Project #2

EE488B Special Topics in EE <Deep Learning & AlphaGo>
2017 Fall, School of EE, KAIST

- Contents
  - Exercise 1: Simple tabular Q-learning
  - Exercise 2: Baby spider learning to walk
  - Task 1 (Linear environment)
  - Task 2 (4-legged spider)
  - Task 3 (Breakout & DQN)

- Source codes & movie files
  - Download project2.zip from KLMS that contains all the source codes needed for this project. It also contains some *.mp4 files for demo.

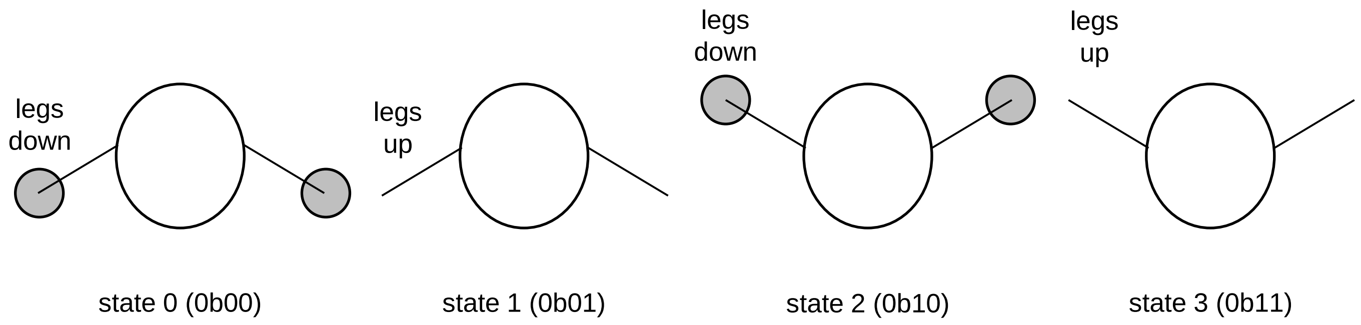
- Exercise 1: Simple tabular Q-learning
  - This is an exercise. You don't need to submit anything.
  - Run 'qlearning_episodic_task_example.py', which trains and tests an agent for the episodic task given in page 25 of lecture notes #10, which you studied in problem 2 in problem set #3.
  - The environment is defined as a class 'episodic_task_example_environment' as shown below.

```
class episodic_task_example_environment:
    def __init__(self):
        self.n_states = 5        # number of states
        self.n_actions = 2       # number of actions
        self.next_state = np.array([[1,2],[1,1],[3,4],[3,3],[4,4]], dtype=np.int)
        self.reward = np.array([[0.,0.],[0.,0.],[1.,0.],[0.,0.],[0.,0.]])
        self.terminal = np.array([0,1,0,1,1], dtype=np.int)
        self.init_state = 0      # initial state
```

  - It contains only one method __init__(), which is the method called when an instance of the class is created as done in 'env = episodic_task_example_environment()' in the code.
  - Inside the method, 'n_states' and 'n_actions' are set to 5 and 2, respectively. 'next_state' is a numpy array of size 5x2 and 'next_state[s,a]' contains the next state (0~4) when the current state is 's' and action 'a' is taken. 's' is from 0 to 4 and 'a' is either 0 or 1 since indices start from 0 in python. 'reward[s,a]' is the reward when action 'a' is taken at state 's'. 'terminal[s]' is 1 if 's' is a terminal state and is 0 otherwise.
  - Training is then performed with 20 episodes, $\alpha = 0.2$, $\gamma = 0.9$. 'max_steps' is the maximum number of steps to take in each episode, which is set to 5, but since this is an episodic task that will terminate after at most 2 steps, 'max_steps = 2' should also work.
  - 'epsilon' is set to 1 initially and decreased to 0 at the end of training. This is done by setting 4 parameters, epsilon.init = 1.0, epsilon.final = 0.0, epsilon.dec_episode = 1.0 / n_episodes, epsilon.dec_step = 0.0. You can see how epsilon is decreased by studying 'qlearning.py', which is imported by 'qlearning_episodic_task_example.py'.
  - Training is done by calling 'Q_learning_train' function, which is defined in 'qlearning.py'. After training, the trained Q table and the total reward for each episode are printed.
  - For testing, 'Q_test' function is called, which is defined in 'qlearning.py'. 'test_n_episodes' is set to 1, i.e., only one episode to run for testing and 'test_epsilon' is set to 0. Then, the total reward is printed.
  - Study the codes 'qlearning_episodic_task_example.py' and 'qlearning.py' and try to change some parameters for training and observe how the results change.

