CISC889-010 2019S
Project Writeup
Ruiqi Wang
Hang Chen

Multiagent Reinforcement Learning in Pacman

Introduction

The classic Pacman game is an ideal environment to simulate concise multiagent learning actions because there exist two adversaries - Pacman and ghosts and they can be scaled into multiple Pacman agents and ghosts to either cooperate or compete within or between each group. In our project, we have tried out implementing SARSA and Deep Q Network on our Pacman agents to do homogeneous cooperative learning. In other words, we have put both SARSA and DQN Pacman agents together along with the ghosts inside of the same game environment to do reinforcement learning to get the maximum possible rewards(score). The way of realizing the process is by extracting several complex features from the environment to provide specific information about the current state to our Pacman agents, and then our agents will take actions according to the information.

Framework

We adopted the single Pacman game simulator from UC Berkeley CS188(https://inst.eecs.berkeley.edu/~cs188/fa18/projects.html). The framework was designed to teach fundamental AI concepts, such as informed state-space search, probabilistic inference, and reinforcement learning, by applying an array of AI techniques to playing Pacman. It offers a classic Pacman game environment in which it provides the mechanisms of moving a single Pacman agent to gain positive rewards by eating dots, eating pallets followed by eating ghosts, or winning the game, and gain negative rewards (lose score) by moving the Pacman after a time step, being eaten by the ghosts, or losing the game. The number of ghosts is determined by the layout files stored in the layouts folder in the game

directory. The framework is written in Python3 and we have added numpy and PyTorch libraries to train our DQN agent. The PyTorch library supports quick matrix operations if the machine used to run this project has NVIDIA graphics card equipped.

Problem Definition

Challenges we faced

- Scale a single pacman agent into multi-pacman agents
- Adjust the layout based on specified pacman amounts
- Properly show graphics updates during training for demonstration purpose since one game iteration shown as in graphics may take a long time
- Extend learning algorithm from single agent to multiagent
- Extract proper features to provide as much information as possible to agents

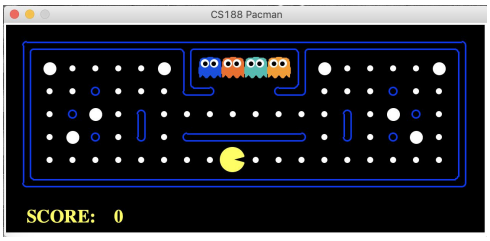
Structural Changes to the Framework

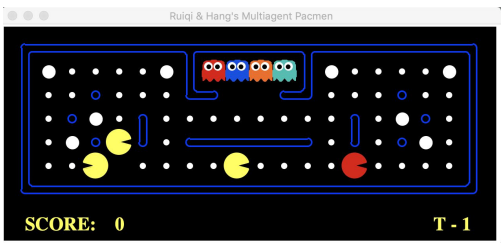
Original framework

- More specifically, the framework provided in CS188 project3 (https://inst.eecs.berkeley.edu/~cs188/fa18/assets/files/reinforcement.zip) is the one we built our project upon. It was originally crafted for single Pacman reinforcement learning. This codebase brings a function called “loadAgent” in pacman.py that scans through all the python files ending with “-gents.py” in the project directory to load that type of Pacman agent. For instance, if we write an ApproximateQAgent.py which contains an Approximate Q Pacman agent class, the program would be able to create this type of agent in the game if we specify its type at the beginning of the game. That enables us to create Pacman that uses different types of reinforcement learning algorithms.
- While creating the agents, Pacman or ghost, the original framework uses a list to store all the agents and load them one by one to perform actions while the game is running. This makes it possible to add more than one Pacman agent to the agents' list and load all of them before the ghosts are loaded.

