CSCE 790-001: Deep Reinforcement Learning and Search
Coding Homework 2
Due: 10/12/2021 at 11:59pm

Your code must run in order to receive credit.
For grading purposes, do not change the signature of the function. Your code must return exactly what is specified in the documentation.

For all problems, your implementation should run on a laptop CPU in 5-7 minutes or less. Excessively long run-times will be penalized. If it is taking longer, profile your code to find the bottleneck.

For training your DNN, make sure your tensors have the appropriate shape. For example, comparing a tensor of shape (N,) and shape (N,1) can give unexpected results.

Key building blocks:

- `env.sample_transition(state, action)`: returns, in this order, the next state and reward
- `env.get_actions()` function that returns a list of all possible actions
- `env.sample_start_states(num_states)` function that `num_state` start states
- `env.states_to_nnet_input(states)` converts a list of states to a numpy array for the input to a neural network
- `env.is_terminal(state)` returns true if state is terminal

Installation

We will be using the same `conda` environment as in Homework 0.

The entire GitHub repository should be downloaded again as changes were made to other files. You can download it with the green "Code" button and click "Download ZIP".

1 Solving the 8-puzzle

We will be using approximate value iteration to learn a value function that can be used to solve the 8-puzzle. The following exercises will require you to design a PyTorch model for the value function. Do this by implementing `get_value_net`.

Hint: You can accomplish these exercises with a neural network with three 100 dimensional hidden layers with ReLU activation functions and a one dimensional output layer with a linear activation function.

1.1 Supervised Learning (10 pts)

To build your first PyTorch model, we will first examine the case in which we train a value function in a supervised manner. In this case, the optimal value is given to us for each state. The function you will implement will be given a one-hot representation of each state (an 81 dimensional vector) and the corresponding optimal value for each state. This will be in the form of a numpy array. After training, the

mean squared error (MSE) will be computed. Your network should achieve an MSE of 3.0 or less.

Running the code:
Implement `supervised`.

```
python run_code_hw_2.py --algorithm supervised --env puzzle8
```

1.2 Following the Greedy Policy (10 pts)

Instead of being given the optimal values, we would like to learn an approximation of the optimal value function using approximate value iteration. We will monitor progress by monitoring the performance of the induced policy. To do this, implement `follow_greedy_policy`. Your implementation should follow the greedy policy until the number of steps has elapsed or until the terminal state is encountered. It is possible for the very first state to be the terminal state.

For this implementation, you can make use of the `expand` function provided in `code_hw2.py`.

1.3 Approximate Value Iteration (30 pts)

Value iteration has the following form:

$$V(s) = \max_a (r(s,a) + \gamma \sum_{s'} p(s'|s,a) V(s')) \tag{1}$$

Since the 8-puzzle is deterministic, we can say that the next state is given by a deterministic function $A(s,a)$. Additionally, we set γ to 1. Therefore, we can simplify the equation:

$$V(s) = \max_a (r(s,a) + V(A(s,a))) \tag{2}$$

This approximate value iteration algorithm is shown in Algorithm 1. In this algorithm, we will generate a batch of 20,000 states using `env.sample_start_states(num_states)`. This function generates states by starting from the goal and scrambling it 0 to N times. This is a form of prioritized sweeping to ensure the learning signal can propagate from the goal state to states further away.

You should make use of your implementation of `supervised` to train your value function.
You should make use of your implementation of `follow_greedy_policy` to monitor your progress.

You can make use of the `expand` function provided in `code_hw2.py`.
You can make use of `misc_utils.flatten` and `misc_utils.unflatten` to do value iteration efficiently. Particularly, the for loop on line 3 of 1 can be sped up by computing this update in a batched manner.

When your code is finished training it will be evaluated using your implementation of `follow_greedy_policy`. At least 600 out of the 1000 test cases should be solved.

Running the code:
Implement `deep_vi` and `follow_greedy_policy`.

```
python run_code_hw_2.py --algorithm value_iteration --env puzzle8
```

Algorithm 1 Deep Approximate Value Iteration

for $m = 1$ to 50 **do**
  $S \leftarrow get\_training\_states(20000)$
  for $s \in S$ **do**
    if s is terminal **then**
      $y_i \leftarrow 0$
    else
      $y_i \leftarrow \max_a(r(s,a) + v_\theta(A(s,a)))$
    end if
  end for
  $train(v_\theta, X, \mathbf{y})$
end for
Return θ

2 Deep Q-learning Learning (40 pts)

For this exercise, you will be implementing deep Q-learning. The representation of the AI farm given to the neural network with be an 800 dimensional vector.

You can monitor the performance of your algorithm with `run_greedy_dqn_policy`. NOTE: this function puts the dqn in eval mode. Therefore, when training, make sure to do `dqn.train()` at the appropriate time.

Your algorithm should be solving the majority of instances after 800 episodes.

Important: In this exercise, we will assume that we do not have access to the dynamics of the MDP. Therefore, in your implementation of Q-learning, you cannot use `env.state_action_dynamics(state, action)`.

Replay buffer:
The training algorithm should make use of a replay buffer, which is provided to you.
You can initialize a replay buffer:
`ReplayBuffer(env, replay_buff_size)`

Add to a replay buffer:
`replay_buff.add_experience(state, action, reward, state_next)`

And sample training data:
`states_nnet, acts, rewards, states_next_nnet, is_terminal = replay_buff.samp_data(num)`

Target network:
Your algorithm should also make use of a target network that is updated periodically. This is done like this:
`dqn_target.load_state_dict(dqn.state_dict())`
Do not forget to put the target dqn in evaluation mode.

Huber Loss:
You can obtain the Huber loss in PyTorch like this: `criterion = nn.SmoothL1Loss(beta=1.0)`.

Hint: Your algorithm will be more stable if you wait until the number of elements in the replay buffer is at least the size of the training batch. You can get the size of the replay buffer with `replay_buff.size()`.

Hint: You can accomplish these exercises with a neural network with a 75 dimensional hidden layer, followed by a 50 dimensional hidden layer (both with ReLU activation functions) and a four dimensional output layer with linear activation functions.

Running the code:
Implement `deep_q_learning`.

```
python run_code_hw_2.py --algorithm q_learning --env aifarm_0
```

Try stochastic environments such as `aifarm_0.1` and `aifarm_0.5`.

What to Turn In
Turn in your implementation of to `code_hw/code_hw2.py` to Homework/Coding Homework 2 on Blackboard.