67750: Advanced Practical Machine Learning 2016/7

Practical Home Exam

TA: Alon Appleboim

1 The Task

Your goal is to learn a policy for a simple variant of the *slither* game. In the game, you are at the helm of a snake that gets bigger when it eats, and dies when it encounters walls or other players on the board. The game is endless, and when a player dies it is randomly re-placed on the board to resume its feeding.

More specifically, the board is a modulus MxN matrix of zeros, in which players are denoted by contiguous stretches of positive integers, and other objects are denoted by various other integers, e.g. food items, walls, and possibly other surprises. At each round of the game, each player can choose to continue straight, turn clock-wise (right), or counter clock-wise (left). If the head of a player encounters a wall or any part of another player (or itself), it dies and is regenerated randomly somewhere on the board. Parts of the dead player are possibly recycled into food items, so in a competitive game, it can be useful to kill other players and eat their remains. When a player's head encounters other items on the board, they might have positive effects (e.g. cause growth, give bonus points), or negative effects (slow down, score penalty, etc.).

At each round, the score is the length of the player, with possible bonuses or penalties from food items encountered during the last round. In case of death, a large fixed amount of points is reduced. Thus, the goal is to eat as much as possible without dying, while avoiding rotten food.

2 Code and Environment

In the provided zip file, the only relevant files for you are the slither.py, the README, and the policies directory/package.The rest of the files/directories are used by the evaluate.py script. It is an example of how your code will be evaluated - it's kinda ugly, but if you want to get a feeling, or even test your code you're welcome to play around with it.

2.1 Basic Game Design

Each policy in the game is a stand-alone process (a multiprocessing.Process instance) that communicates with its player via python Queue objects. All this is encapsulated, so in order to implement a policy, all one has to do is subclass the "Policy" class, and override five simple methods: "act", "learn", "init_run", "cast_policy_args", and "get_state". These are documented in the "base_policy.py" file, and a basic implementation example of is given in the "policy_0.py" file.

At each round of the game, the policies receive the current game state (via the "act" method) and the game awaits their responses for a certain period of time (∼10 milliseconds). If no action was supplied by that time, the player will continue to move in its current direction. At each such round, the policies are also provided with the reward resulting from the previous round (via the ”learn” method). Since the communication is asynchronous, each such transaction is supplemented with the absolute time (round index), so in case a round, or several rounds were missed due to calculations on the policy's side, the policy can deal with time gaps as it sees fit.

A simple policy (”AvoidCollisions”), is provided for your convenience, and is set as the default policy of players in the game. Another policy (”Policy2345678”) shows how a submitted policy should look like.

2.2 Starting a Game

To start a game, simply run:

```
$ python3 slither.py
```

This will start a game with the default settings. Note that all options and default settings can be viewed by asking for help:

```
$ python3 slither.py -h
```

Important options include the game duration, number of players and their policies, whether it's rendered or not, whether to write a log or not, whether to record the game for a later playback, and others.

For example:

```
$ python3 slither.py -D 1000 -r #play for 1000 rounds, and don't render
$ python3 slither.py -p "last\_game.dat" #playback last game
```

2.3 Advanced Instructions and Features

Note that since each policy is an independent process, context-managed object, such as tensor flow sessions, and other environment resources are not shared between policies, or even with the game. For this reason, any such object should be initialized when overriding the Policy.run_init() method. This method is called only when the policy process is created. The real constructor of the policy (__init__) is better left untouched, as it's overriding the multiprocessing.Process class, so do this at your own risk.

There are several features provided by the environment for easier coding and/or debugging. Logging is provided in a centralized form by the Policy.log function. All players log to the same file, but it's easy to look at specific log entries using shell commands such as ”cut”, ”grep”, and ”awk”. Do not rely on other logging mechanisms (e.g. file creation permissions might be limited, and logging will fail). Also, a game recording option is provided (and turned on by default

Another feature provided by the environment is policy argument passing. For example, suppose you want to test several learning rates. Instead of running 3 different games and changing a global rate constant in each run, you can have the rate be a member of your policy, allowing every policy instance to have a different learning rate. Now the question is how to pass the desired rate of each policy to the constructor? Simply enough, all you have to do is override the Policy.cast_string_args method, and the rest is completely generic. For example in the above case, all one has to do is the following:

```
class MyPolicy(Policy):

    def cast_string_args(self, policy_args):
        policy_args['r'] = float(policy_args['r']) if 'r' in policy_args else 0.001
        return policy_args
```

And start the game as follows:

```
$ python3 slither.py -P "MyPolicy(r=.1);MyPolicy(r=.01);MyPolicy(r=.001)"
```

This will generate three independent policies that will have an ”r” member with the three different values.