Our modified multi-pacman framework based on Problem Definition

- Scaled from single to multiple Pacmen with multiple types
    - The original framework serves a `-p` or `--pacman` argument used to specify the agent type(or to say, the reinforcement learning algorithm because one certain type of Pacman agent uses a specific reinforcement learning algorithm), e.g. `-p ApproximateQAgent` will create a Pacman agent that uses Approximate Q learning algorithm in the game.
    - In our modified framework, we have changed the behavior of `-p` from accepting a string containing one Pacman type, to a string that contains multiple available Pacman type. That is, `-p "ApproximateSarsaAgent, PacmanDQAgent"` will introduce a Sarsa agent and a DQN Agent simultaneously into the game, and as it's shown, the different types of Pacman need to be separated with a comma. In addition, we have also included an option to specify an amount for the corresponding Pacman type, and that's by supplying numbers to `--pacmanAmounts`. Suppose we have `-p "ApproximateSarsaAgent, PacmanDQAgent"`, then, `--pacmanAmounts "1,3"` will introduce 1 Sarsa agent and 3 DQN agents to the game correspondingly.
    - We have also hardcoded different color to draw different types of Pacman to better visualize their performance. We implemented this by replicating how the original framework draws different colors for different ghosts. The color corresponds to which agent will be told when the game begins if graphics is turned on for the game.
    - Virtually, we have changed the framework from



to


And the initial console output will address which color is assigned to which type of Pacman agent

```
Game 1 displays.
Beginning 2 episodes of Training on ApproximateSarsaAgent index 0 with color Yellow
Beginning 2 episodes of Training on ApproximateSarsaAgent index 1 with color Yellow
Beginning 2 episodes of Training on ApproximateSarsaAgent index 2 with color Yellow
Beginning 2 episodes of Training on PacmanDQAgent index 3 with color Red
```

  - Furthermore, to make the learning procedures more meaningful, we have made the Pacman agents store their own score respectively. The original framework stores the game score inside of the variable `state.data.scoreChange`, which is ideal for a single Pacman case. However, during the evaluation of our post-learning agents, some DQN agents performed badly, and thus we realized the problem should be associated with the score update. In other words, if we kept `state.data.scoreChange` and let every Pacman refer to this score as their own score, one Pacman die would deduct 500 from the game score, and this may confuse another Pacman who's eating a dot with thinking this dot is "poisonous". Thus, we created a `self.score` attribute for the Pacman agent and let the agent choose actions based on their own corresponding score.
  - We've set the rule to end the game only if all Pacmen die. The dead Pacman will be removed from the screen immediately and skipped being loaded the next round.
- Layout will be customized to fit the Pacman amounts while being loaded
  - The original framework provides a `-l` argument to specify a layout for the game to use, which is stored in the layouts folder.
  - However, all the layout files provided by the framework only supports a single Pacman. We have modified it to fit the total specified Pacman amounts by randomly replacing some food dots and/or spaces with available Pacman positions.
  - The newly generated layout file will be stored inside of layouts/Multi_Pacmen_Layouts/ folder, named as {originalLayoutPlusDateAndTime}, and then loaded to the game. For example, if we specified more than one Pacman for powerClassic layout by `-l powerClassic --pacmanAmounts "1,3"` on May

30th at 16:36:39, a layout file called `powerClassic_20190530-163639` will be created and it supports 4 Pacman position. We add a timestamp to the new layout to fulfill random positions each time we start a new game with the same layout, and if we still want to use an older generated layout, we can reuse it by copying it to the layout/ folder and specify its name.

    - We do limit the available Pacman positions for a certain layout to be less than or equal to the third of the entire amount of the dots and spaces in the original specified layout. The game will return an error message if excrescent Pacmen are specified.

- Game can be run in either text mode or graphics mode in both training and evaluation phases
    - The framework provided in CS188 Project 3 totally removes the option to run the game for reinforcement learning agents in graphics mode. One possible reason might be displaying the game with agents performing learning would dramatically slow down the training procedure.
    - However, we think it is beneficial to have graphics available for some game runs just to virtually check how our agents are learning in action, especially when we are doing multi-agents learning, since it can help to debug and it also helps demonstrate the performance of our post-learning agents while in the evaluation process.
    - The original framework supplies `-x` and `-n` arguments to indicate the number of training rounds and total runs. For instance, if we run the game by `-x 20 -n 32`, all the established Pacman agents will be trained for 20 rounds and evaluated for their post-learning performance for 32-20=12 rounds. Depending on these two arguments, we have added three additional arguments `--trainingGraphics`, `--trainingGraphicsUpdateFrequency` and `--evalGraphics` to specify graphics display for some certain game runs.
    - While the `--trainingGraphics` argument is provided solely, the game will be displayed every round during training. While the

`--trainingGraphicsUpdateFrequency` argument is specified with a number along with `--trainingGraphics`, say 4, then the game will be displayed at every 4th round. For instance, `-x 20 --trainingGraphics --trainingGraphicsUpdateFrequency 4` will make the game Round 1, 5, 10, 15, and 20 to be displayed. Game 1 will always be displayed and this information will also be shown in the initial console output. If `--trainingGraphicsUpdateFrequency` is specified by itself without `--trainingGraphics`, the game will not be displayed during training.

