Assignment 02

In this assignment, you will get familiar with programming using TensorFlow. Try the given example written in Python, then rewrite it using TensorFlow based on the example. Try different values with the parameters for better results.

The starter notebook was provided and can be downloaded from HuskyCT. There are 2 dependencies described below:

1. **TensorFlow**: Use the instructions at https://www.tensorflow.org/install/pip to install TensorFlow. Using GPUs is not required for this assignment, but if you have one, you may want to consider installing the GPU version. Setting a virtual environment, namely for example "tensorflow", is recommended (see the link above). To use Jupyter Notebook in the virtual environment with tensorflow:

activate tensorflow

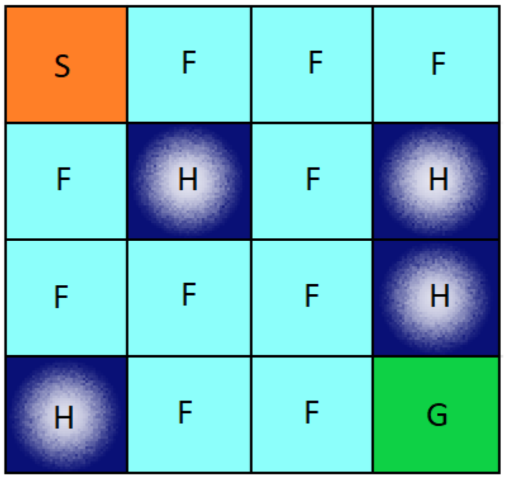
You will see the name of the environment in the prompt, then run

conda install jupyter

Then look for the IDE, usually "Jupyter Notebook (tensorflow)", in Windows Start menu.

2. **OpenAI Gym**: We will use environments in the OpenAI gym, a testbed for reinforcement learning algorithms. For installation and usage instructions see https://gym.openai.com/docs. OpenAI gym provides an easy way to experiment with learning agents in a number of toy games.

You are going to create a FrozenLake environment using the OpenAI gym. The FrozenLake environment consists of a 4x4 grid of blocks as follows, with each one either being the start block (S), the goal block (G), a safe frozen block (F), or a dangerous hole (H).



The objective is for an agent learn how to navigate from the start block to the goal without falling in a hole. At any time the agent can choose an action either to move up, down, left, or right. In the financial context, a similar problem could be thought of for trading decisions.

A table (called Q table – Q is for quality) records a value for how good it is to take a given action within a given state. There are 16 possible states, one of which for each block, and 4 possible actions (left, right, up, down), so the table is of size 16x4. The table is initialized with all zeros, and updated when rewards received for various actions are observed. The value of each cell is the maximum expected future reward for that given state and action. The reward at every step is 0, except for entering the goal block, which provides a reward of 1.

The Q table is updated using the following rule (so called Bellman equation).

$$Q^{new}(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Note that the new value is a weighted sum of the old value and the new information. Only a small portion of the new information is taken into the new value. The update in each iteration:

$$\Delta Q(s_t,a_t) = \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right]$$

In which, α is the learning rate, $\Upsilon \in [0,1]$ is the discount factor that controls the importance of future rewards. $Q(s_t, a_t)$ is the Q-table value of action a_t in state s_t and time t. When $\Upsilon = 0$, only immediate rewards are taken into account by the agent. Actions that result in short-term reward are preferred than those that result in long-term reward.

In the given Python script, the outer loop runs for a number of episodes. An episode of the algorithm ends when the agent arrives at the final goal G or falls in a hole H. The inner loop is where an action is tried, which then has its new state and reward:

new_state, reward, done, info = env.step(action)

The Q table is learned iteratively using the Bellman equation:

*Q[state,action] += alpha * (reward + gamma * np.max(Q[new_state]) - Q[state,action])*

The agent can explore the space randomly, or exploit the Q table. Balancing between exploration and exploitation is via ε-greedy method, in which the agent selects at each time step a random action with a probability ε instead of selecting greedily one of the learned optimal actions with respect to the Q-function. At a given time if an uniform random number $x < \varepsilon$, the agent can explore; otherwise he takes the best action learned in the Q table using np.argmax(Q[state]).

The learn Q table then can be used as a cheat sheet to play the game (See the code in the notebook). Note that action that should be taken would have the highest value in the Q table

Submission includes:

a. A notebook with name as "group_xxx_assignment_yyy.ipynb" with detailed explanation of the work as markdown text
b. Link to your GitHub for the shared notebook
c. All the other files
d. For (a) and (b), if your group has more than 1 notebook (each per student), please include all of them but select one for being graded