Programming Assignment 2: Tetris

Due: Wednesday 05/01/2024 @ 11:59pm EST

The purpose of programming assignments is to use the concepts that we learn in class to solve an actual real-world task. We will not be using Sepia for this assignment: I have developed a game engine for us to use. In this assignment we will be writing agents to play Tetris.

RL with Neural Networks

Reinforcement Learning on play difficult games like Tetris are slightly more complicated than we have talked about so far in lecture (but we will get there shortly). This is mostly due to the fact that there are too many unique states possible in the game of tetris, so calculating a policy (which, remember, is a function that maps states to actions) is not possible to store explicitly. So far, the policies we've talked about can be implemented as giant lookup tables, which is not feasible when playing games like Tetris.

Instead, we turn to neural networks, as they are function approximators but have much smaller memory requirements. We can use a neural network to compute a policy by asking the neural network to convert a (state, action) pair into a number called a **q-value** (bigger is better). To calculate the action in a certain state, we ask the neural network to calculate the q-value for each of the actions in that state and then choose the action with the largest q-value. Thus, we can compute a policy without paying the infeasible memory cost of a proper policy lookup table.

As a bonus, neural networks "learn," meaning that we can try to generalize tetris behavior on states that we have not explicitly seen before but are similar to the ones that we have. Learning here is a double edged sword though. When we perform gradient descent to update the neural network's parameters, we run the risk of the gradient adjusting the parameters away from any combination that generated good behavior on prior experience if the batch of data we generated the gradients from does not contain those prior experiences. In other words, when we train on a batch of data, we can forget what we have learned in the past. To solve this, the lifecycle of using a neural network is often a form of policy iteration (except we don't know the transition probabilities anymore):

1. Play a bunch of "training games" and record the trajectories seen in those games.
2. After these "training games", update the neural network with the collected experiences.
3. Play a bunch of "evaluation" games with the updated neural network to evaluate its new performance.

This lifecycle is then repeated for as many cycles as we want (typically a massive amount of times). In our setting, we will call one iteration of this lifecycle a **phase**, and we will repeat this **phase** until our neural network starts to get really good at playing tetris.

1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/pa2/lib/tetris.jar` to `cs440/lib/tetris.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/pa2/lib/argparse4j-0.9.0.jar` to `cs440/lib/argparse4j-0.9.0.jar`.
  This is a jarfile that `tetris.jar` depends on. It provides similar functionality to Python's `argparse` module. The documentation for `argparse4j` can be found here.
- Copy `Downloads/pa2/src` to `cs440/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/pa2/tetris.srcs` to `cs440/tetris.srcs`.
  This file contains the paths to the `.java` files we are working with in this assignment. Just like in the past, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- Copy `Downloads/pa2/doc/pas` to `cs440/doc/pas`. This is the documentation generated from `tetris.jar` and will be extremely useful in this assignment. After copying, if you double-click on `cs440/doc/pas/tetris/index.html`, the documentation should open in your browser.

2. Test run

If your setup is correct, you should be able to compile and execute the following code. A window should appear:

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @tetris.srcs
java -cp "./lib/*:." edu.bu.tetris.Main

# Windows. Run from the cs440 directory.
javac -cp "./lib/*;." @tetris.srcs
java -cp "./lib/*;." edu.bu.tetris.Main
```

NOTE: The commands above will **not** run your code. There are several command line options available to you. If you want to see the exhausive list, please add a `-h` or `--help` argument to the command line and see the help message! Most of these command line arguments are your way of configuring the way you train your model (such as learning rate, batch size, etc.).

Task 1: `TetrisQAgent.java` (100 points)

Please complete the implementation of `TetrisQAgent.java`. In this file, I am asking you to devise and implement several different methods which are crucial for a good RL agent. I have listed them below in order of priority:

1. method `getQFunctionInput`. This method takes a `GameView` object which contains all the information about the current state of the game, as well as a `Mino` object. A `Mino` (short for `Tetramino` are the pieces that you can place in a game of tetris. The `Mino` provided here as an argument is a possible resting place for the `Mino` that needs to be placed on the board.

   Your method needs to convert these two objects into a row-vector which will be used as input to your neural network. You are responsible for engineering the representation, i.e. for engineering the conversion of game information into a vector. You want you vector to contain all the information necessary for the neural network to provide a meaningful ranking (q-value) of the `Mino` placed on the board.

2. method `getReward`. You have to engineer your reward function. This method takes a single `GameView` object as input, and your method should calculate the reward for being in that state of the game. If this state is bad, you should produce a small number (you are allowed to go negative), and if this state is good, you should produce a large number. I encourage you to be creative when devising your reward function, we want to determine "goodness" and "badness" that correlates to actual good tetris behavior.

3. method `initQFunction`. Once you have designed your vector representation and figured out your reward function, you will now need to actually build your neural network object. You are only allowed to use feed-forward neural networks in this project: I could not get convolutions working in time for use this semester. The size of the input vector is now fixed: you had to decide this when implementing `getQFunctionInput`, so your input layer should expect a vector of that size. The output of your neural network should be an unbounded (i.e. no output layer activation function) scalar value (i.e. the q-value). This means that your output layer should only have a single unit in it.

   The difficulty part will be how many hidden layers do you use, how big are each hidden layer, and what are their activation functions? I have implemented a little library of layers for you to take a look at, just know that convolutions are not ready yet, so please don't use them.

4. methods `shouldExplore` and `getExplorationMove`. These methods are how we implement curiosity into our agent. As your agent learns, it will start to discover actions that it thinks are good, and start suggesting them more frequently. This can be a blessing and a curse. The blessing is that the model is doing things that are "good" (as measured by the reward function). The curse is that as we continue to follow the policy, we will stop exploring and gaining novel experiences, and we risk getting stuck. This isn't really a problem if our policy happens to stumble upon really good actions, however this is unlikely, and instead our policy will get stuck on (what it deems to be the best its seen so far but are actually) mid-tier actions.

   To encourage the agent to ignore the policy (sometimes) and instead explore for novel experiences, we need to implement a version of curiosity. The first method, `shouldExplore` should return `true` when the agent determines that it should explore in this state (and ignore what the policy recommends), and `false` otherwise. The second method, `getExplorationMove` is how we generate an action that will (hopefully) lead to a novel experience (assuming we have decided to ignore the policy in the first place). You should implement some notion of curiosity here.

To earn full credit, you must demonstrate that your agent has learned. If your agent can score an average of 10 points when playing 500 games of tetris, then I will award you full credit.

Task 2: What to Submit

This time you will need to submit more than just your Java source code. As your model trains, my code will write the parameters of the model to files on your machine after every phase. You need to pick one of these files (ideally the file that corresponds to the model that has the best performance), rename it to `params.model`, and submit it along with your `TetrisQAgent.java` file on gradescope.

Task 3: Extra Credit (50 points)

If your agent can earn an average of 30 points when playing 500 games of tetris, then I will award you full credit.

Task 4: Tournament Eligibility

In order for your submission to be eligible in the tournament, your submission must satisfy all of the following requirements:

- Your submission must be on time.
- You do not get an extension for this assignment.
- Your agent compiles on the autograder.
- Your agent can play 1000 games of tetris and earn an average of 20 points or more.