APML Project - Snake

Vitaly Rajev - 320929227 | Ziv Mahluf - 316417385

March 6, 2020

Part 1 - General

1. Board Representation:

We chose to represent the board using a feature vector of the area around the snake's head position within a certain radius r, such that each position in the area can be reached in up to r steps.

The area is in the shape of a diamond around the head, rotated such that the snake is always going up, and for each position in the area, there are 11 positions in the feature vector, indicating weather the correspondng particular value is at the corresponding position in the window.

We chose to use a feature vector since the values represent different discrete items on the board, and each value in each position should be considered differently, and not by the position alone.

The representation is extracted with a feature function of the state, which is used in both policies and will be denoted as $\phi(\cdot)$.

This representation was used for both the linear and the custom policy.

2. Replay:

In both policies, we keep a memory buffer from which we sample batches of examples to learn from. The buffer is represented by a queue with a fixed length, meaning in each learning step, the policies learn from multiple examples at once.

Each example in the buffer is a tuple $(prev\_state, prev\_action, reward, new\_state)$, in which the states are feature vectors.

Part 2 - Linear Policy

1. Policy Details:

The linear policy is represented by a weights matrix of 3 by the number of features (which is determined by the radius), which we'll denote as w.

The agent using this policy calculates, given a state s, $Q(s, a)$ for each $a \in ActionSpace$ according to $w \cdot \phi(s)$.

We chose to calculate the q values, which represent the expected reward achievable from s by doing the action a, for all actions at once, since if we decide to use the action as an input, we'd have to perform additional calculations for future states, as well as calculate the q-values multiple times each time, and these calculations might be wrong, or inefficient, and did worse when attempting to implement the policy to predict the q-value for a state and an action as an input, so we decided to let the agent learn these considerations implicitly in the matrix.

2. 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 $j = index(action(s))$ (the index of the action we took at state s, might have been random) 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 we took in that state is updated in each round according to the reward it produced, best future reward expected.

In each learning step, a batch is sampled from the memory buffer and for each example, an update step is performed as described.

3. Exploration-Exploitation Tradeoff:

For this policy, we decided to keep a low, consistent, exploration rate ϵ. Since the linear approximation is less accurate in many cases, we decided that we wanted the agent to explore consistently at a low probability, and constantly learn.

For the last rounds (the score scope), we zeroed ϵ in order for the agent to try and maximize its score by only exploiting what it learned up to this point. We attemppted starting with a larger value and using a decay, but the results were less successfull.

4. Test Results:

- Full test for 5000 rounds, with score scope of 1000, vs. 2 Avoid policies:

Test Number	Linear	Avoid(0)	Avoid(0.4)
1	0.0950	0.1770	0.2370
2	0.1420	0.1790	0.1090
3	0.2620	0.1100	0.1070
4	0.3690	0.1100	0.1610
5	0.2460	0.1850	0.1990

Part 3 - Custom Policy

1. Policy Details:

We decided to represent the custom policy using a neural network. We decided to use a fully connected network which receives the representation of the state as a feature vector, and predict the q-values for the given state for each action, similar to the linear agent.

We decided to use a single hidden layer in the network, since we wanted a model which would be relatively simple and quicker to train compared to deeper, more complex network architectures.

In addition, we tested multiple sizes for the hidden layer, and different activation functions.

Our final network architecture is:

$Input(num\_features) \rightarrow FullyConnected(size = 16, activation = ReLU) \rightarrow FullyConnected(size = 3, activation = Linear)$

We decided to predict all q-values at once for similar considerations to those of the linear model, with a larger emphasis on the efficiency in predicting and training.

2. Learning Algorithm:

The learning algorithm we used utilized a single neural network, such that in each learning step, we sample a batch to learn from from the memory buffer. From the batch, we construct the batch on which we will perform prediction (the previous states), and the batch of corresponding labels, which is constructed using the prediction vectors in which the value at the index of the action taken at that step is changed from the prediction to $reward + \gamma \cdot max\_future\_q$ (max_future_q is the best q-value according to the prediction on the next state).

The labels are calculated as such since for state s, action a, reward r, and next state s', the loss is $((r + \gamma \cdot max_{a'} Q(s', a')) - Q(s, a))^2$, which in the model training is MSE between the prediction ($Q(s, a)$) and the target we calculated.

After the construction of the batches, we call Keras' fit function for training the model.

3. Test Results:

- Full test for 50000 rounds, with score scope of 5000, vs. 4 Avoid policies:

Test Number	Agent	Avoid(0.5)	Avoid(0.3333)	Avoid(0.16667)	Avoid(0)
1	0.1234	0.1084	0.0916	-0.0276	-0.0672
2	0.1840	0.1308	0.0346	-0.0304	-0.0878
3	0.1772	0.0904	0.0462	-0.0198	-0.0982
4	0.2504	0.1230	0.0494	-0.0258	-0.0694
5	0.1238	0.1104	0.0558	-0.0488	-0.0754

5. Additional Considered Solutions:

Before considering a shallow network, we considered fully connected deep networks, and convolutional models which would receive a standard rectangular window over which a convolution would be applied. These models were dropped since they were more complex, and would require a far longer time to train, even with additional time and bigger batches, we would need more rounds of training.

In addition to the chosen model, we considered multiple shallow fully connected networks, with different numbers of neurons, and different activation functions. In order to choose which model we would use, we constructed multiple variations and rand them in different sessions of testing to determine which seemed to do better. Once we chose the architectures which seemed best, we ran multiple tests over each architecture to determine which one seemed the most consistent.

6. Exploration-Exploitation Tradeoff:

For this policy, we started with an initial ϵ and used an exponential decay every 500 rounds (~100 learning steps). This way, as the agent trained and learned to generalize, it explored less and less, since it needed to do so less and less.

At the score scope, ϵ is set to 0 in order for the model to 'play it safe' and try to maximize its score by only taking the best option which it predicts at each step.