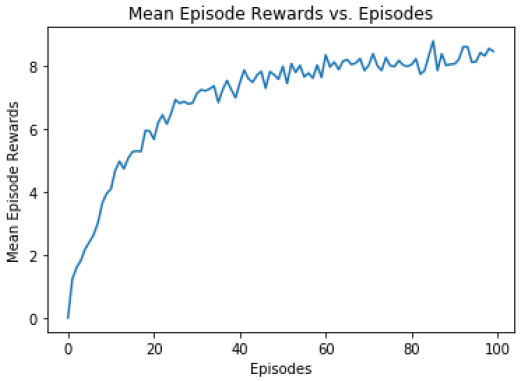
That is the action we will take next time...

Part1.2

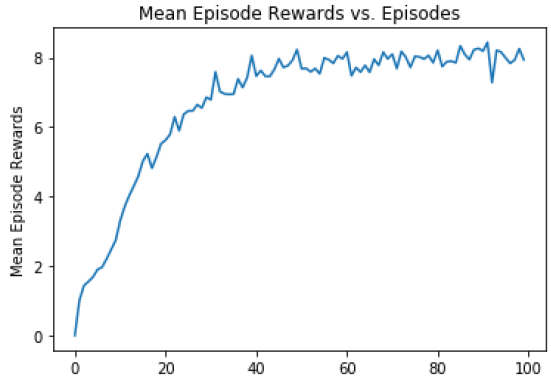
A agent 200 times tests average bounce: 16.83

B agent 200 times tests average bounce: 15.555

The Mean Episode Rewards vs. Episodes plotsplot for A agent:



The Mean Episode Rewards vs. Episodes plotsplot for B agent:



In order to train the agents in the new environment, we modify the reward function, and some part of the state. We change the paddle_x to 0, as shown in the homework page. The reward function is also changed in order to correctly judge whether the paddle successfully bounce the ball. The Part2 in the file shows the implementation of the A agent, it imports the Q-table from Part1 which is the code of Part1.1. It is shown that agent A has a better performance than agent B, and the Q-table converges more quickly in agent A.

Part1.3

This GUI uses the pygame to show the animation of the 200 times test in part1.1. The file also included a 40 seconds video of this GUI shows that the trained agent can bounce the ball greater than 9 times before missing it in these 200 tests on average.

According to the performance of the agent shown in the GUI, this agent can easily defeat a human player. It is shown that the agent can control the paddle to bounce ball every time when the velocity is stable, but as the velocity of the ball increases dramatically (although the speed increment is picked uniformly in the given range, sometimes the ball continue to increase the magnitude of the velocity), the agent will increase the chance of missing the ball. However, the human player always lose the game before the agent misses the ball.

Part 2 Deep Learning:

Part 2.1 Cloning the Behavior of an Expert Player

General Implementation: Expert training using tensorflow neural networks and cross entropy with a three-layer fully-connected network. File: pong_game.py, q_learning.py.

1. **Answer the following question in the report: What is the benefit of using a deep network policy instead of a Q-table (from part 1)? (Hint: think about memory usage and/or what happens when your agent sees a new state that the agent has never seen before).**

Answer: The benefit of using a deep network policy instead of a Q-table is that a deep network training method uses less memory space as it generally needs less time to train a model. Moreover, deep Q-learning works better with a new state that the agent has never seen before.

2. Implement the forward and backwards functions of a neural network and give a brief explanation of implementation and what neural network architecture you used.

