COMP9444 Neural Networks and Deep Learning

Session 2, 2018

Project 3 - Deep Reinforcement Learning

Due: Sunday 21 October, 23:59 pm
Marks: 15% of final assessment

Introduction

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

There are no constraints on the structure of network to be implemented; 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 training run 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 priveliges 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 the CartPole environment 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 CartPole environment.

Next, download the starter code from `src.zip`
All functionality should be implemented in this file:

`Neural_QTrain.py`

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`. There is no main loop, the script simply runs in the order it is declared.

Code Structure

The sections that are marked with `TODO:` are the ones you should complete. You are not required to structure your code in this way - if you have a solution that is more compact and achieves good results without them you are welcome to do that. Code that is marked with `--DO NOT MODIFY--` must not be modified and must be run as part of the script (i.e. don't surround it by a conditional that is never True). This refers to the entire code-block following the comment. Lines 105 to 120, for example, must all remain unedited. Some code is marked with neither - this code is there to help you start but you may want to modify it as you increase the complexity of your solution. You may add code anywhere else you feel is appropriate.

A breif explanation of key parts of the script follows:

Placeholders:

- `state_in` takes the current state of the environment, which is represented in our case as a sequence of reals.
- `action_in` accepts a one-hot action input. It should be used to "mask" the q-values output tensor and return a q-value for that action.
- `target_in` is the Q-value we want to move the network towards producing. Note that this target value is not fixed - this is one of the components that seperates RL from other forms of machine learning.

Network Graph:

- You can define any type of graph you like here, cnn, dense, lstm etc. It's important to consider what is the constraint in this problem - is a larger network necessarily better?
- `q_values`: Tensor containing Q-values for all available actions i.e. if the action space is 8 this will be a rank-1 tensor of length 8
- `q_action`: This should be a rank-1 tensor containing 1 element. This value should be the q-value for the action set in the action_in placeholder
- Loss/Optimizer Definition You can define any loss function you feel is appropriate. Hint: should be a function of target_in and q_action. You should also make careful choice of the optimizer to use.

Main Loop:

- Move through the environment collecting experience. In the naive implementation, we take a step and collect the reward. We then re-calcuate the Q-value for the previous state and run an update based on this new Q-value. This is the "target" referred to throughout the code.

Implementation Steps

We will be implementing Neural Q-Learning. The details of this algorithm can be found in the Deep Reinforcement Learning lecture slides - page 12. The following tasks outline the suggested way of approaching the assignment. You are not required to implement functionality in this order, and each task is not assigned individual marks - they are a guide only.

Step 1: Basic Implementation

The first thing you should do is complete the specified TODO sections and implement a basic version of neural-q-learning that is able to perform some learning. Following the provided structure, we perform one update immediately after each step taken in the environment. This results in slow and brittle learning. A correct implementation will first achieve 200 reward in around 1000 episodes depending on hyperparameter choices. In general you will observe sporadic and unstable learning.

Details of the CartPole environment are availible here.

Step 2: Batching

An easy way to speed up training is to collect batches of experiences then compute the update step on this batch. This has the effect of ensuring the Q-values are updated over a larger number of steps in the environment, however these steps will remain highly correlated. Depending on you network and hyperparameters, you should see a small to medium improvement in accuracy.

Step 3: Experience Replay

To ensure batches are decorrelated, save the experiences gathered by stepping through the environment into an array, then sample from this array at random to create the batch. This should significantly improve the robustness and stability of learning. At this point you should be able to achieve 200 average reward within the first 200 episodes.

Step 4: Extras

Once these steps have been implemented, you are free to include any extra features you feel will improve the algorithm. Use of a target network, hyperparameter tuning, and altering the Bellman update may all be benificial.

Step 5: Report

As with Assignment 2, you are required to submit a simple 1-page pdf or text file outlining and justfying your design choices. This report will be used to assign marks in accordance with the marking scheme if we are unable to run your code.

Marks:

Marks are assigned on the basis of learning speed and stability. Specifically, what is the reward at 100 episodes, and, once 200 is reached, does the performance ever degrade. Your code will be run once, so it is important that your learning rate is consistent, as well as being fast.

The provisional marking schema is:

- no run/incomplete: marks based on report, capped at 5
- Any learning shown over 500 episodes: 5 marks.

At ep 100, reward is:

- >20: 6 marks
- >40: 7 marks
- >60: 8 marks
- >80: 9 marks
- >100: 10 marks
- >120: 11 marks
- >150: 12 marks
- >170: 13 marks
- >180: 14 marks
- >190: 15 marks

If learning is not monotonic and stable (i.e. reward decreases by more than 10 between test runs): -2 marks

Restrictions

1. The only other major restriction aside from the do not modify blocks is that for each increment of the step and episode variables in the main loop, env.step() may be called only once. In other words, you may not collect multiple state action pairs in a single iteration.
2. **Do not submit a file that prints extra information to std out other than what is already present.** Of course during development this is fine, but make sure it is removed on submission.
3. The env must learn with no prior knowledge - i.e. you can't hardcode in pretrained weights or initialize the network with values that have been learned previously (although you can and should experiment with different initialization schemes.
4. You may not import any libraries other than those already included in the file, you do not need them.
5. Your algorithm cannot take an excessively long time to train - all runs will be timed out after 2mins on a high-end PC. If your code is taking longer than 5 minutes to reach 500 episodes when run on a laptop, you need to simplify your approach.
6. Try to adhere to the general spirit of this assignment. It is the final assignment and may appear difficult on first glance, but by developing an understanding of the problem and provided code structure first, a solution will fall into place quickly. Trying to shoe-horn in existing code and attempting to avoid the --DO NOT MODIFY-- blocks is a bad path to go down.

Groups

This assignment may be done individually, or in groups of two students. In the same way as for Assignment 2, groups are determined by an SMS field called hw3group. Every student has initially been assigned a unique hw3group which is "h" followed by their student ID number, e.g. h1234567. If you plan to complete the assignment individually, you don't need to do anything (but, if you do create a group with only you as a member, that's ok too). If you wish to work with a partner, one of you needs to create a group in WebCMS and add the other person. Click on "Groups" in the menu to the left. Then click "Create". Click on the menu for "Group Type" and select "hw3". After creating a group, click "Edit", search for the other member, and click "Add". WebCMS assigns a unique group ID to each group, in the form of "g" followed by six digits (e.g. g 012345). We will periodically run a script to load these values into SMS.

Submission

You should submit the assignment with

```
give cs9444 hw3 Neural_QTrain.py report.pdf
```

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

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

Additional information may be found in the FAQ and will be considered as part of the specification for the project. You should check this page regularly.

If you believe you are getting an error due to a bug in the specification or provided code, you can post to the forums on the course Web page. Please only post queries after you have made a reasonable effort to resolve the problem yourself. If you have a generic request for help you should attend one of the lab consultation sessions during the week.

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

DO NOT COPY CODE FROM THE INTERNET. This approach has a very specific structure. Copying/adapting code is likely to take much longer than understanding the logic behind the provided file. It's also plagiarism.

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!
