Homework 6 (due January 23, 2018) – Reinforcement learning

I) warmup: Tic-Tac-Toe in Python

To get used to the main concept, we'll start with a simple example: Q-learning for Tic-Tac-Toe. This can be done without neural network, so we won't use tensorflow in this part.

Tip: There are many possible ways to implement Q-learning. Part of the homework is to find suitable ones yourself. Probably, the first you come up will not be the most suitable one, so be prepared to throw away (part of) your code at some time and try something else.

I.1) Tic-Tac-Toe

If you don't remember it, familiarize yourself with the rules of Tic-Tac-Toe (`https://en.wikipedia.org/wiki/Tic-tac-toe`)

I.2) Representations

Choose a representation of the playing board and the action space. You should be able to

- read out/overwrite individual locations by an index
- convert the board into a *hashable* type (typically *integer* or *string*)

Write a subroutine that checks if for any board configuration, if Player 1 has won (output +1), Player 2 has won (output -1). Otherwise, output 0.

Write a subroutine that checks for any board configuration if there's any legal moves (=actions) left to make.

Write a subroutine that for a given board configuration, move location, and player ID updates the board accordingly, and that computes the reward of the action (-1/0/+1 depending on the value of the new board configuration).

I.3) Random play

Write a program that plays randomly. For a given number of rounds:

- create an empty board
- repeat
    - let Players 1 and 2 alternatingly make random (legal) moves
- until there are no moves left or one of the players has won the game

a) run for 10000 games (if this takes longer than a minute or two, better rethink your implementation).

b) evaluate in three ways: i) record sequence of all rewards and plot its cumulative sum as a curve,

ii) compute the *total reward* (i.e. the sum of the sequence), iii) build a histogram of how many wins/draws/losses there were for Player 1 won

iii) modify the code such that in each round it is randomly chosen if Player 1 or Player 2 move first.

iv) reevaluate as in b)

I.4) Stronger opponents

a) Replace the random Player 2 by a *smart* one: in any situation, it checks if there's an immediate winning possibility available. If yes, it takes it. If not, it plays randomly. Let it run again a random player 1, what's the outcome?

b) Make the opponent even *smarter*: it should still make winning moves if it can, but if can't it should check if the opponent has a winning move and if yes, it blocks that. If not, it plays randomly.

c) Perform a tournament of all all options (random, smart, smarter) for Players 1 and Player 2 and record the *total reward* in each case.

To facilitate this, you might want to refactor your code such that the players' strategies can be switched without having to every time change a lot of code.

d) (optional) Now we make the opponent *supersmart*: it does the check of the *smarter* player, but instead of just playing randomly, it tries to move to the center. If that's impossible, it tries a random corner. If all of those are impossible, it picks a random remaining localtion.

I.5) Q-learning with random exploration

We now integrate Q-learning, such that Player 1 is able to learn from its experience.

Create a data structure that allows table lookups for the Q function. For any board state it should output 9 values, one per action (including impossible ones, which we'll filter out by other means).

Tip: a convenient option is a *defaultdict* from Python's *collections* package. It can be indexed by any hashable type and the *default* allows recieving a default value (e.g. all zeros) even for entries that haven't be written yet.

Write a routine `updateQ(s,a,s',r)` that performs the *Q-learning update*: given an old state s, an action a, a new state *s'* and a reward value r

- $Q(s)[a] := r + \gamma \max_{a'} Q(s')[a']$ # score for action a in situation s is *reward* plus score of best continuation

where γ is an adjustable parameter (for the experiments, use $\gamma = 0.9$).

Integrate the Q-learning update in the above random play routine. For this, interpret Player 1's move as the action, and Player 2's move (if one is possible) as part of the reaction by the environment.

I.6) Putting things together

Now, we'll use the learned Q-function so Player 1 learns to play better over time.

a) Implement a routine that makes *greedy* moves for Player 1:

- in any situation, pick the action a with highest values of $Q(s)[a]$

Integrate the greedy Player 1 into the code from I.5) and let it play against random, smart, and smarter Players 2.

a) Compare the total rewards of a) to the results from I.4.c) for the 'smarter' Player 1. Is Q-learning doing better or worse?

b) Now plot the curves of cumulative rewards. What do you observe? What would happen if you had played 20.000 instead of 10.000 games?

c) (optional) Can you imagine a situation in which Q-learning with the *greedy* player does less well than it could? How could you prevent that?

II) Tic-Tac-Toe in Tensorflow

We now switch from table-based Q-learning to using a neural network. Note that technically this makes little sense for Tic-Tac-Toe, as the simpler table-based system works well enough. We just use it here to practice the tensorflow integration.

II.1) Q-function in tensorflow

Instead of the above look-up table, implement a Q function (approximator) using a neural network.

- as input, feed the board configuration (in the original form) via a placeholder
- convert the board to a one-hot encoding (if it wasn't already) and flatten it
- create one densely connected layer with 10 hidden units and a nonlinearity of your choice
- create one densely connected layer with 9 output units (that corresponds to the Q-values of different actions)

For the fully connected layers, use an initialization with very small values, such that the Q-values don't vary too much initially.

II.2) updating the Q function

Updating the Q function is the main problem in deep Q-learning, as one cannot simply overwrite values in the function table by others. Instead, we have to run a few steps of network training to make the specific output we want to influence closer to the value we want it to have. There's multiple ways to do this, here is a simple (though not very elegant) one:

- create a `tf.float32` placeholder t for the target value (the right hand side of the Q-update)
- create a `tf.int32` placeholder a for the index of the action you want to affect
- define a tensor that is the squared difference between the a-th entry of the Q-function and t
- create an operation for mimimizing this squared difference (e.g. using a `tf.train.GradientDescentOptimizer`. What's a good learning rate? Your guess is as good as mine. . .)

Write a new `updateQ(s,a,s',r)` routine that

- evaluates $Q(s')$ (using `sess.run`) and writes it to a python vector $qnew$
- computes the desired right hand side (in python), $r + \gamma \max_{a'} qnew[a']$
- call the above minimization operation a certain number of steps (e.g. 10) to achieve the Q-update

(optional challenge): replace as many python/numpy operations by tensorflow operations as possible (e.g. computing $\max_{a'} Q(s')[a']$). Can you replace the two `sess.run` calls by a single one? How?

II.3) experiments

Let network-based Q-learner play against 'random', 'smart' and 'smarter' Player 2. Does it work better or worse than in part I)? Why do you think that is? Can you think of anything to improve the system (further)?

III) Connect-4 (due January 30th)

Write a network-based system that learn to play Connect-4 (`https://en.wikipedia.org/wiki/Connect_Four`) against a random opponent. Try to find a better architecture than the one in II. Maybe convolutional? What filter size?

Hand-in requirements

1. upload your code to I), II) and eventually III), as well as the results table from I.4.c) and the curves of cumulative reward from I.6.b). to the IST *git* server.