Answer: Implementations of forward and backward functions are mainly based on "algorithm 1 Three Layer Network" in assignment4 webpage. We use tensorflow, an open source machine learning module in python, to implement neural network functions. Tensorflow has features supporting neural network setup that we can directly implement without formulating specific functions. For Affine-forward and -backward functions, tf.matmul (same functioning as np.dot) is used to perform a fully-connected layer of network. For ReLU-forward, tf.nn.relu is used directly from tensorflow library. ReLU-backward function is implemented with formula in assignment webpage. It consists of a three-layer fully-connected network and number of units of a single network layer depending on batch size and (if affine network) column number of input weight vector.

```
21  def affine_forward(x, w, b):
22      out = tf.matmul(x, w) + b
23      cache = (x, w, b)
24      return out, cache
25
26  def affine_backward(dout, cache):
27      x, w, b = cache
28      dx, dw, db = None, None, None
29      dx = tf.matmul(dout, w.T)
30      dw = tf.matmul(x.T, dout)
31      db = tf.matmul(dout.T, np.ones(N))
32      return dx, dw, db
33
34  def relu_forward(z):
35      out = tf.nn.relu(z)
36      cache = (z)
37      return out, cache
38
39  def relu_backward(dout, cache):
40      dx, x = None, cache
41      dx = tf.identity(dout)
42      dx[x <= 0] = 0
43      return dx
44
45  def weight_variable(shape):
46      """ Initialize the weight variable."""
47      initial = tf.truncated_normal(shape, stddev=1.)
48      return tf.Variable(initial)
```

3. Train your neural network using minibatch gradient descent. Report the confusion matrix and misclassification error. You should be able to get an accuracy of at least 85%

and probably 95% if you train long enough. Report you network settings including the number of layers, number of units per layer, and learning rate.

Answer: confusion matrix for 500 epochs: https://github.com/hiitsAndre/mp4/blob/master/solve.txt. (uploaded as a text file to github public repository)

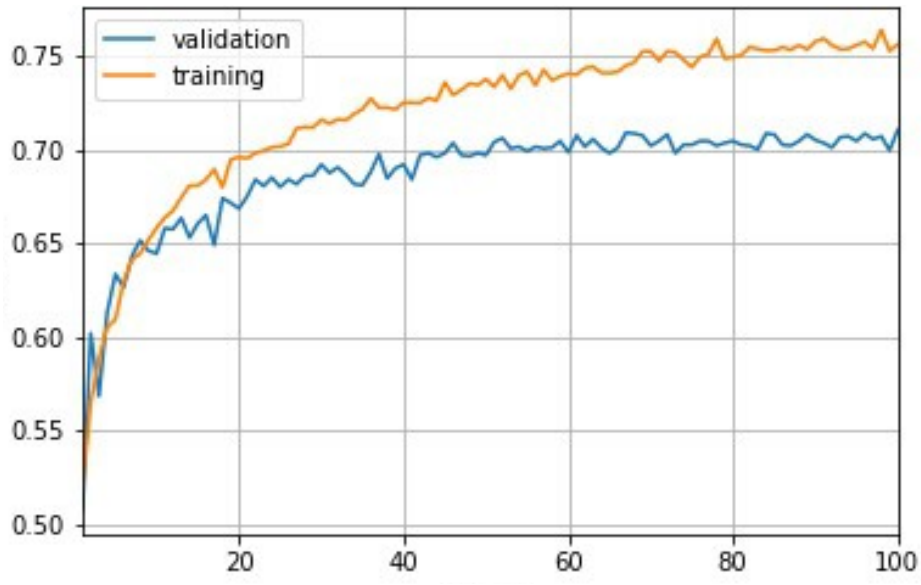
Consists of a three-layer fully-connected network and number of units of a single network layer depending on batch size and (if affine network) column number of input weight vector.

Learning rate = 1e-6.

4. Plot loss and accuracy as a function of the number of training epochs. You do not need to do a train-validation split on the data, but in practice this would be helpful.

Answer:

Loss as y-axis, epoch as x-axis, loss vs epoch graph



5. Run your pingpong agent with the trained policy. How many bounces does your agent achieve? You should get greater than 8 bounces averaged over 200 games although your results may be significantly higher. Your deep network agent does not need to beat your agent from part 1.

Answer: number of bounces in 200 games recorded in a txt file. URL link: https://github.com/hiitsAndre/mp4/blob/master/num_bounces.txt. (uploaded as a text file to github public repository)