Reinforcement Learning - MP7

Due Wednesday, May 6th 2020, 11:59pm

Starter code

You can download the starter code here: zip, tar

Run the main functions of "tabluar.py" and "dqn.py" to train your RL model. Then run "mp7.py" to run/test the resulting model locally. Note that "mp7.py" is not that important. It's only there to help you see how your model performs. You can visualize each episode by turning `Render` as `True` .

General guidelines

Basic instructions are similar to previous MPs: For general instructions, see the main MP page and the course policies.

Code and model will be submitted on gradescope.

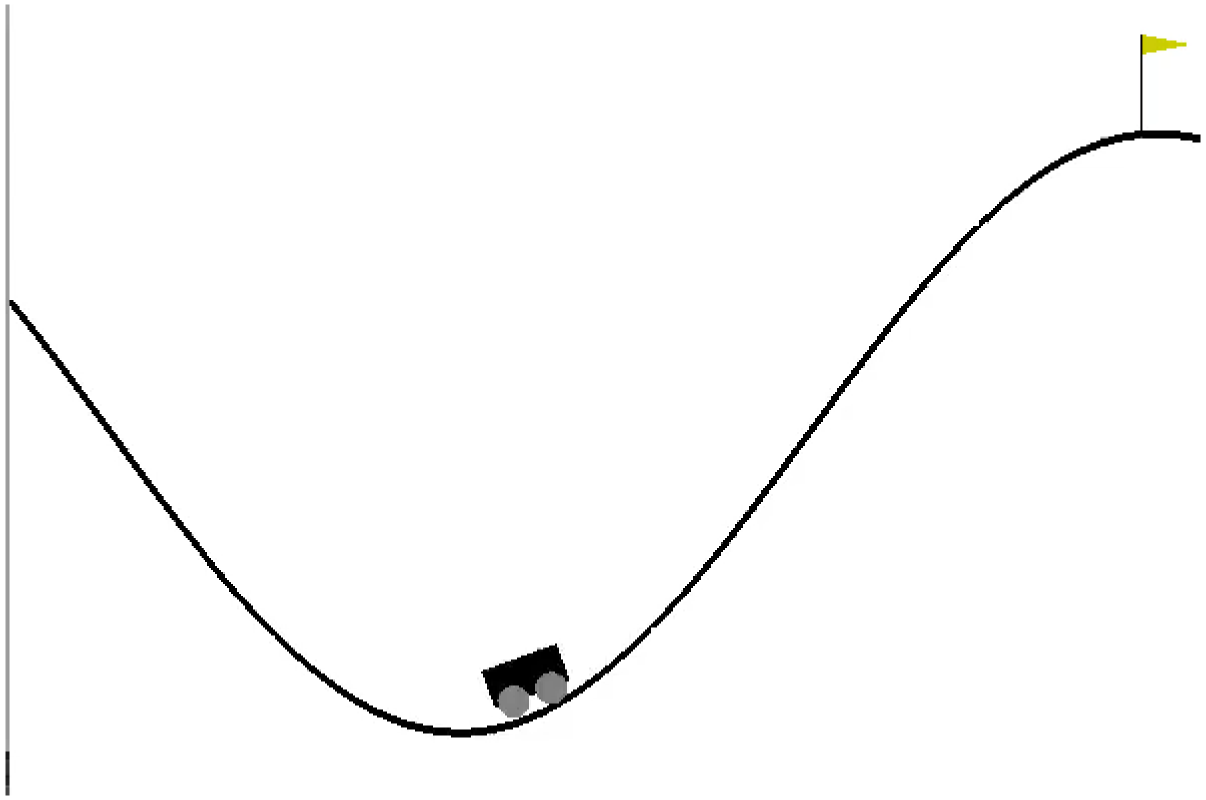
The extra credit portion will be submitted separately on gradescope. In your MP average, the extra credit is worth 10% of the value of the regular part of the assignment.

Objective

Create an RL agent to do cool things, like balancing a pole, swinging a multi-pendulum as high as possible or drive a car uphill.