- While `--evalGraphics` is specified, the game will be displayed every round during evaluation.
- We have added a helper sign to the graphics to show whether this is a training or evaluation round with its round number indicated.

- training round 1

- evaluation round 1

Learning Methods

Feature Extractor

We extract 9 complex features from the game state, including distance to food or ghost, the number of ghosts, the direction of ghosts, etc. To prevent the parameters in the deep neural network from being too large, we perform normalization by dividing the distance by the size of the whole maze, which can be considered as the longest distance in the maze, so the value of the distance is between 0 and 1.
Here is the list of features we extract:

1. #-of-ghosts-2-step-away: number of ghosts with distance 2 away from agent
2. #-of-ghosts-5-step-away: number of ghosts with distance 5 away from agent
3. closest-ghost: the distance to the nearest ghost

4. closest-ghost-direction: the direction of the nearest ghost corresponds to agent
5. closest-ghost-status: if the nearest ghost is scared, which means it is edible
6. closest-food: the distance to the nearest food
7. closest-food-direction: the direction of the nearest food corresponds to agent
8. closest-capsule: the distance to the nearest capsule
9. closest-capsule-direction: the direction of the nearest capsule correspond to agent

In these features, the first three will tell agents how urgent it is to escape from being caught, and the direction information will help agents to decide which way to run. However, if the nearest ghost is edible, the action taken by agents will be definitely different. After eating scared ghosts, agents will get 200 points rewards. So this feature will help agents to decide whether to run or chase. The remaining features are about the direction and distance of food and capsule which will give pacmans some rewards and eating all food dots is the only way for pacmans to win the game. Therefore, the last 4 features will lead well-trained pacmans to victory.

SARSA

SARSA is a typical temporal difference (TD) learning algorithm, which enables the agent to learn through every single action it takes. It is acronym for 'State - Action - Reward - State - Action'. An agent will choose one action base on state and receive some rewards, then it will be in a new state. From the new state, the agent will choose another action. So each step, the agent will update its Q values based on the tuple of (S, A, R, S', A'), as shown in Figure 1.1

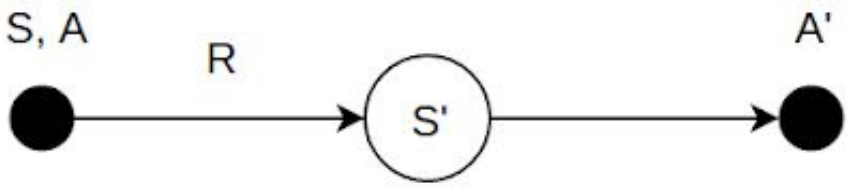


Figure 1.1 Illustration of SARSA

TD method updates the knowledge of the agent on every timestep rather than reaching the terminate state. And there are two kinds of different TD method,

on-policy and off-policy. On-policy means agents will choose action and update Q values with the same policy, which is the one they are learning. However, agents using off-policy learning algorithms will choose actions with a different policy, and Q learning algorithm is one of the most famous off-policy learning algorithms. Base on Q table, SARSA will update Q values according to the following Bellman Equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[reward + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Where the part $reward + \gamma \max_{a'} Q(s',a')$ is defined as the target. From this equation, it is easy to conclude that SARSA is an off-policy algorithm.

In this project, we adopt two ways to perform SARSA, the first one is to use Q table to store Q values for each state, like if there are 9 features as we have, the Q table should have 9 dimensionalities. This is simple and easy to implement, however, when the number of features increases, there will be a curse of dimensionality. So the second method is to use a function approximator to 'predict' the Q value given the state features, and instead of updating Q values directly, we update parameters in the approximator. In this way, similar states will have similar features which will result in similar output from function approximators. So it is both reasonable and effective. The algorithm used to update parameters is stochastic gradient descent and loss is defined by the difference of approximator's output and the expected discounted reward named 'target'.

Deep Q Learning

Benefit from the representation power of the deep neural network, we can replace Q tables or function approximators with the deep neural network, and this is the main idea of Deep Q Network(DQN)[3]. As the name suggests, Deep Q Learning is a way to combine deep learning with Q learning, and deep learning will take features as the input of neural network and output the optimal action in the current state.

