Programming Assignment 3

Assigned: Oct. 27
Due: Nov. 15

In this assignment you will build a program that learns to play a simple game of chance in somewhat the same way that the AlphaZero program learned to play chess, Go, and other games. (We will discuss the differences later in the semester. This assignment includes the easy part of AlphaZero; what it leaves out is the hard part.)

The game is a variant of the card game "Blackjack". Two players alternately roll dice, and keep track of their total across turns. They are each trying to reach a sum that lies in a specified target, between a fixed low value and high value. If a player reaches a score in the target range, they immediately win. If they exceed the high value, they immediately lose.

The players can choose the number of dice to roll on each turn, between 1 and a fixed maximum.

The game thus has four parameters:

- *NSides,* The number of sides of the die. The die is numbered 1 to *NSides* and all outcomes are equally likely.
- *LTarget*, the lowest winning value.
- *UTarget*, the highest winning value.
- *NDice,* the maximum number of dice a player may roll.

For instance, with *NSides*=6, *LTarget*=15, *UTarget*=17, *NDice=2*, the following are two possible games. (The players are not necessarily playing well in the games below, just legally.)

Game 1:

Player A rolls 2 dice, which come up 5 and 6. A total: 11.
Player B rolls 2 dice, which come up 3 and 4. B total: 7.
Player A rolls 2 dice, which come up 5 and 5. A total: 21. A loses.

Game 2:

Player A rolls 2 dice, which come up 3 and 4. A total: 7.
Player B rolls 2 dice, which come up 5 and 6. B total: 11.
Player A rolls 2 dice, which come up 3 and 1. A total: 11.
Player B rolls 1 die, which comes up 4. B total: 15. B wins.

Game 3:

Player A rolls 2 dice, which come up 3 and 4. A total: 7.
Player B rolls 2 dice, which come up 1 and 4. B total: 5.
Player A rolls 2 dice, which come up 2 and 5. A total: 14.
Player B rolls 2 dice, which come up 3 and 5. A total: 13.
Player A rolls 1 die , which comes up 1. A total: 15. A wins.

The learning algorithm to be used is as follows: The machine play a series of games with itself, playing both sides. Initially, both players play randomly; as the series progresses, they hopefully play better and better.

The program maintains two three-dimensional integer matrices of size $LTarget \times LTarget \times [NDice + 1]$. The matrices are `WinCount[X,Y,J]` and `LoseCount[X,Y,J]`, where `X` is the current point count for the player about to play; `Y` is the point count for the opponent; and `J` is the number of dice the current player rolls. The value of `WinCount[X,Y,J]` is the number of times that the current player has eventually won when the state was $\langle$`X`, `Y`$\rangle$ and the current player rolled `J` dice. `LoseCount[X,Y,J]` is the number of times that they lost.[1]

After each game, these two matrices get updated, reflecting the result of the game. For instance, after completing Game 1 above.

Game 1:

Player A rolls 2 dice, which come up 5 and 6. A total: 11.
Player B rolls 2 dice, which come up 3 and 4. B total: 7.
Player A rolls 2 dice, which come up 5 and 5. A total: 21. A loses.

the result will be that `LoseCount(0,0,2)`, `LoseCount(11,7,2)`, and `WinCount(0,11,2)` will all be incremented by 1.

After completing Game 3:

Game 3:

Player A rolls 2 dice, which come up 3 and 4. A total: 7.
Player B rolls 2 dice, which come up 1 and 4. B total: 5.
Player A rolls 2 dice, which come up 2 and 5. A total: 14.
Player B rolls 2 dice, which come up 3 and 5. A total: 13.
Player A rolls 1 die , which comes up 1. A total: 15. A wins.

the result will be that `WinCount[0,0,2]`, `LoseCount[0,7,2]`, `WinCount[7,5,2]`, `LoseCount[5,14,2]`, and `WinCount[14,13,1]`, will all be incremented by 1.

