Balalayka - Agent1

Dmitri Koltsov and Chris Daniels

Description

Agent1 is a pair of agents competing against other agents in a game of Pacman. To beat the enemy team an artificial intelligence method will be used to train and test Agent1. Agent1 will consistently win games as either the red or blue team on various maze layouts.

In order to beat opposing agents, the entirety of the `gameState` will be taken into account. Agent1 will actively avoid enemies on the enemy side, while seeking the closest paths to food. Friendly agents and scared states will also be considered.

To make Agent1 consider these possibilities, a reinforced learning neural network will be used to train and test Agent1. Features from the `gameState` will be features used to calculate the best legal moves at any state. Reinforced learning is highly used for training a model while data streams. Since this pacman has options for running multiple games and training a model, reinforced learning was chosen.

Requirements

Since we are using reinforced learning to calculate the next best move for Agent1, various actions will be considered positive or negative. The Keras_RL library will be used to generate the predictions.
The most important actions are:

1. Shape reward function for moving towards food
2. Shape reward function for avoiding ghosts
3. Shape reward function for other game mechanics (Returning food, power capsules, defending food, etc)

Rewards are given for:

- Every food consumed
- Every food returned

Penalties are given for:

- Every time our agent is eaten by an opponent
- Every time food is dropped

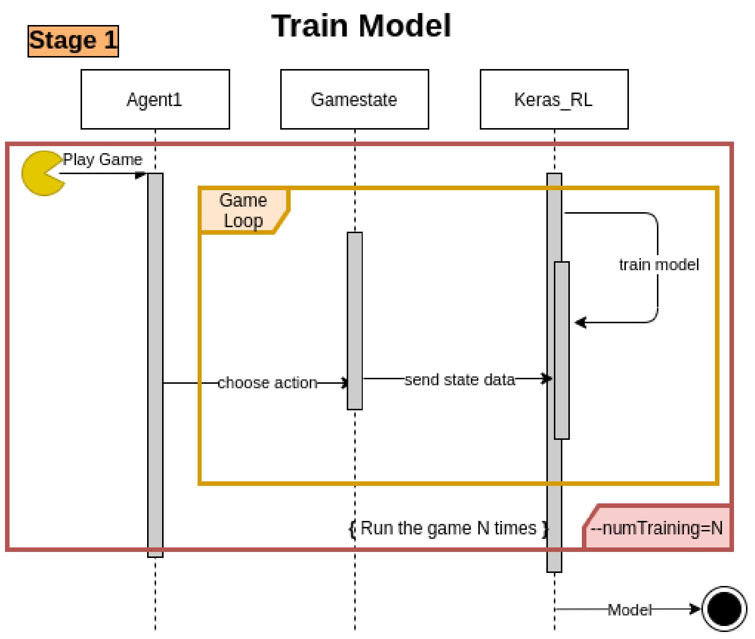
Modules

The agent will go through stages of learning what actions work well and what actions don't work well. The first stage will involve allowing the agent to randomly attempt moves until the rewards are maximized.

Stage 1: The agent will take legal actions that accrue positive rewards and avoid receiving penalties. The agent will play the game itself choosing random actions. The Pacman game has a command line argument that runs the game `N` times. The command is `$ python capture.py --numTraining=N` and can be used with any agent.

During this stage the game will be run for a tentative time of 5 to 8 hours. Since each game takes around 10 seconds to finish, approximately 360 can be played per hour. The amount of games that will be played will vary from 1,800 to 2,880.

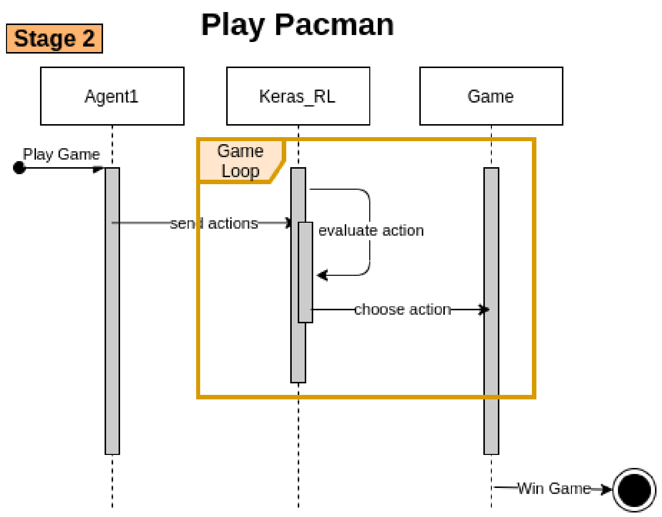
Below is the sequence diagram for how the model training will move. The game will be played by Agent1, which will choose an action. Each action has a state that is used to infer what penalties or rewards will be given. After this stage, the testing of the agent will begin.



