HW3

Due: 05/03/2023 at 11:59pm EST

Collaboration Policy: As mentioned in the syllabus, you are encouraged to work in pairs. If you have a partner, each pair should submit **one** submission on gradescope. Therefore, please make sure that in your source code (preferably in a comment above the class name), you list the names and BU IDs of each person who worked on it!

1 Setup

After downloading hw3 from Piazza and unzipping the .zip file, you should notice that there are three directories in the starter code: `src`, `data`, and `lib`.

Moving Files to the Right Place

Please create a new package called `hw3` in your workspace (i.e. in Eclipse, IntelliJ, etc.) and place the two files in it. I will be providing better agents to compete against (so far just `BadCombatAgent.java`) as my agents continue to train. You will be completing `QAgent.java`.

Back to the top level of your extracted .zip file, if we go inside the `data` sub-directory, you will see just one sub-sub-directory also called `hw3`. Fear not, this is intended, and is only to help your `data` directory be less cluttered. Please copy the `data/hw3` directory over to the `data` folder you are using in your workspace.

The `lib` directory is new, and this should contain a single file `hw3.jar`. Inside of this jar is all of the neural network functionality (and matrix functionality) that I have created for you. I will be publishing the documentation on this tomorrow, but it is late and I am tired. Please add this jarfile to your `lib` directory in your workspace, and then add it to your build path like to did to `sepia.jar` many weeks ago.

Modifying the `.xml` files

If you want, you can modify the `CombatConfig.xml` files. I have left comments in that xml file for how to change the arguments given to your `QAgent`.

Creating `hw3` Run Configuration

We will at the moment only need to create a single run config, one to run `CombatConfig.xml`. I suggest calling this run config "QAgentvsBadAgent" or something along those lines.

The Goal

Your goal is to write an agent that implements Q-learning using a feed-forward neural network. I have already implemented quite a bit of machinery for you, however, you still need to implement the following:

- The network architecture itself. How many layers, the number of units per hidden layer, and what activation functions they use are entirely up to you. Remember, a larger network is theoretically more powerful (i.e. it can learn more complicated functions), but takes way longer to train and is more sensitive to overfitting. Since the Q-function should always spit out a scalar (i.e. output dimension = 1), if you choose to use an activation like `Sigmoid` or `Tanh` for this layer, I recommend determining what the largest possible reward value is (i.e R_{max}) and using that as a scaling coefficient for the activation (passed into the constructor of the activation function).
- The reward function. How will you decide to reward a unit or punish a unit?
- The feature representation of a state. I recommend viewing the network from the perspective of getting a Q-value conditioned on a specific attacking unit (i.e. one of your units) when it is considering attacking a unit (i.e. one of the enemy units). In this context, we don't really need to consider making the state contain all information about every unit on your team, and every unit on the other team. Instead, this lets us make more generic features, such as "how many of my teammates are attacking that unit?" or "how many of the enemy units are left alive?" etc. (don't forget to normalize your input, this may help with performance). With this view, your Q-function produces a value for how good is it for the attacking unit to attack the target unit, which can certainly capture "ganging up" on enemies, etc.

2 Neural Network Architecture (20 points)

Please decide what your architecture should be. You are free to change it however many times you want, and to explore with different kinds of architectures. If you want, you are more than welcome to extend my neural network library to add layers types, etc., but this is not necessary.

3 Reward Function (40 points)

As described above, please think hard about how you want to reward/punish a unit. The performance of your agent will be very sensitive to this.

4 Feature Representation (40 points)

As described above, please think hard about how you want to describe the game to the Q-function. I highly suggest you view the Q-function as described above: this will make your agent way easier to reason about and to program. The performance of your agent will also be very sensitive to this.

5 Grading

We will be grading your submission based not only on the quality of your code, but on the quality of the performance that your agent has. In this assignment, quality of code includes how easy it is for us to read your code (please be descriptive and leave comments), the algorithm itself (for correctness), and the choices behind your reward function and feature representation. To earn full points, you must be able to demonstrate that your agent has learned to fight (as shown through a learning curve, the data of which is generated for you if you play enough games). A reminder that any solution that compiles and is turned in on time will participate in the extra credit tournament!

As always, please feel free to ask us any questions you may have! Good luck!