APML Project - Snake

Omri Ben-Dov (204033971) & Franziska Weeber (888012168)

February 2020

1 Board Representation

Our goal in finding a board representation was to find a solution in which we only use the relevant close surroundings of the snake where it can move in a small number of rounds. However, we also want to learn a general direction the snake should move to from the representation. We tried two different board representations fulfilling these criteria. They are shown below in figures 1-3, while figure 2 and 3 represent two versions of the same approach. The fields taken into account depend both on the position and direction of the snake's head. We split that area into multiple regions, which are represented by different colors (a field split into two colors is counted for both regions). For each region, the count of occurrences of each board value $i \in [-1:9]$ is counted and used as part of our feature vector.

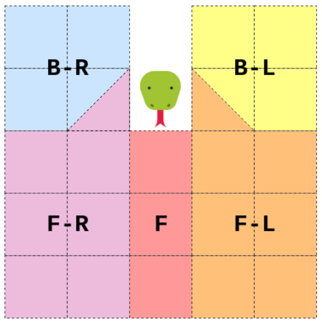

Figure 1: Board Representation with 5 Areas

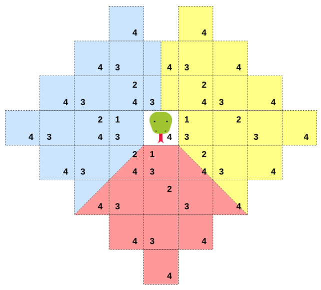

Figure 2: Board Representation with 4 Steps and 3 Areas

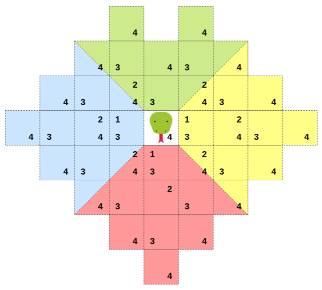

Figure 3: Board Representation with 4 Steps and 4 Areas

Our first and eventually preferred version uses a five by five quadratic field around the snake's head. Since the snake cannot move backwards, the position of the head is not centered but one field lower, giving more attention to the part of the board in front of it than the one in it's back. Our five regions for which board values are counted separately are left, forward-left, forward, forward-right and right. They differ in size, so the maximum possible count is three for the forward region and seven for the diagonal regions. Besides those regions, we also consider the three fields to which the snake can visit with one move separately. Since we have eleven possible values for the three adjacent fields as well as the five regions, the the feature vector includes $11 \cdot (3 + 5) = 88$ features for the board representation and a bias.

Although this representation gives us results meeting the evaluation criteria, we test another approach to the board representation. It focuses on the fields the snake's head can reach in a predefined number of steps. Two versions of this second approach are shown in figure 2 and 3, where the numbers in each field indicate by how many moves it can be reached given the current position and direction of the head. Then, for each combination of steps and region, the field values are counted as in the previous version. We tried four different combinations of three and four regions as well as three and four as the maximum number of moves with feature vectors between lengths of 99 and 176. However during testing, these second versions perform worse than the first one and are therefore discarded.

2 Our Custom Model: Policy Stochastic Gradient

2.1 Model and Learning Algorithm

For the custom policy, we chose to use policy stochastic gradient. For this model, the exploration-exploitation is done using a stochastic policy which gives each action a probability to be taken: $P[a|s] = \pi_\theta(a|s)$. We chose this model because we did not want another model that uses Q-learning, yet we still wanted a deep-learning algorithm. The policy gradient method seemed simple and elegant.

In order to learn the probabilities for each action at each state, we used a simple neural network. The network's input is the state vector, which goes through 2 fully-connected layers with 32 nodes each and a ReLU activation, and ends with fully-connected of 3 nodes with softmax activation in order to get a probability for each action.

According to the policy gradient theorem:

$$\nabla_\theta R_\theta = R(\bar{s}) \sum_{t=1}^{T} \nabla_\theta \log\left(\pi_\theta\left(a_t|s_t\right)\right)$$

The gradient of the log of each action in this formula is achieved by minimizing the cross-entropy loss of the network (Minimization is done by the ADAM optimizer. We also tried SGD, but ADAM worked best). The summation is done by using batches, because the gradient is calculated using the mean gradient of the batch (which is the summation divided by a constant number). Finally, the multiplication of the reward is done by weighing each episode by its reward. That way, when the gradient is computed, it will be multiplied by its weight (its reward).

2.2 Evaluation and Results

We chose the hyper-parameters of the model using cross-validation. We played games with the same model and different parameters. As a result we set the network with 2 hidden layers with 32 nodes each. We also set the discount factor to $\gamma = 0.5$ and the learning rate to 0.001.