- Run ‘qlearning_continuing_task_example.py’, which trains and tests an agent for the continuing task given in page 29 of lecture notes #10, which you studied in problem 3 in problem set #3.
- This environment has 2 states and none of them are terminal states.
- Since this is a continuing task, ‘n_episodes’ is set to 1 for training. ‘max_steps’ is set to 1,000 for sufficient training.
- Learning rate and discount factor are the same as before, i.e., $\alpha = 0.2$, $\gamma = 0.9$.
- ‘epsilon’ is set to 1 initially and decreased to 0 at the end of training. This is done by setting 4 parameters, epsilon.init = 1.0, epsilon.final = 0.0, epsilon.dec_episode = 0.0, epsilon.dec_step = 1.0 / max_steps. Since epsilon needs to be decreased by 1.0 / max_steps at the end of in each step, we set epsilon.dec_step = 1.0 / max_steps. We have ‘epsilon.dec_episode = 0’ since there’s only one episode.
- After training, Q table and total reward are printed.
- For testing, the number of episodes, the maximum number of steps and epsilon are set to 1, 100 and 0, respectively. After testing, the total reward is printed, which should be 50 if the Q table gives an optimal policy.
- Observe that the Q values after training are similar to what you calculated in problem 3 in problem set #3.
- Change some parameters for training and observe how they affect the learning performance.

- Exercise 2: Baby spider learning to walk
  - This is an exercise. You don’t need to submit anything.
  - Run code ‘baby_spider.py’, which will train a baby spider to walk efficiently using tabular Q-learning. ‘baby_spider_env.py’ defines the environment and ‘baby_spider_ani.py’ is for animation. It also calls functions in ‘qlearning.py’ for training and testing using tabular Q-learning.
  - There are 4 states as shown below.



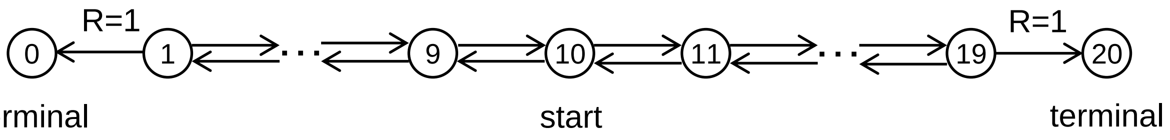
    - state 0 (0b00): both legs are down and pointing backward (starting state)
    - state 1 (0b01): both legs are up and pointing backward
    - state 2 (0b10): both legs are down and pointing forward
    - state 3 (0b11): both legs are up and pointing forward
    - Note that 0b11010101 means a binary number 11010101 (213 in decimal) in python

- Exercise 2: Baby spider learning to walk (cont'd)
  - The agent can take one of 4 actions (0~3), i.e., move legs up (0), down (1), forward (2), and backward (3).
  - Reward is +1 if the baby spider moves forward, i.e., if state transition from 2 to 0 occurs by taking action 'move legs backward'. Reward is -1 if the baby spider moves backward, i.e., if state transition from 0 to 2 occurs by taking action 'move legs forward. Otherwise, the baby spider does not move and the reward is zero. Therefore, cumulative reward is proportional to the total distance traveled by the baby spider.
  - Current state 's', action 'a', next state, and reward values are summarized in the table below, where each entry in the table is a tuple of (next state, reward).

s \ a	0 (up)	1 (down)	2 (forward)	3 (backward)
0	(1, 0)	(0, 0)	(2, -1)	(0, 0)
1	(1, 0)	(0, 0)	(3, 0)	(1, 0)
2	(3, 0)	(2, 0)	(2, 0)	(0, 1)
3	(3, 0)	(2, 0)	(3, 0)	(1, 0)

