APML Project - Snake

Vitaly Rajev - 320929227 | Ziv Mahluf - 316417385

March 5, 2020

Part 1 - Linear Agent

1. Board Representation:

We chose to represent the board by using a feature vector on a diamond-shaped window of a given radius around the position of the snake's head on the board, meaning all positions around the head within 'radius' number of non-diagonal moves.

In addition, we rotated the window around the head of the snake so the direction of movement whithin the window is always up.

We chose this form of representation since it allowed the agent, to learn weights for each value of each position, allowing it to approximate the q-values better, since it takes into account more possible states.

2. The Linear Policy:

The linear agent itself approximates 3 q-values at once for each state - corresponding to $Q(s,a)$ for each $a \in ActionSpace$ on a given state.

The agent approximates the values using a linear feature function of the state (as described in the board representation), and a matrix with 3 rows, each with a length depending on the radius chosen.

We chose this way in order to avoid having to calculate following states given a state and an action, and calculating the q-values multiple times per action or learning function.

3. Learning Algorithm:

The learning algorithm used for the linear agent is similar to the standard q-learning algorithm with a modification on the update step.

Since the agent doesn't manage a q-table and instead approximates the q-values using a linear feature function of the state, the weights vector is updated as follows:

The weights vector w in the row $j = argmax_i\{w \cdot \phi(s)\}$ is updated as $w_j = (1-\alpha)w_j + \alpha(r + \gamma \cdot max\{w \cdot \phi(s)\}) \cdot_{element-wise} \phi(s)$ (Where α is the learning rate, ϕ is the feature function of the state, γ is the discount factor, and r is the reward). Meaning that the row corresponding to the action which produced the most valuable step is updated in each round.

4. Exploration-Exploitation Tradeoff:

For this policy, we started with a certain value of ϵ, and a threshold δ below which we wanted to get by the last $score - scope$ rounds, starting from round $BATCH - THRESHOLD$ (before this round, ϵ does not decay) - we'll denote the number of rounds as r. We want that $\epsilon_0 \cdot \epsilon_{decay}^r \leq \delta$, therefore, we calculated $\epsilon_{decay} \leq \sqrt[r]{\frac{\delta}{\epsilon_0}}$ as the decay rate for the exploration rate of the agent.

This ensures that during the last rounds, which count towards the score, the agent will not explore with high probability.

Part 2 - Custom Agent

1. Board Representation:

We chose to represent the board in the same way as the linear agent, since like with the linear agent, we wanted the model to assign weights to each possible value in each position, with a consistent moving direction.

2. Custom Model Details:

The model we chose is a simple neural network, with one fully connected hidden layer, and 3 output neurons, whose purpose is to attempt to predict for a state as an input, the q-values for the state with any action ($model(s) = Q(s, a)$, for every $a \in ActionSpace$). This architecture was chosen for the simplicity of the network - allowing it to train faster compared to more complex models, and for efficiency - by avoiding adding the action as an input, we avoid having to run prediction multiple times per call for the acting and learning functions, instead predicting the q-values for every action at once.

3. Learning Algorithm:

The learning algorithm used for the custom agent consists of two neural networks with an identical architecture - a target network which is seen as an expert, and a training network, which constantly trains to mimic the expert.

Both networks are initialized randomly, and in each learning iteration, the target network predicts the q values of the resulting states possible from the current state (for each action). To this prediction, in the index of the action taken in the previous state, we add $(reward + \gamma \cdot max(training\_network(prev\_state)))$, to mimic the q-learning update step for a state and an action, and then we fit the training network with the previous state as an input, and the predictions resulting from the target network's prediction, with the addition to the corresponding action, as the label.

In addition, every few learning rounds, the target network's weights were updated to a weighted average of its current weights, and the learning network's current weights. This is done in order for the target network to improve (since the learning network is trained to predict according to the target network, and the update step of the q-learning algorithm) from what the training network learns, and gradually decrease the weight of the randomness in the initialization.

4. Testing the Custom Agent:

We tested the custom agent under the given conditions of the actual tests, as well as in environments alone and against one or two players, and for various numbers of rounds, in order to assess the performance and advancement of the model in a shorter time before testing it on a full game.

At first, when approaching with different solutions, the model got low scores, often finishing with a negative score even when alone on the board. After deciding to use a simpler model, with pre-processing of the states.

After deciding on the model, we started testing the effects of different hyper parameters on the results, including the number of neurons in the hidden layer, the decay rate of the exploration probability, the discount factor, the size of the window around the snake which we took into consideration, the batch size, and more.

In addition, in order to stay within the time constraints, we attempted to optimize the learning and acting proccesses, which allowed us to test bigger batches, and represent the board as we did (since the increase in the number of inputs slowed down the agent).

5. Additional Considered Solutions:

In addition to the chosen model, we also considered a convolutional model with multiple filters, followed by fully connected layers. We also considereddeeper fully connected networks.

We decided against these models since they were more complex and would require more training time than was given in order to achieve better results.

6. Exploration-Exploitation Tradeoff:

We dealt with the exploration-expoitation tradeoff by starting with a higher exploration rate, which allows the model to aquire many different experiences and increase the potential to stumble upon good rewards (perhaps eating fruits with positive rewards), and decayed it over time using a similar method to the linear agent, since the probability of randomly

stumbling on a better reward decreases as the model aquires more experience and learns to better predict the rewards for different actions. In later stages of the game, the model hardly ever attempts to explore different actions than those it learned, and when the game is at its score scope, the exploration is low enough that the model does almost no exploration, and only exploits the best reward it can in order to maximize its score ('playing it safe').

3. General

Test Results:
For the custom model, the test results (for 50000 rounds, with action time of 0.01, learning time of 0.05, and score scope of 5000) are:

Test Number	Agent	Avoid(0.5)	Avoid(0.3333)	Avoid(0.16667)	Avoid(0)
1					
2					
3					
4					
5					

For the linear model, the test results (for 5000 rounds, with action time of 0.005, learning time of 0.01, and score scope of 1000) are:

Test Number	Agent	Avoid(0.1)	Avoid(0.4)
1			
2			
3			
4			
5			