Bonus Homework 12 - Deep Reinforcement Learning

February 2021

(soft submission deadline)
10 Feb 23:59 via `https://forms.gle/H31ckx5251Qg3Ndm6`

1 Homework summary

As you all are probably pretty stressed out during the exam time and we uploaded the corresponding courseware chapter pretty late this week, we decided to make this homework a bonus homework only. You will learn a great deal, hopefully have some fun and of course if you submit an outstanding you will get a bonus point! But if you feel you don't have the time, do not worry about it! We wish you all the best for your exams!

In this bonus homework you will learn the basics of dealing with gym environments and train a simple reinforcement learning agent to solve one environment which is trained with the most basic deep reinforcement learning, namely Depp Q-Networks (plus experience replay). We know, it's exam time and you are all pretty busy, nevertheless we think it is crucial to be familiar with the basics of DRL, which is why we still offer this homework. Nonetheless, you will get plenty of instructions and tips to solve it.

2 The task

The task is to implement an reinforcement agent solving the 'cartpole-v1' gym environment. We thus want to you implement a simple DQN algorithm with experience replay. Your code should consist of three parts:

1. The Q-Network class
2. The Replay-Buffer class
3. The training loop

Optionally, include a test function, visualizing your agent's performance before and after training.

3 Requirements

Additionally to our old friends tensorflow and numpy, make sure to install and import gym. [1]

4 The Q-network Agent

Your agent is at its core a simple ANN with some additional functions. [2] You should implement the following methods:

- **init:** You can implement your network using just a few simple fully connected layers [3] and an appropriate readout layer, where the number of units should correspond to the number of actions available in the environment you want to use. [4][5]
- **call:** Pass the input through the network.
- **act:** Given an input, compute the q-values and then implement epsilon greedy sampling to output an action. [6]
- **max_q:** Given an input this method should output the maximum q-value.[7]
- **q_val:** Given an input and actions (actions because we consider having a batch size > 1) output the q-values corresponding to the actions. [8]

5 The Replay Buffer

Implement a replay buffer class to collect (state, action, reward, new state) values for the experience replay. The buffer should have the following methods:

- **init:** Initalize the buffer. The buffer should have a fixed size. [9]
- **store:** Stores a (state, reward, action, state_new) pair for a single time step in the buffer. If the buffer is full, delete the oldest entries and add the newest one (First In First Out. [10]
- **sample:** Sample a trajectory for a given sample size from the buffer. [11]

6 Training

To write the training procedure, carefully consider the Pseudo code in the courseware.

1. **initialize**: Before training you should initialize the following things:
    - An agent. 0he gym environment. Here are the docs: `https://gym.openai.com/docs/` [12]

- The number of episodes, the number of steps to collect experiencce, the number of train steps and the optimization batch size used to optimize your agent for each episode.
- The optimizer and the loss function. [13]

2. **Experince Replay:** At the beginning of each epoch collect experience to store in the buffer using your current agent to interact with the environment. Starting with an initial state you get by calling env.reset(), make a loop and [14] let your agent choose an action based on the current state, pass on this action to the environment (be careful, the gym environment wants a single integer as an action!)[15] to collect the new state and the reward. Store everything in your replay buffer. . The environment will always return a boolean indicating whether the agent is 'done'. Make sure to stop the sampling then and reset the environment before continuing. [16]

3. **Optimize**: In each epoch, after the experience replay, loop through the number of training steps. In each step, get a sample from your buffer (the sample size is the optimization batch size). [17]
   Compute the targets: target = rewards + gamma * max q-values given the new states
   Compute the q-values of the old states given the chosen actions.
   Make a train step as you are used to. [18] [19]

7 Some words on the training and the choice of hyperparameters

You can nicely visualize your training progress and the performance of your agent by making use of env.render(). A mean of comparing the performance of your initial random agent and the trained one would for example be to compare the number of time steps it needs to solve the environment. Do not rely on the loss for estimating your training success!! The loss should actually not go down too much as the agent should see some new training samples in each training episode. If your loss goes down quickly but the performance does not get better, you might be overfitting. Consider increasing the buffer size, decrease the network parameters and/or the optimization step in each episode. Have fun!

Notes

[1] pip install gym, import gym

[2]Remember to inherit form tf.keras.Model in the class declaration

[3]We used 3 a 16 units

[4]Remember the readout layer should output raw q-values and therefore should have no activation function.

[5]Once you have created the gym environment you can access its number of actions with env.action_space.n

[6]To compute the q-values you need to pass the input through the network using the call method. For epsilon greedy sampling you should with a low probability choose a random action and else choose the action that corresponds to the maximum q-value. Thus, you will need a hyperparameter epsilon fixing those probabilities. If you are stuck here, google will help you, it is a common thing and there are a lot of simple implementations out there already.

[7]Again you need to pass your input through the network and then only select the maximum value of the output by e.g. using tf.reduce$_max(x, axis = -1)$

[8]First, you should compute the q-values by calling the network on the input. You can select the q-values corresponding to the actions by using tf.gather(x, actions, batch$_size = 0$

[9]Our tip would be to implement the buffer in form of a dictionary, where the keys are ['state', 'action', 'reward', 'state_new'] and the values for each key is a list.

[10]To remove the earliest entry of a list you can use list.pop(k)

[11]You may want to consider using random.choices() to sample from a list

[12]To create an environment use gym.make('env-name')

[13]We used Adam and the mean squared error.

[14]Remember your Q-network wants a batch size but your state will be a single array. Make sure to us np.expand_dims()

[15]Consider calling .numpy() on the action of your agent.

[16] new_state, reward, done, info = =e nv.step(action)

[17]Make sure that everything has the correct shape. Shape 0 of everything should be the batch size.

[18]The q-values you computed are equal to the 'predictions' when computing the loss.

[19]Make sure to do everything except calculating the q-values and the loss **outside** the gradient tape! That is to detach the target and not to slow down your code!