  - The goal is to maximize the discounted return with $0 < \gamma < 1$, which will make the baby spider walk as fast as possible in the forward direction.
  - To be able to walk efficiently in the forward direction, the baby spider needs to move its legs so that the state changes as follows:
    - 0 (starting state) → 1 → 3 → 2 → 0 → 1 → 3 → 2 → 0 → 1 → 3 → 2 → …
  - If you run 'baby_spider.py', which trains and tests the baby spider, you will see it will walk a distance of 5 units in 20 steps, which is optimal. By checking 'test_sum_rewards[0]', i.e., the total reward during the first test episode returned by function 'Q_test', you will be able to see it is indeed equal to 5. It will also print out values of 's', 'a', 'sn', and 'r' at each test step so that you can analyze each transition. Study 'baby_spider.py' and 'baby_spider_env.py' and understand how the codes work.
  - 'baby_spider.ani.py' defines a class 'baby_spider_animation' that handles animation. Interpolation is used to show smooth movements for animation, i.e., 20 frames are generated per step so that legs and the baby spider move more naturally. However, the agent does not see the animated frames, it only gets the state and reward in each time step.
    - You can change 'frames_per_step' parameter to speed up or slow down animation. In 'ani = baby_spider_animation(Q, env, test_max_steps, test_epsilon)' in 'baby_spider.py', 'frames_per_step' is not specified and the default value of 20 will be used. If you do 'ani = baby_spider_animation(Q, env, test_max_steps, test_epsilon, 10)', for example, then it will generate 10 frames per step and the animation will appear 2 times faster.
  - You can save the resulting animation as a *.mp4 movie file. See 'baby_spider.py' for details.
  - The speed of animation is supposed to be 20 frames per second since 'interval' is set to 50, i.e., 50 msec in __init__ method in the class 'baby_spider_animation' in 'baby_spider_ani.py'. However, animation may appear slower when your computer is not so fast or when you access a server computer remotely. This problem may be more severe especially if you access a server computer from outside KAIST due to slow network connection. But, if you save the movie as a *.mp4 file and play it on a local computer, it will play at the normal speed of 20 frames per second.

- Task #1 (Linear environment – 10 points)
  - Modify ‘qlearning_episodic_task_example.py’ to train and test an agent for the following task.
  - The environment has 21 states in linear topology as shown below. The center state (s=10) is the starting state and the first (s=0) and the last (s=20) states are terminal states. Agent can go left or right and the reward is 1 when one of the terminal states is reached and is 0 otherwise.



  - First, set epsilon = 1 during whole training, i.e., random walk.
    - In your code, set n_episodes = 1, max_steps = 1000, alpha = 0.2, and gamma = 0.9 for training and set test_n_episodes = 1, test_max_steps = 1000, test_epsilon = 0 for testing. Run your code multiple times. Does the agent ever exhibit an optimal policy during **testing**, i.e., does ‘test_n_steps[0]’ ever become 10? In either case, explain why.
    - Now, change n_episodes to 5 for training while keeping all the other parameters unchanged. Run your code multiple times. Does the agent ever exhibit an optimal policy during **testing**? You will see sometimes it does and sometimes it does not. Explain why.
    - Now, change ‘n_episodes’ to 1000 for training while keeping all the other parameters unchanged. Run your code **once**. Does the agent exhibit an optimal policy during testing? What is the average number of steps per episode during **training**? To calculate the average, compute the mean of ‘n_steps’ returned from ‘Q_learning_train’ function, which is an array containing the number of actual steps taken for each episode. Try to explain why it takes such many steps on average. Derive Q values analytically and verify that the Q values you obtained by running your code are close to the theoretical values.
  - Now, choose epsilon = 1 initially and linearly reduce it to 0 at the end of training.
    - Set n_episodes = 100, max_steps = 1000, alpha = 0.2, and gamma = 0.9 for training and set test_n_episodes = 1, test_max_steps = 1000, test_epsilon = 0 for testing.
    - Plot the number of steps for each episode, where x axis is the episode number (1 through ‘n_episodes’) and y axis is the number of steps for each episode (values in ‘n_steps’ returned from ‘Q_learning_train’ function).
    - Does the number of steps converge to the optimal value at the end of training?
  - In your report (hard copy that you bring to class on the due date), include the following:
    - Source code and plot
    - Explanations on how you designed your code
    - Answers to questions given above
  - In addition to submitting a hard copy of your report, submit your source code electronically as follows:
    - Place your source code in your home directory of your computer in N5. Its name should be ‘linear.py’. The name of the class defining the environment should be ‘linear_environment’.
    - Even if you use your own computer, you should copy your file to your home directory of your machine in N5. Refer to Appendix A in Project #1 for the IP address of the machine you are supposed to use and your login id.
    - Immediately after the deadline, all the files will be collected automatically. Double check the file name and class name since if they are incorrect the file will not be collected and/or automated checking will fail.

