Snake on a Plane

Department of Mathematics and Computer Science, Davidson College, Davidson, NC 28035

Overview

We design and implement an intelligent agent that learns to play the classic video game *Snake*. The agent uses the Q-learning algorithm and a set of well-defined features describing the game to learn to maximize its score. After many repeated attempts at the game, the agent refines a policy that enables it to play *Snake* at an expert level.

Q-Learning

The Q-learning algorithm allows an agent to learn an optimal policy from repeatedly interacting with its environment. Each combination of state and action is mapped to a real value, called a Q-value:

$$Q : S \times A \to \mathbb{R}$$

The Q-values are updated whenever an action is taken. The Q-learning algorithm in environments with discrete state spaces is:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

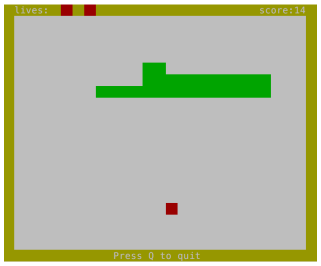
In environments with sufficiently large or continuous state spaces, the algorithm depends upon a number of well-defined features. Features quantify aspects of the problem space in order to provide additional information to the agent. The Q-values are then defined as linear combinations of the values of features. The weights assigned to each feature are calculated using an adapted version of the Q-learning algorithm:

$$weights[i] = learning\ rate * (reward + discount * \max_a Q - Q(s_t, a_t)) * s_a$$

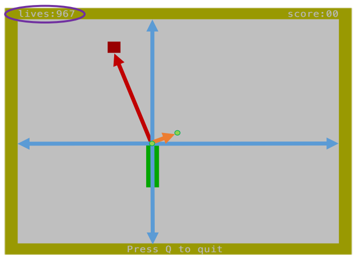
The action chosen at each state is then the action that yields the maximum possible Q-value.

Features

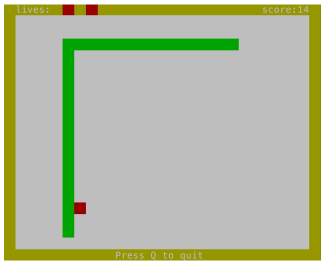
- **Constant** – allow flexibility of linear combinations.
- **Distance from Walls** – measured from head of snake.
- **Lives** – current number of lives before game over.
- **Distance from Center** – measured from head of snake.
- **Snake Length** – units in the snake's body.
- **Distance from Apples** – measured from head of snake.
- **Compactness of Snake Body** – quantifies how coiled or spread out the body of the snake it.



Compact snake.



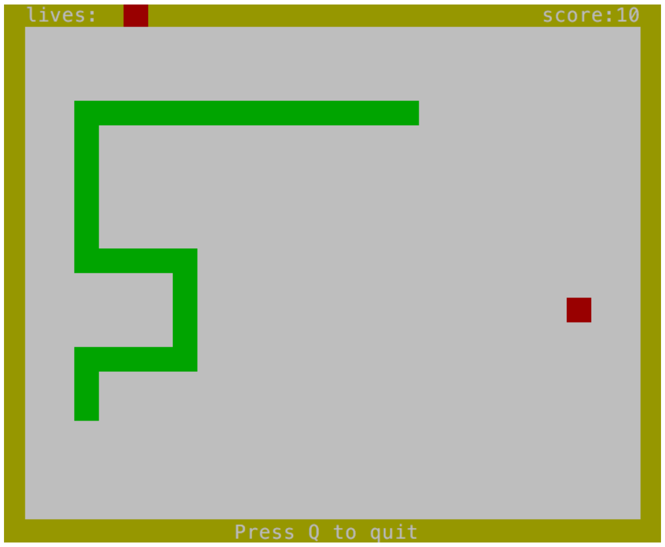
Example *Snake* illustrating features.



Non-compact snake.

Snake

The goal of *Snake* is to maneuver a snake character towards 'apples' while avoiding colliding with obstacles. The 'apples' increase the score and length of the snake. Collision with the boundaries or the snake results in GAME OVER.



Example *Snake* game. Boundaries are yellow, apple is red, and snake is green. Implemented in Python, runs in OSX terminal.

Results

States

Since this poster went to press we defined the available states as follows:

$\{w_s, w_l, w_r, q_f, q_t\}$

Where the w's indicate whether a wall is adjacent to the head in the straight, left and right positions and the q's indicate the distance from the head of the snake to the apple and the tail (Ma, Tang, Zhang 2).

Actions

The action space is defined as turning the head straight, left, or right.

Reward Function

The reward function is as follows:

Condition	Reward
Eat Apple	+500
Death	-100
Else/Default	-10

Policy

Features	Weights
Constant	-412.872
Snake Length	0.576
Distance from Center	61.360
Wall Adjacent (Straight)	-0.262
Wall Adjacent(Left)	-0.235
Wall Adjacent(Right)	0.553
Distance from Apples	152.977
Distance from Tail	-1.251
Compactness of Snake	66.258

After training our agent for 10,000 lives using an epsilon greedy exploration the agent generated the above policy. We can see that it placed the highest weight on the agent's distance from apples, the distance from the center and compactness.

Conclusion

The policy generated from the algorithm does not enable the agent to play Snake successfully. The reasons for this could be inadequate lives, inadequate time given to converge for a policy, bug in the implementation of the algorithm, and/or poor choices of state definitions and features.

To continue this work, we will test out hard coded weights to see if our features are adequate. If we manage to find a combination of weights that successfully teaches our agent to play the game, then the implementation of the algorithm is correct as is. Otherwise we intend to intensively debug the algorithm.

We will also consider giving the agent more time to converge on an optimal policy and testing non-linear combinations of features to approximate our Q-function.