Another important feature you should take advantage of is the ”get_state” method. This method allows the game to instruct you to save your current state. The game can then pass this state to another instance of the your policy so the experience of a learning task is not lost. For example, if your policy is used in consecutive games, you can have the option to use a previous model as a starting point for a current game (see below).

3 Guidelines and Tips

Start early. As mentioned in class, reinforcement learning takes time. In addition, this is the first time this kind of exercise is given in the course and there will be integration issues and bugs. The sooner you find out about them, the less they'll affect you. There is a forum dedicated for the hackathon, questions/comments will **only** be replied to in the forum.

Learning a policy for this task can be very complex. At the most basic level, a policy should avoid death, but when looking a bit further ahead, a policy should avoid getting into a snake-knot, and aim at eating good food, while avoiding rotten food. Additionally in an aggressive game, the policy could also include trying to trap/block other players. Therefore, when building your policy it's recommended that you start from a very basic setting - one player, no walls, only good food. Once you have a reasonable player, incrementally add complexity to see which model features and/or state representations help your learner learn faster, and achieve better results.

Importantly, the exercise is not a coding exercise but a learning exercise, therefore, to discourage hardcoding behaviors into your policy, you can't rely on any game rule, other than the basic physics, the existence of obstacles (static and dynamic) that cause death when encountered, and positive/negative items on the board. These rules will be consistent throughout the evaluation process, but may differ from the rules you'll use while training. All values may change, and in certain cases, values of certain items may even change depending on the game context.

There are several aspects that are worth special attention, as they are at the base of any learning algorithm:

The representation of the world is extremely important. If you are a snake playing in a 1000x1000 world, there's a very small part of the world you're interested in. If your world representation involves a 1000x1000 vector, then besides the excessive amounts of computation you'll have to perform, it will be much harder for your learner to home in on the important aspects of the game. However, This is not to say that far-off information is irrelevant, just that serious thinking should be put into the way you represent the world. This should also be coupled to the model you're trying to learn - certain features are more amenable to learning with linear models for example, while other representation will be easier to learn with non linear functions. In other words - use your prior understanding of the task and the algorithm you're using to achieve the simplest and most successful learner.

Another important aspect is the "learning units" - what batches (or episodes) of samples will you use for learning, and how. What will be the effect of learning with small batches vs larger batches, and what about other options? Also, the weighing of actions/observations when learning ("advantage"/"discounting"), can have a significant effect, as Shai mentioned in class. The exploitation/exploration tradeoff might also have a major contribution, depending on the environment. There are many more such considerations, some were discussed in class, some are intuitive, and some can be found online, but in any case - these are the basics of any successful learner, so they're well worth your time.

Notice that with the way the game is executed, learners that require too many computations will miss opportunities to act and collect data, so unlike in typical learning tasks where learning time is not a bottle-neck resource, in this case it might have an effect on overall performance in a finite game. Also, note that the memory footprint of your policy should not exceed 1GB.

You can use any python library, but it's your responsibility to make sure it's installed in the environment in which the interpreter will be executed. The environment is located in:
/cs/wetlab/Alon/hackathon/slitherenv/
To install packages, you can connect to the environment via the "source" command, and then install any package as you would on a regular python environment. Make sure to make all files completely public using "chmod -R 777".

```
$ source /cs/wetlab/Alon/hackathon/slitherenv/bin/activate
(slitherenv)$ pip3 install important_library
(slitherenv)$ deactivate
$ chmod -R 777 /cs/wetlab/Alon/hackathon/slitherenv
```

4 Submission

You are required to submit exactly 2 files: a README file and a python3(!) code file.

The code file, named "policy_⟨id⟩.py", should contain a single class named Policy⟨id⟩ that inherits from the Policy class. Do not rely on any argument-passing for the instantiation of your policy, besides the "load_from" that may or may not be provided, depending on the game. All policies will be placed in a folder and will be loaded as in the slither.py code you are provided with. Write only one of the IDs in the file/class name (the only purpose of this id is to prevent file/class name overload).

Your README file should conform to the following format (see provided example):

1. First line is a semicolon delimited line describing your learner: class_name;short_name;short_description
2. both IDs
3. A paragraph about the state representation in your code (i.e. what features do you feed into your learner)
4. An explanation of the model you are learning, if it's generative, explain the assumptions, if it's discriminative, explain its architecture
5. A paragraph about the exploration/exploitation tradeoff in your policy
6. A short description the tests and results you got when you trained your policy
7. Other stuff you want to note

5 Testing and Evaluation

The grading will be split between the answers in your README, and the performance of your learned policy. To test your learners, they will be first trained and evaluated on a board without other players (for at least 100,000 rounds, hopefuly much more). After this stage, and given their state at this point, your policies will be trained against AI snakes, again for at least (100,000 rounds).Finally, computation time allowing, your policy willbe evaluated against policies of other students (see the "evaluate.py" script to get a sense of the testing the policies might be subject to).

Good luck