With features we extract from the game environment, we implement a neural network for Deep Q Learning, and the network structure is shown in Figure 1.2.

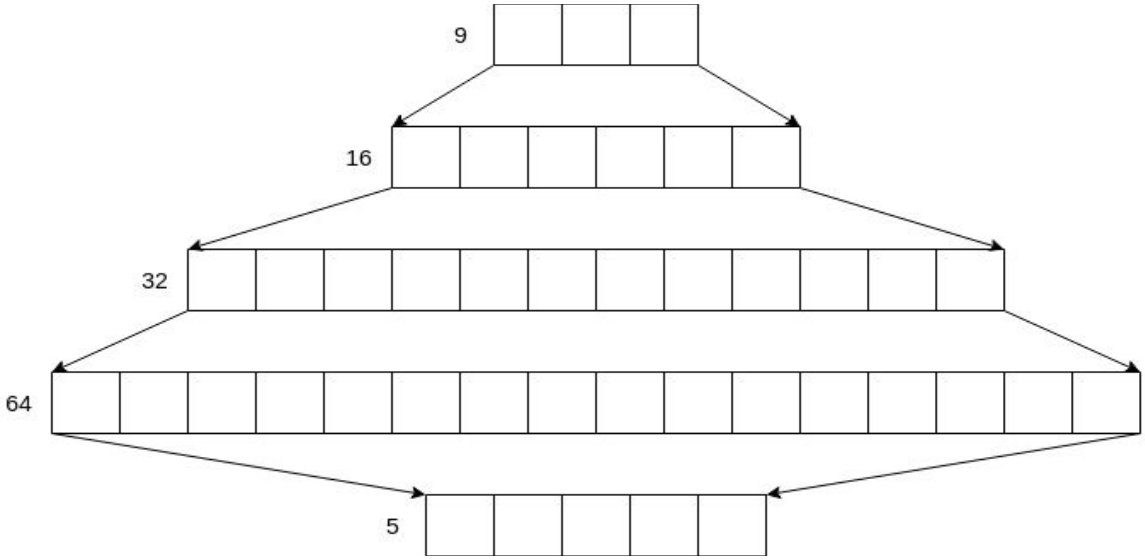


Figure 1.2 The structure of our neural network model, each of the outputs indicate the Q value of corresponding actions, including North, South, West, East and Stop

Result

SARSA

Using Q tables will take SARSA lots of time to converge, since each state contains the position of ghosts and pacman, number and location of food dots. Even with the same location of ghosts and pacman, only one missing food dots is a completely new state for SARSA. So the algorithm with Q tables only gets 34% winning rate after 1,000 iterations of training.

```
Win Rate:      17/50 (0.34)
Record:        Loss, Win, Win, Loss, Loss, Loss, Win, Loss, Loss, Loss, Win, Loss, Win, Loss, Loss, Loss,
 Loss, Loss, Loss, Win, Win, Win, Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Loss, Loss, W
in, Loss, Win, Win, Win, Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Win, Loss, Loss
```

Figure 2.1 win rate of SARSA agent using Q tables after 100 training iterations

Using the approximation function, SARSA can learn much faster than using Q tables. The result shows that after 1,000 training iterations, the winning rate is 56% out of 50 evaluations.

```
Win Rate:      28/50 (0.56)
Record:        Loss, Loss, Win, Win, Loss, Loss, Loss, Win, Win, Loss, Loss, Win, Loss, Loss, Loss, Loss,
 Win, Win, Win, Loss, Win, Win, Win, Loss, Loss, Loss, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win,
 Loss, Win, Win, Loss, Win, Loss, Win, Win, Loss, Loss, Win, Win, Win, Loss
```

Figure 2.2 win rate of SARSA agent using approximation function after 100 training iterations

Deep Q Network

We extract 9 features from the given game state (including pacman's position, ghost's position, food dots' position, etc.) and trained a fully connected neural network to learn the Q value. The network we use has 4 layers, and it uses mean squared error as loss criterion, Adam as the optimizer. The loss curve of one of the agents is shown in Figure 2.3.

However, DQN doesn't work so well with manually extracted features, since we may lose some useful information and some features can be imprecise, like the distance, we only consider the Manhattan distance from pacman to ghost, but we ignore the block of wall. Similarly, when features show a ghost is on the left, that ghost might not come from the west because of the wall, but south or north. This imprecise information causes some problem with convergence. We think this is the main reason why the loss curve is not monotone decreasing in general.
One solution to the imperfect features is to use a convolutional neural network (CNN) to extract features from pixels. That works for a single agent, but for multi-agent CNN is not capable of differentiating agents from each other and output the Q values corresponding to each one.