The neural network will have one hidden layer with 1000 neurons. The output layer will have 5 nodes for each action. The input layer will take in each `gameState` feature. The hidden layer is going to have the sigmoid or hyperbolic tangent activation function. For the outer layer, the softmax function is used.

Stage 2: The game will be played with the resulting model from stage 1. When the game is active, we will observe how Agent1 plays the game. These observations will be recorded for later consideration.

Agent1 should move around the maze well enough to consume food and return it to the friendly side. Opponents should be avoided during these sessions as well. The observations on how Agent1 chooses actions are going to be used for stage 3.



Depending on how well Agent1 plays these games will determine how we shape rewards and penalties. Stage 2 is meant for testing the model. During this stage, the agent will still be training, but will have successfully run for several hours. Stage 2 is a checkpoint on how Agent1 is doing.

Stage 3: If Agent1 is not doing as well in various states, we will revise the reward structure. This stage will be used to review how rewards are being seeked, and how penalties affect the decisions. Agent1 may try to minimize the penalties, or maximize the rewards. We want Agent1 to have a healthy balance of avoiding penalties and gaining rewards.

The reward structure will be adjusted according to the observations from stage 2. If adjustments are made, then Agent1 will return to stage 1 for training. The adjustments should allow Agent1 to make decisions that win games a majority of the time.

Milestones

August 3rd:

1. At least one training session of 5 to 8 hours using the Keras_RL library.
2. Agent1 should move around the maze without getting stuck.
3. Agent1 should win 50% of the games played.

August 17th:

1. At least one more training session after the initial training session(s).
2. Agent1 should win 80% of the games played.
3. Agent1 should avoid opponents on the enemy side.

Detailed Design

To feed in data for the input layer of the reinforced learning neural network, the various features of the `gameState` will be used. We are going to use a 1-D array of size `field_width` * `field_height`. Each element will represent one cell in the maze.

To associate the actions to the state, we are going to collect data from the area surrounding our Agent1. This area is square-shaped, so we will call it a square. The square has a radius of 5 tiles. So we have: `n = radius * 2 + 1` and the square has size `n x n` with Agent1 in the center.

We have 9 positional groups of features, and for each group we are going to use a 1-D array of size `n^2`. Each element will cover the cells inside the square surrounding Agent1 (value = 1 if feature is present in the cell, value = 0 if it's not). The features are as follows:

1. If the cell is inside the maze grid. (in the case Agent1 is close to the border of the maze, the square radius may by partially located off the maze grid)
2. If the cell has a wall.
3. If the cell has food for our agents.
4. If the cell has food for the opponents.
5. If the cell has a power capsule for our agents.
6. If the cell has a power capsule for the opponents.
7. If the cell is the location of the friendly agent.
8. If the cell is the location of the 1st opponent.
9. If the cell is the location of the 2nd opponent.

As a result we are going to have vector of 0's and 1's with size of `9 * n^2`.

In addition to positional features, we have 11 qualitative features:

1. Scary timer reading for the 1st friendly agent.
2. Scary timer reading for the 2nd friendly agent.
3. Scary timer reading for the 1st opponent
4. Scary timer reading for the 2nd opponent
5. Amount of food inside the friendly agents
6. Relative horizontal position of the friendly agents, calculated by formula
    a. `(x - half_grid_width) / grid_width`
7. Relative vertical position of the 'agent', calculating by formula

    a. `(y - half_grid_height) / grid_height` (x and y coordinates of the 'agent')
8. Relative horizontal position of the friendly agent, calculating by formula
    a. `(x - half_grid_width) / grid_width`
9. Relative vertical position of the friendly agent, calculating by formula
    a. `(y - half_grid_height) / grid_height` (x and y coordinates of the friendly agent)
10. Horizontal coordinate of the 'agent'
11. Vertical coordinate of the 'agent'

As a result, we have 1098 features, plus the game field size features fed into the input layer.

References

- https://github.com/keras-rl/keras-rl
- https://keras.io/
- Reward Shaping - https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0
- Keras Tutorial - https://hub.packtpub.com/build-reinforcement-learning-agent-in-keras-tutorial/
- Diagrams - https://draw.io