- Task #2 (4-legged spider – 10 points)
  - Modify 'baby_spider.py' and 'baby_spider_env.py' to train and test an agent for the following task. Name your codes as 'spider.py' and 'spider_env.py'. In 'spider_env.py', define a class named 'spider_environment' that implements the environment described below.
  - Instead of 2 legs, now the spider has 4 legs. Previously in 'baby_spider_env.py', both legs moved together. But, now all 4 legs can move independently. Therefore, there are 4^4=256 possible states since each leg has 4 states. The agent can move all 4 legs simultaneously and therefore the number of possible actions is also 4^4=256.
  - A state is a 8-bit integer (0~255) and in binary format it should be encoded as follows, i.e., the LSB (bit 0) is 1 if the left front leg is up and is 0 if down and the MSB (bit 7) is 1 if the right back leg is pointing forward and 0 if pointing backward.

right back leg		left back leg		right front leg		left front leg	
1=forward	1=up	1=forward	1=up	1=forward	1=up	1=forward	1=up

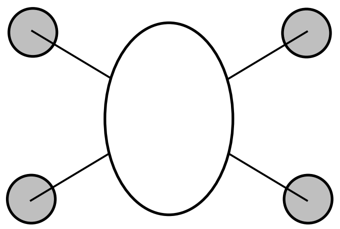
MSB LSB

  - An action is a 8-bit integer (0~255) and in binary format it should be encoded as follows:

right back leg	left back leg	right front leg	left front leg
00: move leg up 01: move leg down 10: move leg forward 11: move leg backward	00: move leg up 01: move leg down 10: move leg forward 11: move leg backward	00: move leg up 01: move leg down 10: move leg forward 11: move leg backward	00: move leg up 01: move leg down 10: move leg forward 11: move leg backward

MSB LSB

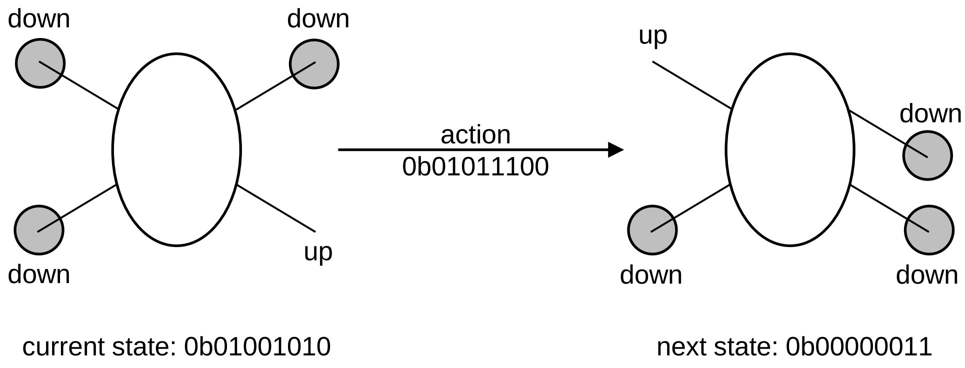
  - The starting state should be 0b00001010 as shown below.



starting state: 0b00001010

  - The next state is determined from the current state and action in the same way as in baby spider, but the state for each leg can be changed independently.

- Task #2 (4-legged spider – cont'd)
  - Reward is determined as follows:
    - Calculate the total # of legs that are currently down and still down after an action is taken. Use 'total_down' variable to store this value. For example, 'total_down' is 2 in the example below because currently 3 legs are down, but the left front leg moved up after an action is taken. The current state is 0b01001010, i.e., left front leg and right front legs are down and pointing forward, left back leg is down and pointing backward, and right back leg is up and pointing backward. The action is 0b01011100, which mean move left front leg up, move right front leg backward, move left back leg and right back leg down. Therefore, the next state becomes 0b00000011, i.e., left front leg is up and pointing forward and all the other legs are down and pointing backward.