Our RL agent will be implemented via Q-Learning in the tabular setting and function approximation setting

Environment - MountainCar-v0



In the Mountain Car environment, a car is on a one-dimensional track, positioned between two mountains. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. The car's state is described by the observation space which is a 1D vector containing car's horizontal position and velocity. The car can take 3 actions: Left, Do Nothing, and Right. You don't need to worry about state transitions. They are handled by Gym environment. The car will start at the bottom of the valley between hills (at position approximately -0.5). The episode ends when either the car reaches the flag on the right hill or it fails in 200 moves (or frames). All the move that fails to reach the flag will get reward -1. Therefore, in the episode where the car fails to reach to goal, the total reward would be -200. The entire episode will be scored as 1.0 if the car reaches the goal ($total\_reward > -200$) or 0.0 otherwise.

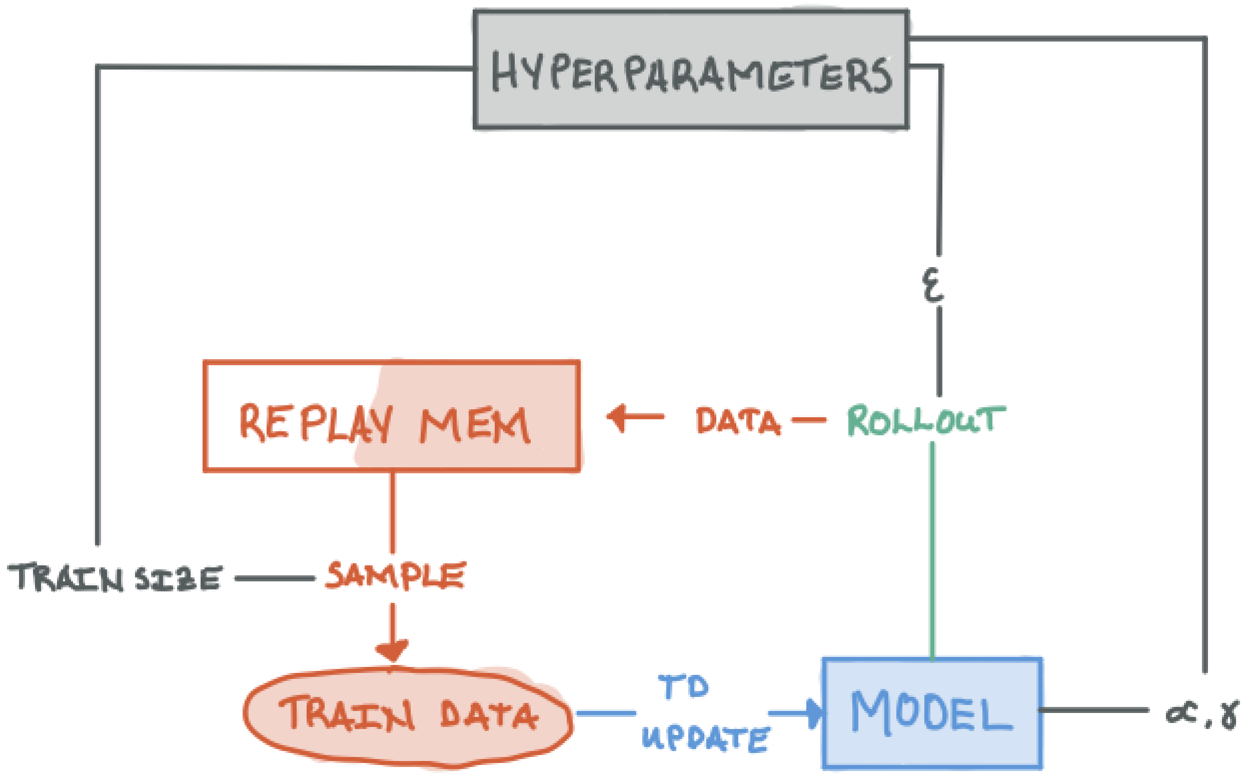
Observation space

Index	Meaning	Min	Max
0	Car Position	`-1.2`	`0.6`
1	Car Velocity	`-0.07`	`0.07`

Action Space

- 0 - Left
- 1 - Do Nothing
- 2 - Right

High Level Algorithm



1. Initialize Model
2. Rollout trajectories using current model and store it into the replay memory
3. Sample data from the replay memory to update the model's Q-value estimates
4. Repeat steps 2-4 until convergence

Your Job

Implement the initialization and training code of Q-learning for the tabular setting.

