COMP9444 Neural Networks and Deep Learning

Session 2, 2017

Project 3 - Deep Reinforcement Learning

Due: Sunday 29 October, 23:59 pm
Marks: 15% of final assessment

Introduction

In this assignment we will implement a Deep Reinforcement Learning algorithm on some classic control tasks in the OpenAI AI-Gym Environment. Specifically, we will implement Q-Learning using a Neural Network as an approximator for the Q-function, with experience replay.

There are no constraints on the structure or complexity of network to be implemented, and marks are awarded on the basis of the learning speed and generality of your final model.

Because we are not operating on raw pixel values but already encoded state values, the training time for this assignment is relatively short, and each problem should only require approximately 15 minutes on a standard laptop PC.

Preliminaries

We will be using the AI-Gym environment provided by OpenAI to test our algorithms. AI Gym is a toolkit that exposes a series of high-level function calls to common environment simulations used to benchmark RL algorithms. All AI Gym function calls required for this assignment have been implemented in the skeleton code provided, however it would be a good idea to understand the basic functionality by reading through the getting started guide.

You can install AI Gym by running

```
pip install gym
```

or, if you need admin privileges to install python packages:

```
sudo -H pip install gym
```

This will install all environments in the "toy_text", "algorithmic", and "classic_control" categories. We will only be using those in "classic_control" If you want to only install the classic_control environments, to save disk space or to keep your installed packages at a minimum, you can run;

```
pip install 'gym[classic_control]'
```

(prepend `sudo -H` if needed)
To test if the required environments have been installed correctly, run (in a python interpreter);

```
import gym
env = gym.make('CartPole-v0')
env.reset()
env.render()
```

You should then be able to see a still-frame render of the cart pole environment.
Next, download the starter code from hw3/src (or src.zip)
`neural_Qtrain.py` Skeleton code - all functionality will be implemented in this file

How to run the code

The structure of this assignment is somewhat different to Assignment 1 and 2. Instead of functions to implement in a separate file, you have been given a partially completed python script.
Once you have implemented the unfinished parts of the script, you can run the code the same way you have with `python3 neural_Qtrain.py`. However, you can also call your `main()` function from another file, or invoke it from the command line if you wish to pass in specific argument values.

If running from the command line, navigate to the directory where the file is located and call the file like so:

```
$ python neural_Qtrain.py "CartPole-v0"
```

Would run the cartpole environment.

Alternatively, you can run the code interactively in ipython with e.g.

```
$ ipython
    In [1]: import neural_Qtrain.py as qlearn
    env, state_dim, action_dim, network_vars = qlearn.setup()
    In [2]: qlearn.qtrain(env, state_dim, action_dim, *network_vars, render=True,
num_episodes=5)
```

You can then run `qlearn.qtrain()` again with different values for "render", "num_episodes", etc. without re-initialising your network.

Tasks

We will be implementing Neural Q-Learning with experience replay. The details of this algorithm can be found in the Deep Reinforcement Learning lecture slides - page 12.

Task 1

The first task is to complete the skeleton implementation for learning on the cart-pole task (https://github.com/openai/gym/wiki/CartPole-v0). Sections of code that require

completion are marked with comments containing `# TO IMPLEMENT:`. There is also generally a larger comment above these sections detailing the functionality required. Note that there is a general structure of the network and batches implied by the skeleton code. A working solution can be implemented only by adding the marked sections of code. This will allow for a standard implementation of Neural Q-learning with experience replay, however you are free to alter other sections of the skeleton code if you wish to implement a different network structure, such as feeding both the state and action values as initial inputs to the network. Just ensure that your code conforms to the api of the functions provided, i.e. don't change the function signatures of any of the functions in neural_Qtrain.py
After implementing the required functionality you should experiment with the hyperparameter settings so as to minimize the training time.
This task will be assessed on the speed with which it is able to learn the cart pole environment. Here, "speed" refers to the average training time, in terms of number of steps/episodes in the environment

N.B. while experimenting, you can set render=False to reduce the wall clock training time, but this will mean you won't be able to visually monitor progress. Please ensure you are not rendering your environment if render is set to False, as it will slow the test code. Marks will be deducted for failing to do this.

Task 2: Generalization

The aim of the second task is to generalize your algorithm so it will also work on other environments, including mountain-car andinverted-pendulum.

Copy your file from Task 1 to a new file named `neural_Qtrain_gen.py`, and try running your code on the (old and) new environments, by typing:

```
   $ python neural_Qtrain_gen.py "CartPole-v0"
   $ python neural_Qtrain_gen.py "MountainCar-v0"
   $ python neural_Qtrain_gen.py "Pendulum-v0"
```

You will most likely find that performance is very brittle, and that significant hyperparameter tuning is required for each environment. This is due to several limitations of the Task 1 model, with the most significant being that initially, the Q-network is being trained on Q-targets that have not yet stabilized. This newly updated network then produces the next Q-targets and "chases it's own tail", causing large spikes in the loss function.
To fix this, implement a "target network" that is used to calculate the one-step lookahead Q-Values and is only updated after a certain number of episodes. Updates should be simply copying the weights of the value network. For more information, see page 16 of the Deep Reinforcement Learning slides. This will increase training time proportionally to the update delay, but should also significantly increase the generalization performance. You are allowed to make changes to any code outside the `setup()` and `main()` functions to achieve this - however you should ensure the function signatures for the existing functions remain the same. Adding additional global variables is also allowed.

For the Pendulum-v0 environment:

1. You will need some modifications for the pendulum environment as it uses a continuous action space - some hints are provided in the docstring for the init function, so that the same code may be used for discrete and (discretized) continuous action spaces
2. You may want to change the value of the `max_steps` argument while experimenting, as it limits the length of episodes.

This task will be assessed on the average training speed (by episodes) on a number of the classic control tasks, including the three mentioned above.

Groups

This assignment may be done individually, or in groups of two students. Groups are defined by a number called hw3group. For convenience, we have initially set hw3group equal to hw2group, on the assumption that you would be working in the same groups as for Assignment 2. If you decide that you want to form a new group, or split an existing group into two individuals, send an email to`blair@cse.unsw.edu.au` and we will modify hw3group accordingly.

Submission

You should submit the assignment with

```
give cs9444 hw3 neural_Qtrain.py neural_Qtrain_gen.py
```

You can submit as many times as you like - later submissions will overwrite earlier ones, and submissions by either group member will overwrite those of the other. You can check that your submission has been received by using the following command:

```
9444 classrun -check
```

The submission deadline is Sunday 29 October, 23:59.
15% penalty will be applied to the (maximum) mark for every 24 hours late after the deadline.

Additional information may be found in the FAQ and will be considered as part of the specification for the project.

Questions relating to the project can also be posted to the Forums on the course Web page.

If you have a question that has not already been answered on the FAQ or the MessageBoard, you can email it to`alex.long@student.unsw.edu.au`

You should always adhere to good coding practices and style. In general, a program that attempts a substantial part of the job but does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

Plagiarism Policy

Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise and serious penalties will be applied, particularly in the case of repeat offences.

DO NOT COPY FROM OTHERS; DO NOT ALLOW ANYONE TO SEE YOUR CODE

Please refer to the UNSW Policy on Academic Honesty and Plagiarism if you require further clarification on this matter.

Good luck!