    - Calculate 'total_force', which is the sum of the following value for all 4 legs
      - (up == 0 and fw == 1 and action_bw == 1) - (up == 0 and fw == 0 and action_fw == 1)
    - In the figure above, 'total_force' is 1 since only the front right leg contributes +1.
    - Reward is 0 if 'total_down' is 0.
    - Reward is equal to (total_force / total_down) if total_down >= 3.
    - Reward is equal to (total_force / total_down) if total_down = 2 such that two diagonal legs (either front left & back right legs or front right & back left legs) are down and they are still down after an action is taken.
    - Reward is equal to 0.25 * (total_force / total_down) in all other cases. The factor 0.25 is included to model friction between the spider's body and the ground due to unbalanced leg positions.
  - After constructing the environment, train and test the agent.
  - Try to achieve a total reward of at least 9.0 during 20 test steps. Watch 'spider_example.mp4' that demonstrates such an example.
  - After training, do 'ani = spider_animation(Q, env, test_max_steps, test_epsilon)' as you did in 'baby_spider.py' for animation. Use 'text_max_steps = 20' and 'test_epsilon = 0.0'. The class 'spider_animation' is defined in 'spider_ani.py'. Save your animation as 'spider.mp4'.
  - In your report (hard copy that you bring to class on the due date), include the following:
    - Source codes
    - Explanations on how you designed your code
    - Values of state, action, next state, and reward during 20 test steps.
  - In addition to submitting a hard copy of your report, submit your files also electronically as follows:
    - Place your source codes and *.mp4 movie file in your home directory of your computer in N5. File names should be 'spider.py', 'spider_env.py', and 'spider.mp4'. If file names are incorrect, they will not be collected. Make sure the name of the class defined in 'spider_env.py' is 'spider_environment'. If the class name is different, automated checking that we will be running will not work.

- Task #3 (Breakout & DQN – 20 points)
  - Study ‘breakout_env.py’, which defines a class for breakout environment.
  - Some simplifications were made for fast training. Some differences from the Breakout game in Atari 2600 are:
    - A brick, ball, paddle are all single pixels.
    - Pixel value is 1 if something (brick, ball, paddle) is present and is 0 otherwise.
    - Reward is 1 if a brick is broken and is 0 otherwise.
    - There is only 1 life, i.e., if you lose a ball then the game is over.
    - The ball can only move in ± 45 degree angles.
  - When defining an instance of the class ‘breakout_environment’, you can specify the screen size (ny * nx), number of rows for bricks (nb), number of empty rows on top (nt), number of most recent frames to be used as the input of DQN (nf). By default, they are set to the following values, but you can try other values if you want.
    - Screen size: 5 (nx) * 8 (ny) pixels
    - There are 3 rows of bricks (nb = 3) and there is one empty row on top (nt = 1).
    - Only 2 most recent frames are used at the input of the D (nf = 2).
  - High-resolution animation is created when you run ‘breakout_ani.py’, which generates multiple interpolated video frames for each time step and by drawing bricks and the ball in a more realistic way.
  - Run ‘breakout_example.py’, which will show an animation. Currently, only random action is implemented in ‘breakout_ani.py’ and therefore the score will be very low. Watch ‘breakout_example.mp4’ for an example produced by the code.
  - Write a code ‘breakout.py’ that implements DQN to train and test an agent to play this game. After training, save the neural network’s parameters as check point files. To save neural network parameters, do the following:

```
*** Define neural network model here ***
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
*** training goes here ***
save_path = saver.save(sess, "./breakout.ckpt")
```

  - To retrieve saved parameters later for testing and for animation, do the following:

```
*** Define neural network model here ***
sess = tf.InteractiveSession()
saver = tf.train.Saver()
saver.restore(sess, "./breakout.ckpt")
*** testing or animation goes here ***
```

  - Modify ‘breakout_ani.py’ so that it can retrieve the saved neural network parameters and run animation using the neural network. Save the animation as ‘breakout.mp4’.

- Task #3 (Breakout & DQN – cont’d)
  - In your report (hard copy that you bring to class on the due date), include the following:
    - Source codes
    - Explanations on how you designed your code
    - Performance of your agent, i.e., the total score, the number of steps it took to achieve the score.
    - 5 screen shots captured from *.mp4 that can show how the agent plays the game. Try to capture important moments such as breaking a column of bricks.
  - In addition to submitting a hard copy of your report, submit your files electronically as follows:
    - Place your source codes, checkpoint files, and *.mp4 movie file in your home directory of your computer in N5. File names should be ‘breakout.py’, ‘breakout_ani.py’, ‘breakout.ckpt.*’, and ‘breakout.mp4’. If file names are incorrect, they will not be collected.