Most of the code is written for you, you just have to fill out some functions

Rollout

Complete one line of the `rollout` function in `utils.py`. This function plays a given number of episodes and returns the replay memory and score. The replay memory is a list of `(state, action, reward, next_state, done)` at each timestep for all episodes, and score is a list of total rewards for each episodes.

We use the replay memory in our `train` function in `utils.py`. During training, the agent samples data randomly from the replay memory to reduce the strong temporal relationship between consecutive pair of `(state, action, reward, next_state, done)`.

Tabular Setting

- Implement the following functions of `TabQPolicy` in `tabular.py`
  - Initialization `__init__`: create the table to hold the Q-vals.
  - `qvals`: given the current state, return the value for each action.
  - `td_step`: look below for the description of temporal difference learning.
- Tune the discretization in main call of `tabular.py`

- The state space for this environment is continuous.We have provided a function for discretizing the state space into bins for you. You just have to choose the granularity of division by tuning the `buckets` argument. The model takes as input - bins, which denotes the number of distinct bins for each dimension of the state space. Note that the default value given to you may not work.

Training: `Temporal Difference`

You have to implement the `td_step` function in `tabular.py`.

The function takes in training data of the form `state`, `action`, `reward`, `next_state`, `terminal`.

Recall the TD update for Q-vals is

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (target - Q(s,a))$$

where $target = r + \gamma \cdot \max_{a'} Q(s', a')$ if the state is not terminal (Note that `done is True` may contain the case when the car fails. This case is not the terminal) and $target = r$ otherwise. In this environment, you will assign reward as 1.0 if `done is True and next_state[0] >= 0.5` so that you won't lose the reward due to discretization.

Return the square error of the original q-value estimate, i.e. the square of the difference between that and `target`.

During training, in order to get a stable convergence, you can maintain a table $N(s,a)$ to save the number of times each $Q(s,a)$ is updated and use $C/\ (C + N(s,a))$ as the decay rule for the learning rate (C is some constant). You can stick with constant learning rate as well.

Hyperparameter Tuning

A large part of this assignment is tuning the hyperparameters to train a good model. See the `hyperparameters` function in the `utils.py` script to see what parameters you can control. The default parameters are good for starter and can generate a working model, but you are free to change parameters by setting arguments.

Extra Credit: DQN on CartPole-v1

In this extra credit part, you need to run DQN policy on a new environment CartPole-v1. In the CartPole environment, there is a pole standing up on top of a cart. The goal is to balance this pole by moving the cart from side to side to keep the pole balanced upright. A reward of +1 is provided for every timestep that the pole remains upright, and the episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. You should use identical `utils.py`. In order to test locally, you can simply replace the environment line in `mp7.py` as `env, statesize, actionsize = gym.make('CartPole-v1'), 4, 2`. The CartPole evironment has a size-4 state space and size-2 action space. In Cartpole, there is no 200 moves limit. We simply evalute each episode by its total rewards. You will need a different set of hyperparameters for this environment.

You have to create a neural network that takes the state as input, and output the score for each action. The state is no longer discrete as in the tabular setting.

- Implement the following functions of `DQNPolicy` in `dqn.py`
  - Initialization `__init__`: create optimizer and loss
  - `qvals`: return the Q-vals for each action give the current state
  - `td_step`: temporal difference learing update
- Implement `make_dqn`

  - This defines your neural network. You have the creative freedom to make it however you want as long as the returned object is a PyTorch nn.Module.

Submission

For tabular Q learning, submit `tabular.py` and your trained model `tabular.npy` on gradescope

If you do the extra credit, submit your `dqn.py` and the trained model `dqn.model` to the separate extra credit assignment on gradescope.

Note that you don't need to submit `utils.py` to the gradescope.

Package Requirements

This assignment requires some Python packages

- `gym` - RL environment
- `numpy` - numerical analysis package
- `tqdm` - progress bars

References

- Gym - A reinforcement library
- PyTorch - A deep learning library