APML Snake Project

Rachel Ben-Hamozeg 300880143, Lirane Bitton 200024677

04/03/2019

An explanation of how you chose to represent the board state and why

The representation of the board state include both state and action. Actually we store the state of current state after performing a specisfic action ('F', 'L', 'R'). We tried to find a representation that would reflect the next steps and the near surrounding status. Our goal was to present the possible steps ahead to the learner while reducing as much as possible the board dimensionality. Moreover we wanted to somehow increase the impact of element near the snake (obstacle/fruits/other snake) so we included in our state representation a wider surrounding.

This was performed by cropping aroud the head a small part of the board and split it in four direction where we turn a bit on if this value appear in the wider surrounding and repeat it four time (it proves itself experimentaly). Since we have no prior knowledge about the board values except the fact that they will be in the range of [-1:9] hence every position or feature we wanted to suggest to the learner we represented it as a one-hot vector. This way we ensure that our solution is generic use less information that the board itself or other representation involving values.

Let assume that f is our feature vector or state representation hence $f_0 = 1$ iff the current position, after performing a step, is -1 and so on.

In the Linear model: the first position is reserved for the bias and then for each one of the 8 position around the head (not including the tail) are represented as a one-hot vector of length 11.

In the Custom model: we hold the same representation as the linear but also added mask of different regimes of the board and update in a vector which element exist there(0/1 again by index).

In both cases, all features are taken as from the point of view of the snake head. This "frees " us from taking care of duplicated state which are only affected by direction (each time we cropped the board after turning it and setting the snake in the middle)

An explanation of the custom model you are learning and the reasons behind your choice of model

We choose the model of a Neural Network. After a lot of trials more or less successful we were really surprised to see that the best trade-off we found between runtime/number of parameter was a "vanilla" network consisting of:

- Layer 1: dense(64) with relu activation - input
- Layer 2: dense(1) - output

The input to the model is the state-action representation defined above as tripplet for each direction and the output is the approxiamation value.

As said above we chosed this model since it was light-weight (~9000 param to learn) and asnwered to our time restriction. We choose to use as less as possible parameter such that no overfitting is performed, quick learning with bigger batch size can be perfromed an more important get a quick response time (combined to the state representation process).

An explanation of the learning algorithm you chose - which one did you use and why

We use the Q-learning ($\epsilon - gready$) algorithm with aproximation on the replayed data that was teached in class.

We used it because it enables us to estimate future rewards with the Q-function (with approxiamation) and establish the exploration/exploitation.

An explanation of how you chose to handle the exploration-exploitation trade-off

We defined an epsilon that is the probability of exploration (selecting action randomly). With the increasing learning - we reduce the epsilon by a constant rate to a minimum probability. These allows us to explore different areas on the board while the learner get more states to learn.

A short description of the test and results you got when you trained you policy

Testing our policies as a single policy: The results were very good, most of the time the policy got high scores. (linear varies between 0.25 to 0.55, custom between 0.35-0.65)

Testing against two or more avoid policies/ other policies we tried:

- The learner took more cycles to get good results
- Sometimes the final result was not sufficient on larger board (more than (100,100))

Testing with different parameters: We tried to test across a variety of different parameter (using a script to create multiple combination): internal: epsilon, epsilon rate, num of parameters in model, replay size, batch size external: board size, num of player Here tried to tune the parameters to achieve the best results/ find correlation between parameters. The results of these runs determined our parameters.

A detailing of other possible solutions that you tried and decided not to hand in and why(if there were any)

1. Use a cropped board (around the snakes head) and fed it to the the NN as is.

2. Use a cropped board with 11 channel (each for every char on board - each channel contains ones only where a certain type exists on board. This didn't work well and we tried to rotate the board according to the snake's direction.
   Both police didn't perform well.

3. A NN with the described representation that outputs three results (for each action) - also didn't give great results

4. A NN with convolutional layers then fully connected: worked well but less than the current one and for a lot more parameters and slower runtime.

5. We also tried to keep the representations as binary sparse matrices then we struggle with all the conversion back to numpy array and astype casting that we needed to make it run. We discovered that a NN cannot be fed with binary matrices(with dtype=bool in order to keep just a bit and not a byte for int8), this yield to uggly results and we involved an entire day to understand this issue. Then all the type casting back is runtime consuming for keeping more data. This trade-off was not advantageous under the restriction we have fot the exercise.

Any other things you would like to note about your implementation

Not exactly about the implementation but we performed some test on the cluster with a virtual environment we created and the proposed one while the only difference was the tensorflow version 1.10 with load module and the provided env while 1.12 installed in our and surprisingly the results are different in average. While in average our custom policy against "Avoid(epsilon=0.1);Avoid(epsilon=0.5);Avoid(epsilon=0);Avoid with default board size and plt and pat as defined in the exercise outperform

with getting a score of 0.4, in the created environement the average was 0.55. We tried to invest some time understanding what differences there is between those version while the network we have is only a 64 neuron dense layer to a single one output with relu. There was no explanation but still noting this.