Finally, we evaluated our model using the description in the assignment: We put it against 4 avoiding policies with different ϵ's (0, 0.1, 0.2, 0.5) for 50,000 rounds with a score scope of 5,000 with an action time of 0.01 seconds and learning time of 0.05 seconds.

These are the results:

Policy	Game #1	Game #2	Game #3	Game #4	Game #5	Average
Policy Stochastic Gradient	**0.3818**	**0.4018**	**0.4188**	**0.357**	**0.4434**	**0.40056**
Avoid($\epsilon = 0$)	0.0548	0.066	0.068	0.0672	0.0508	0.06136
Avoid($\epsilon = 0.1$)	0	0.0428	0.0124	0.0226	0.001	0.01576
Avoid($\epsilon = 0.2$)	-0.0294	-0.022	0.0036	0.0066	-0.0206	-0.01236
Avoid($\epsilon = 0.5$)	-0.1376	-0.1326	-0.148	-0.1496	-0.142	-0.14196

The best score for each game is in bold. Our Method outperforms all avoiding policies by an order of magnitude consistently.

3 Discarded Solutions

3.1 Quadratic Q-Value Approximation

For the linear Q-value approximation, we try to learn some weights matrix θ s.t. given a state vector ϕ we can approximate $Q(s,a) = \left(\theta^T \phi\right)_a$. So the a'th element of the resulting vector is the Q-value. A trivial way to build on top of this idea would be to try a quadratic approximation.

In this case we try to learn a weights matrix Θ s.t. $Q(s,a) = \psi^T \Theta \psi$. Where, in this case, ψ is the state the snake will end up in after taking the action a. We approximate the new state ψ by moving only the snake and capturing its state representation.

One difference between the linear and quadratic cases are the sizes of the weights matrix ($3 \times d$ for the linear case, $d \times d$ for the quadratic case. d is the number of dimensions of the state representation). While the quadratic weights matrix is $d \times d$, since symmetric elements relate to the same elements in the result, we can use it as a triangular matrix and thus reduce the number of variables by a factor of almost 2.

Another difference is that in the linear case we use the current state and produce a vector with 3 Q-values, while in the linear case we approximate the next state for each action and produce only its Q-value.

We tried this method against the given avoiding policy. While the quadratic approximation learned fast and received a positive score of 0.3 during the first few hundreds rounds, it quickly began to lose points and got to the negative score range. We tried lowering the learning rate, but success was yet to be achieved.

If we would invest more time in the optimization of this idea, we could maybe get good results, but ultimately we decided on discarding it in favor of the policy gradient.

3.2 Genetic Algorithm

Another fun idea we tried (which surprisingly worked) was using a genetic algorithm. We based it on the already written Q-value approximation from the linear case. Meaning, instead of learning the weights vector θ using Q-learning we learned θ using evolution.

We start by creating random weight vectors (drawn from a normal distribution) and giving each of them a fitness score by using them sequentially, each for 100 rounds. So instead of playing multiple snakes, we used one snake with a dissociative identity disorder. We then randomly choose pairs of parents such that the higher the score, the higher the probability to be chosen. For each of these pairs we produce a new child (weight vector) by setting each element to either the corresponding element in either parent, chosen with a 50-50 chance. For each element in each new child, we mutate it with a 5% chance by adding it a normal random number. We then replace the old generation, with the new generation.

This process can repeat indefinitely. But, when the game is reaching its end (when we enter the score scope), we choose the child with the best score and play him until the end. The algorithm is also shown on the next page.

As already said, this algorithm eventually learned to win and even beat the avoiding policy with $\epsilon = 0$. The algorithm has some disadvantages:

1. This algorithm is that it still relies on the linear approximation.
2. It takes a lot of times to train and create children with positive score.
3. The results are inconsistent.

We could try changing the number of rounds each child is playing, or the number of children in order to get better performance. But at the end, our policy gradient is more consistent and learns faster, so we discarded this idea as well.

Algorithm 1: Q-Values Genetic Algorithm

```
/* Initialization                                                          */
Create 10 i.i.d. random vectors using normal distribution (children) ;
while not in score scope do
    /* Get fitness score of each child                                     */
    for each child do
        Use child for 100 rounds and store its score (no exploration) ;
    end
    /* Selection                                                           */
    Produce probability distribution using given scores ($p = \frac{e^{S_i}}{\sum_i e^{S_i}}$) ;
    Choose 10 pairs of parents using new probabilities (higher score has better chance to be chosen) ;
    /* Crossover                                                           */
    for each pair of parents do
        Create new child by taking each element from a random parent (50% each);
        /* Mutation                                                        */
        With a 5% chance, for each element add a normally distributed random number;
    end
    Replace the old generation with the new children;
end
Choose child with best score, and keep playing it until end of game;
```