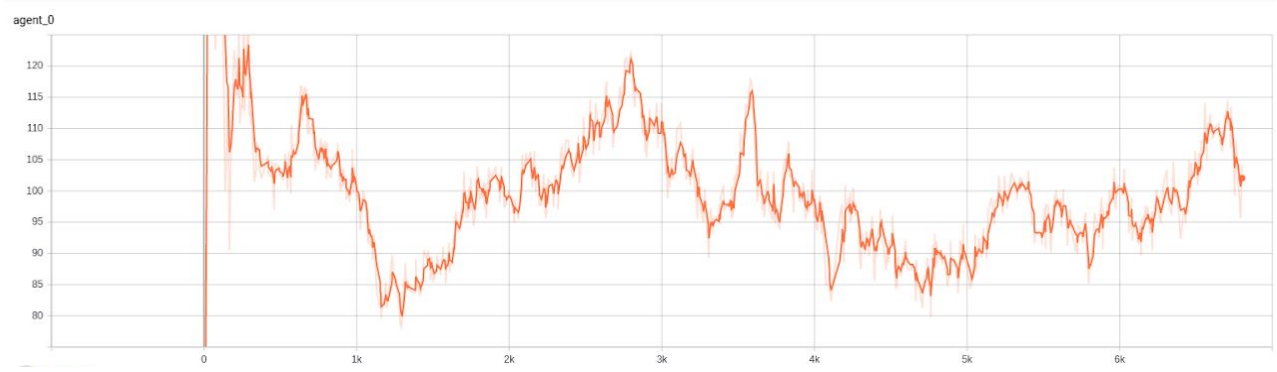


Figure 2.3 loss curve of one agent

Future Work

In our project, Pacman agents are not aware of the existence of each other, so there is no explicit cooperation among agents. We can add more features such as the distance to the nearest agent, to let pacmans know that there are some other homogeneous agents doing the same thing and, in this game settings, they can break the tie by choosing a different direction and taking care about food dots in different regions. Besides, different agents can share parameters to tell others "what they have learned", this can also speed up the learning process since it is like a kind of transfer learning. As for rewards, there are two approaches to assign rewards to agents, the first one is that all agents share the same reward at each time step. That means when one pacman dies all other agents will be punished. The other one is that each agent has its own reward and will not be neither encouraged by nor blame for other agents' actions. Although we are using the second one, the first one is also reasonable because all pacman agents work as a team. When any one of them was eaten, it will slow down the process of eating all food dots.
We will also try implementing the other reinforcement learning algorithm as well since the framework is scale-friendly.

Reference

[1] UC Berkeley Pacman Project:
https://inst.eecs.berkeley.edu/~cs188/fa18/projects.html
[2] Tziortziotis, N., Tziortziotis, K. and Blekas, K., 2014, May. Play Ms. Pac-Man using an advanced reinforcement learning agent. In *Hellenic Conference on Artificial Intelligence* (pp. 71-83). Springer, Cham.
[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. Nature, 518(7540), p.529.

Supplementary Material

Evaluate Our Multi-Pacman Program

- Environment Setup
  - Please install git, python3, numpy and pytorch==1.1 to your OS before running the program
  - An NVIDIA CUDA GPU is recommended to have for the system to speed up the training process for DQN agent. A list of CUDA GPUs can be found at https://developer.nvidia.com/cuda-gpus
- Clone our git repo
  - All of our project files and progress traces(commits) can be viewed at https://github.com/hanglearning/ucb-pacman-multiagent-reinforcement.
  - Use either HTTPS or SSH to clone our git repo by
    - `$ git clone https://github.com/hanglearning/ucb-pacman-multiagent-reinforcement.git`
    - Or
      `$ git clone git@github.com:hanglearning/ucb-pacman-multiagent-reinforcement.git`
- Run the program
  - Useful arguments(mostly mentioned in Structural Changes section)
    - '-p', '--pacmen'
      Specify the Pacman agent types to be used. Separated by a comma.
      E.g. `-p "ApproximateSarsaAgent, PacmanDQAgent"`
      Pacman type can be repeated to assign a different color. (Shown in the demo video)
    - '--pacmanAmounts'
      Set the desired amount to the corresponding Pacman agents