Game Play

Game play proceeds as follows. Suppose that the current state is $\langle$`X`, `Y`$\rangle$.
Let $K = NDice$, to simplify writing formulas.
For $J = 1 \ldots K$ let $f_J =$ `WinCount`$[$`X`, `Y`, `J`$]/($`WinCount`$[$`X`, `Y`, `D`, `J`$] +$ `LoseCount`$[$`X`, `Y`, `J`$])$.
(If the denominator is 0, then $f_J = 0.5$).
Let B be the value of J with the highest value of f_J (break ties arbitrarily).
Let $g = \sum_{J \neq B} f_J$ (sum over all J that are not B)

Let $T = \sum_{J=0}^{K}$ `WinCount`$($`X`, `Y`, `J`$)$ + `LoseCount`$($`X`, `Y`, `J`$)$, the total number of games that have gone through state $\langle$`X`, `Y`$\rangle$. Let M be a hyperparameter set in the input. (As we will discuss below, M controls the "explore/exploit" trade-off.)

Then

[1] Since you are not allowed to play 0 dice, the value of both matrices for `J=0` is always 0. Why not just eliminate that column of the matrices, and have `J` be the number of dice rolled minus 1? Because the saving in memory is not worth the cost in off-by-one errors.

The player rolls B dice with probability

$$p_B = \frac{Tf_B + M}{Tf_B + KM}$$

For $J \neq B$, the player rolls J dice with probability

$$p_J = (1 - p_B) \cdot \frac{Tf_J + M}{gT + (K-1)M}$$

For instance, suppose that $NDice = 3, M = 4$, the state is $\langle 2, 3 \rangle$ and

```
WinCount[2,3,1] = 0.   LoseCount[2,3,1] = 2.
WinCount[2,3,2] = 3.   LoseCount[2,3,2] = 1.
WinCount[2,3,3] = 1.   LoseCount[2,3,3] = 1.
```

Then
$f_1 = 0/(0+2) = 0$. $f_2 = 3/(3+1) = 0.75$. $f_3 = 1/(1+1) = 0.5$.
$B = 2$ (the best move).
$g = 0 + 0.5 = 0.5$.
$T = 0 + 2 + 3 + 1 + 1 + 1 = 8$. So:

$p_2 = (8 \cdot 0.75 + 4)/(8 \cdot 0.75 + 3 \cdot 4) = 0.5556$.
$p_1 = (1 - 0.5556) \cdot (8 \cdot 0 + 4)/(8 \cdot 0.5 + 2 \cdot 4) = 0.4444 \cdot (4/12) = 0.1481$.
$p_3 = (1 - 0.5556) \cdot (8 \cdot 0.5 + 4)/(8 \cdot 0.5 + 2 \cdot 4) = 0.4444 \cdot 8/12 = 0.2963$.

You play a specified number of games and then output the winning strategies with the associated probabilities.

Choosing the action: Explore/exploit trade-off

When one is combining actions with learning in an unknown environment, there is generally an "explore/exploit" trade-off to consider. That is, how much effort should you spend exploring all the possibilities, versus exploiting actions that have been found to be successful. Generally, when you are starting in a new environment, you want to favor exploring; as you get to know the environment, you increasingly favor exploiting it. For instance, if you move to a new city and enjoy eating out, then at first, you try lots of possibilities; after you've been there for a while, you increasingly go to old favorites.

In this assignment, the way in which the action is chosen in a state goes from exploring (choosing an action at random) to exploiting (choosing the best action) gradually, at a speed controlled by the hyperparameter M; the larger M, the longer it takes to change from one regime to the other.

Specifically the parameter M affects the probabilities that govern the choice of action. We have the following two formulas above: The player chooses to roll B dice with probability

$$p_B = \frac{Tf_B + M}{Tf_B + KM}$$

For $J \neq B$, the player rolls J dice with probability

$$p_J = (1 - p_B) \cdot \frac{Tf_J + M}{gT + (K-1)M}$$

Suppose that M is reasonably large; say $M = 100$. Consider the state of things after the player has only played a small number of games so that Tf_B and gT are both much smaller than M. In that case the fraction $(Tf_B + M)/(Tf_B + KM)$ will only be slightly larger than $1/K$ and the expression $(1-p_B)\cdot(Tf_J+M)(gT+(K-1)M)$ is only slightly smaller than $1/K$. (Note that, since p_B is about $1/K$, $1-p_B$ is about $K-1/K$ so we have $p_J \approx ((K-1)/K)\cdot(M/(K-1)M) \approx 1/K$.) That is, the player has only a slight preference for playing the moves that have worked out well for them so far; they are choosing nearly at random.

