

XCS229ii Problem Set 3 — REINFORCE

Due Sunday, 18 April at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at <http://xcs229ii-scpd.slack.com/>
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset L^AT_EX submission. If you wish to typeset your submission and are new to L^AT_EX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be found at <https://communitystandards.stanford.edu/policies-and-guidance/honor-code>.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- **basic:** These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- **hidden:** These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the [Anaconda Setup for XCS Courses](#) to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden: Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic: Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic: Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None ← This error caused the test to fail.
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
yield
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
testMethod()
File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
result = func(*args, **kwargs)
File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
result = func(*args, **kwargs)
File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
submission.extractWordFeatures("a b a") ← In this case, start your debugging
in line 23 of grader.py.
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
assertion_func(first, second, msg=msg)
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError': {'a': 2, 'b': 1} != None
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError': {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)


1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

1. Reinforcement Learning: Policy Gradient

In this problem, you will implement the REINFORCE policy gradient algorithm to optimize on a stochastic policy that solves a simple gridded environment.

The Environment

In this problem, we will explore the Grid World environment, implemented in `GridWorld.py`. This provides a gridded rectangular world of cells, each of which has an associated reward. In each cell, the agent can perform one of four actions: [RIGHT, UP, LEFT, DOWN]. The episode is terminated when the agent enters one of the termination cells. You should feel free to experiment with different Grid World setups, but you will be graded on the one shown below. In this 5x5 environment, the agent always starts at the top left corner (0,0). There is a -1 reward at the bottom left corner (4,0) and a +1 reward at the top right corner (0,4). All other cells have a reward of zero. The non-zero reward cells ((0,4) and (4,0)) are also termination cells- the episode ends when the agent lands in one of these cells.

START 	0	0	0	TERMINATE +1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
TERMINATE -1	0	0	0	0

Your policy must learn to maximize reward in the Grid World, which for the above environment means to travel to the top right cell as quickly as possible.

Policy Gradient

Below is the REINFORCE algorithm derived in lecture. Note that any policy will work with this algorithm, as long as it is differentiable and results in a probability distribution over all possible actions. For simplicity, we will grade you on your implementation of the softmax policy. The steps below will guide you through deriving the update step for a softmax policy.

Loop:

Sample: $(s_0, a_0), (s_1, a_1), \dots$

Compute Payoff: $\text{payoff} = R(s_0) + R(s_1) + \dots$

Update Weights: $\theta := \theta + \alpha \left[\frac{\nabla_{\theta} \pi_{\theta}(s_0, a_0)}{\pi_{\theta}(s_0, a_0)} + \frac{\nabla_{\theta} \pi_{\theta}(s_1, a_1)}{\pi_{\theta}(s_1, a_1)} + \dots \right] (\text{payoff})$

Let's take a look at the payoff term. In lecture, we presented the case of a non-discounted MDP payoff. However, REINFORCE tends to converge better in the discounted case. This is because transitions closer to a reward have a higher impact on the weight update. To calculate the discounted payoff, notice that each step must now have a different payoff, which we will call G_t :

$$\theta := \theta + \alpha \left[G_0 \frac{\nabla_{\theta} \pi_{\theta}(s_0, a_0)}{\pi_{\theta}(s_0, a_0)} + G_1 \frac{\nabla_{\theta} \pi_{\theta}(s_1, a_1)}{\pi_{\theta}(s_1, a_1)} + \dots \right]$$

To calculate G_t , consider a simple episode with five steps, each earning a reward of zero except the final step, which earns a reward of one:

$$R = [0, 0, 0, 0, 1]$$

Applying a discount factor $\gamma = 0.9$, we can calculate the following discounted sum of future rewards:

$$\begin{aligned} G &= [\gamma^4, \gamma^3, \gamma^2, \gamma^1, 1] \\ &= [0.66, 0.73, 0.81, 0.9, 1] \end{aligned}$$

Here you can see that early transitions are given less payoff, which is intuitive since they likely had less impact on achieving the later reward. To calculate G_t in any series of rewards, use the formal definition below (using initial timestep t and reference future timestep k). This is known as the **discounted sum of future rewards** or the **discounted return**.

$$\begin{aligned} G_t &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \dots \\ &= \sum_{k=t}^{T-1} \gamma^{k-t} R_k \end{aligned}$$

We will now walk through deriving the policy gradient for a softmax policy (part (a)), then implement the REINFORCE algorithm in practice to solve Grid World (part (b)).

(a) [10 points (Written)]

While many policies will succeed, we will be grading you on the implementation of a softmax policy similar to the one presented in lecture. In lecture, we presented the following policy (designed for the CartPole environment):

$$\begin{aligned} \pi(S, \text{"RIGHT"}) &= \frac{1}{1 + e^{-\theta^\top S}} \\ \pi(S, \text{"LEFT"}) &= 1 - \frac{1}{1 + e^{-\theta^\top S}} \end{aligned}$$

Let's expand the softmax policy to accommodate the four possible actions of Grid World: [RIGHT, UP, LEFT, DOWN]¹.

$$\pi(s, a) = \frac{e^{\theta^\top s[a]}}{\sum_{a'} e^{\theta^\top s[a']}}$$

As presented in lecture, the state, s , represents the features used by the policy to calculate the next action (as well as a constant 1 for the bias weight). In Grid World, s features are a multi-hot representation of the agent's position. A constant 1 is provided in the first index. For a 5x5 grid, the next five indices are a one-hot vector indicating the agent's row. The next five indices are a one-hot vector indicating the agent's column. The length of the state vector is then $5 + 5 + 1$. For succinctness, we will refer to this as `num_state_params + 1`.

θ is the weight matrix and has a shape of `(num_state_params + 1, num_actions)`. Therefore, $\frac{e^{\theta^\top s}}{\sum_{a'} e^{\theta^\top s[a'()]}}$ has shape `(num_actions, 1)` and defines a probability distribution over all possible actions, as required by the REINFORCE algorithm.

When implementing this, you will be required to write code for the policy gradient weight update, $\alpha G_t \frac{\nabla_{\theta} \pi_{\theta}(s_t, a_t)}{\pi_{\theta}(s_t, a_t)}$. G_t is formally defined above. In the space below, prove the following:

$$\left[\frac{\nabla_{\theta} \pi_{\theta}(s_t, a_t)}{\pi_{\theta}(s_t, a_t)} \right]_{i,j} = \left[s_t[i] \begin{cases} 1 - \pi_{\theta}(s_t, j) & \text{iff } a_t = j \\ -\pi_{\theta}(s_t, j) & \text{otherwise} \end{cases} \right]$$

¹When it is not describing a set of objects, we use `[]` as an indexing operator instead of subscripts. This is to avoid confusion with subscripts associated with episode timesteps, t .

Hint: You may find it helpful to note that, using the chain rule in reverse,

$$\frac{\nabla_{\theta} \pi_{\theta}(s_t, a_t)}{\pi_{\theta}(s_t, a_t)} = \nabla_{\theta} \ln(\pi_{\theta}(s_t, a_t))$$

(b) [25 points (Coding)]

Now, implement the REINFORCE algorithm by completing the missing code within `REINFORCE.py`. The partially completed functions contain additional help and instructions. When you have correctly implemented the required code, run the REINFORCE algorithm with the following command `python train.py`. This will train many agents (default is 20) using REINFORCE on the Grid World, then save the visualized results, averaged over all the trained agents, to `policy_gradient_results.pdf`. A correct implementation will look similar to the image below.