E.g. `-p “ApproximateSarsaAgent, PacmanDQAgent” --pacmanAmounts “3, 2”` will introduce 3 Sarsa and 2 DQN agents to the game

- ‘-x’, ‘--numTraining’
  Number of training iterations. Take an integer as input.
  E.g. `-x 100` will train all Pacman agents for 100 iterations. One iteration ends when all Pacman agents die in that iteration.
- ‘-n’, ‘--numGames’
  Total number of games. Take an integer as input. The absolute value of the difference between -x and -n is the number of iterations of the evaluation process.
  E.g. `-x 20 -n 32` will train all Pacman agents for 20 rounds and evaluate their performance for 12 rounds.
- ‘-l’, ‘--layout’
  Specify a layout to be used for training and/or evaluating. All available layout files are stored inside of the layouts/ folder. The layout file is human-readable and customizable.
  E.g. `-l capsuleClassic` will use capsuleClassic layout for the game
- ‘--trainingGraphics’
  Providing this argument will enable graphics display of the game during training iterations
- ‘--trainingGraphicsUpdateFrequency’
  While --trainingGraphics is provided, --trainingGraphicsUpdateFrequency can be used to specify the graphics update frequency during training, as displaying every training iteration for demonstration purpose can dramatically slow down the training progress.
  E.g. `-x 20 --trainingGraphics --trainingGraphicsUpdateFrequency 4` will display round 1, 5, 10, 15, and 20.
- ‘--evalGraphics’’
  Providing this argument will enable graphics display evaluation

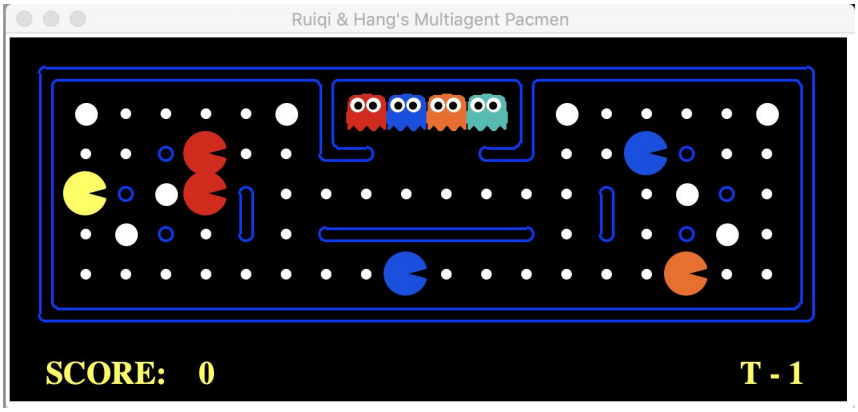
- Sample run
  - Argument Provided

```
$ python3 pacman.py -p
"ApproximateSarsaAgent, PacmanDQAgent,
ApproximateSarsaAgent, PacmanDQAgent" -x 20
-n 32 -l powerClassic --pacmanAmounts 1,2,2,1
--trainingGraphics
--trainingGraphicsUpdateFrequency 4
--evalGraphics
```

  - Console Output

```
evalGraphics set, games will be displayed in graphics during evaluation.
Training starts. Round 1, 5, 10, 15, and 20 will be displayed.
Game 1 displays.
Beginning 20 episodes of Training on ApproximateSarsaAgent index 0 with color Yellow
Beginning 20 episodes of Training on PacmanDQAgent index 1 with color Red
Beginning 20 episodes of Training on PacmanDQAgent index 2 with color Red
Beginning 20 episodes of Training on ApproximateSarsaAgent index 3 with color Blue
Beginning 20 episodes of Training on ApproximateSarsaAgent index 4 with color Blue
Beginning 20 episodes of Training on PacmanDQAgent index 5 with color Orange
```

  - Graphics Output



  - Video Demo
    https://youtu.be/X_JOLY-A14g (6:34)

Fork Our Project and Implement Your Own Reinforcement Agent

- Please refer to sarsaAgents.py or dqAgents.py for the format of a Pacman agent class and its attributes and methods. Once you are done with your agent class, save it as your_agent_type_Agents.py.

- Your agent class name is the input needed to be given to the `-p` argument. After you specify it, the program will pick up your agents automatically and introduce them to the game.
- More details at https://inst.eecs.berkeley.edu/~cs188/fa18/projects.html.