Now, consider the state of things after the player has played many games so that Tf_B is much larger than M. In that case, the fraction $(Tf_B+M)/(Tf_B+KM)$ is nearly 1, so the player chooses their best move almost all the time, and only occasionally chooses an inferior move.

Input/Output

The input is just the parameters of the experiment: $NSides$, the number of sides of the dice; $LTarget$, the lowest winning score; $UTarget$, the highest winning score; $NDice$ the maximum number of dice that can be rolled per turn; M, the hyperparameter described above, and $NGAMES$ the number of games to play. M is a floating point number; the rest are integers.

Output two $LTarget \times LTarget$ arrays; the correct number of dice to roll in state $\langle \texttt{X}, \texttt{Y} \rangle$ and the probability of winning if you roll the correct number of dice. The probability of winning if you roll `J` dice in state `X,Y` is estimated as `WinCount[X,Y,J]`/(`WinCount[X,Y,J] + LostCount[X,Y,J]`) (Output -1 if the denominator is 0.)

Sampling from a given distribution

If you are given a distribution $p_0 \ldots p_k$, then the way to sample from the numbers $0 \ldots k$ with that distribution is to execute the following code:

$u_0 = p_0$
for $(i = 1 \ldots k)$ $u_i = u_{i-1} + p_i$.
$x =$ **rand**(); % Uniformly distributed between 0 and 1.
for $(i = 0 \ldots k-1)$
    if $x < u_i$ return i;
return k; % You don't want to count on u_k being exactly 1, because of round-off error.

In Python there is a library function `random.choices()` to do this, which you can feel free to use. It is described here: `https://docs.python.org/3/library/random.html`

Limits on the quality of results

More or less, if enough games are played and if the parameter M is not set too small, then the answers should more or less converge to the true answer. But there are a number of important limitations on that:

A) A state may be such that the player has probability 0 of winning whatever he does. For instance if the target range is [10,20], $NDice = 2$, $NSides = 3$, it is player B's turn and he is behind 2 to 9, then B has lost. In that case, of course, the "best move" is arbitrary.

B) A state may be mathematically unattainable starting from [0,0]. For instance, if $NDice = 2$, $NSides = 2$, then there is no way to attain the state $\langle 5, 0 \rangle$.

C) A state may be unattainable unless one or both players play really stupidly. For instance, suppose the target range is [10,20], $NSides = 3$, $NCards = 2$. Then obviously, both players should always roll 2 dice, because you can't overshoot the upper bound. The state $\langle 1, X \rangle$ for any X is unattainable unless the first player rolls 1 die. Such states will presumably only be achieved in early games in the sequence, because the program will learn not to make the stupid moves. Therefore, the probability estimates for these may be extremely far off, and the policy may be wrong.

D) A state may be attainable if both players are playing optimally but very unlikely . For instance, suppose that $NSides = 4$ and $NDice = 10$. Then the state [0,40] is possible, if the first player rolls ten 4s, but will occur only in 1 in a million games, so it would take an enormous number of plays to build up reliable statistics. Again, with reasonable number of games played, the computed probability may be far off and the policy may be wrong.

E) If the probabilities for success for two different actions in a given state are close, then it is quite possible that the recommended action will be not the truly optimal one.

So the most that you can expect is this: For states which will occur reasonably frequently if both players play reasonably well, the probability matrix should be fairly accurate, and if the best move is significantly better than the alternatives, then the policy matrix should recommend it.

Examples

I will post a few examples later. Because of the above considerations, and because this is a random process, you should expect that your answers will agree with mine (or with your classmates, or with your own results in multiple experiments) only approximately, and with the above caveats.