Wesley DeVore, Samuel Tapia
862081656, 892097969

CS 179M Final Report

Introduction

Our team is designed to learn a set of weights, its policy, through reinforcement learning that we learned through UC Berkeley's lecture slides on Q learning [1]. Through the use of a vector of features and a vector of weights we are able to use a linear function to approximate the value of the game state and action. Using these approximations allows the agents to perform their best action, or a random one while training, and will receive rewards based on the outcome.

Our agents work independently from one another, and we achieve this by using two separate policies. We made the design choice to actively go for an offensive and defensive agent, although both can participate in offense and defense if necessary. Each agent is rewarded with regards to their role. For our agents this means that only the offensive agent is rewarded for getting food and scoring, and the defensive one can only be rewarded for eating enemies. The agents will write different policies so that they can work independently from one another. Although our agents are independent they do communicate with each other. This happens through a shared sight of enemies around them and a separate class that interfaces with both of them to provide information to one another.

The agents are implemented using a method called approximate Q learning. This method allows the agent to adjust it's weights as it explores game states in the beginning of its learning phases. The agent explores more often in the beginning stages and as games progress it will more often exploit states it is already aware of when it knows the rewards. The learning sessions for our agents are about 1000 to 2000 games. We based the amount of sessions from a Stanford research paper on reinforcement learning [2], after that our agents will have a working policy. The learning

rate and exploration rate are then set to zero and it will be able to exploit, acting solely on the policy.

Structure

Our structure originated from the UC Berkeley's reinforcement learning python template that was geared towards the single player game of pacman [3]. Although our final structure does not use the code anymore, we did use the template as a starting point for base q-learning.

Design

The agent is able to learn through rewards and experiences using the approximate Q learning algorithm. The functionality that makes approximate Q learning work, comes from a few core functions which are the following:

- chooseAction
- update
- getQValue
- getReward

qTeam.py

This file is responsible for generating a team of two agents with specified policies. It is here where we have created the approximate Q learning class so that both our agents can inherit from it and learn through training episodes.

chooseAction - The core function that is required in order for any agent to work. The function takes in the current game state and with the use of exploration rate (epsilon) it either explores or exploits.

Using the epsilon value that we set at the initializer for the agent, the agent will make a random choice based on the probability of epsilon in a simple manner of a coin toss. The higher the epsilon value the more likely that it will choose to explore versus exploit, to make a random choice as its action. If the agent does not make a random choice it will take the best action possible. This allows our agent to explore more often in the beginning stages of its learning process and exploit more towards the end.

After the agent has selected its action to take, we check a buffer of past states to ensure that it is not full, if the buffer has reached capacity we remove the oldest state. If no action is available we return none, this happens at the terminal state of the game.

update - This function is what makes the approximate Q learning algorithm work [4]. It takes in the current game state and current reward. The function gathers the necessary values based on the formula in order to be able to adjust the agent's current weights, w_i. Update calls the finder class *getFeatures* function in order to propagate the current active features for the specified agent. Using *getQValue* we can determine the max Q value for the current state and action, and the previous Q value from the previous game state.

It is here where the agent uses the buffer with the to find the average current max Q value and reward. The buffer is used to mitigate the issue of the ballooning weights that occur over time. Therefore, instead of taking the difference of max Q value and the old Q value we take 100 samples from the buffer and obtain the average max q value and reward.

This allows for the agent's weights to stabilize over a larger amount of episodes to train on. The agent will then update w_i accordingly based on the difference, the current weights, and the active features.

Although the buffer supplies the agent with a lesser possibility of ballooning, there is still potential of a snowball effect to occur on the policy's weights. Thus, we apply a rubber band feature(L1 regularization) to bring the weights back down so that the values don’t go out of proportion.

getQValue - This function takes in a game state and action. Allowing the function to generate a new game state through the given action as a successor. Then computing the q value based on the successor’s current active features and current weights. It returns the Q value for the given action to take.

getReward - This function is overwritten within the child class this is because each agent plays a different role in the team. This function can be extended to hold base rewards for each agent, and then modified within the child class to fit the specific role needed for the agent.

finder.py

The finder class is the interface between the two agents, and also what determines the feature space. It is called at the creation of the team and is given to each agent so they can share information. The class can be extended to include new features, or modified to include only a set of the original features.

getFeatures - The core of the finder class, this function is called every time our agent needs to assess its current state or a successor state. The function creates a counter and adds features into it by calling their respective functions. These functions follow the guidelines of being numbers between zero and one to help prevent any ballooning of weights in the update. The only time a feature goes above one is when the distance between two places is zero (i.e. the next state is the goal state) or with the foodCarrying function which depends on how much food the agent is carrying.

Wesley DeVore, Samuel Tapia
862081656, 892097969

AddLocations - Updates the locations of food and enemies that can be used by the feature functions. This is called at the start of every chooseAction and is necessary because we get the features at states around the current position by using the successor, and depending on the action the available food or location of enemies can be changed from how it actually is. (if the agent steps onto the enemy pacman to eat it, the pacman is no longer there but back at its starting point which can cause problems when a getPosition is called)

updateMyFood - The only function that takes in the last state as well as the current state. It checks if the defending food list has changed and if it has will save the position of the changed food.

Features Implemented:

- closestFood: Returns the reciprocal of the distance from the current position to the closest food unless there are only two food left where it will go to zero.
- foodCarrying: This function recursively finds the minimum distance to the center by calling successor states, stopping when it becomes a ghost. If the successor state encounters an enemy ghost its position in the next state will be the initial position, and if that is detected the distance is set to a very high number so it will not be chosen if a better path is available.
- ghostsNear: This feature comes from the same function as the pacmanNear feature, it returns the reciprocal of the distance to the nearest ghost. The feature gets changed when the ghost detected is scared. It will ignore the ghost initially, as the timer is almost done it flips its value positive to incentivise the agent to eat it, then when it is only a few moves away from changing it goes back to normal.

- pacmanNear: This feature is similar to ghostNear and returns the reciprocal of the distance away from an enemy pacman. If the agent is scared the feature will flip its value negative when very close so as not to get eaten, but once it leaves the range of two moves away it goes back to normal allowing it to shade the pacman.
- nearestEatenFood: This function returns the distance away from the most recently eaten food. If there is no food eaten yet it targets the furthest away from the initial position. The positions around the target location all return a one so that the agent is free to explore in that space.
- deadend: This function returns the reciprocal of the distance away from a deadend. The function is given a count, which is how far away we want to check for a deadend. It recursively moves through states until it reaches a deadend in which case it uses that distance, or until the count is exhausted where it will determine that there is no deadend in range.
- inTunnel: This function returns a binary one or zero based on whether the agent is in a tunnel or not in one by checking the walls around it.

Agent1

The offensive agent class that inherits from the parent class *approximate Q learning*. This agent is designed to be more offensive minded. Therefore the *getReward* function awards the agent positively for doing actions that lead to scoring, eating food, and getting closer to food.

The agent is still awarded for eating pacman, but it is not as great as the other rewards since eating a pacman is considered to be more on the defensive side.

The end results of agent1 are as follows with the current policy:

- Try to eat a pacman if it is very near.
- Avoids ghosts, but prefers pellets more, if more than two pellets are nearby chances are it will try to get both.
- Typically likes to eat a single pellet and score, depending on the pellet's location.

Agent2

The defensive agent class, similarly to *Agent1*, This agent is more focused on being on the defensive side of the board. It is rewarded for eating pacman, staying near defending pellets, and going towards the most recent eating pellets. Unlike the offensive agent, this agent does not care about ghosts nor does it get a reward for seeing and or eating them.

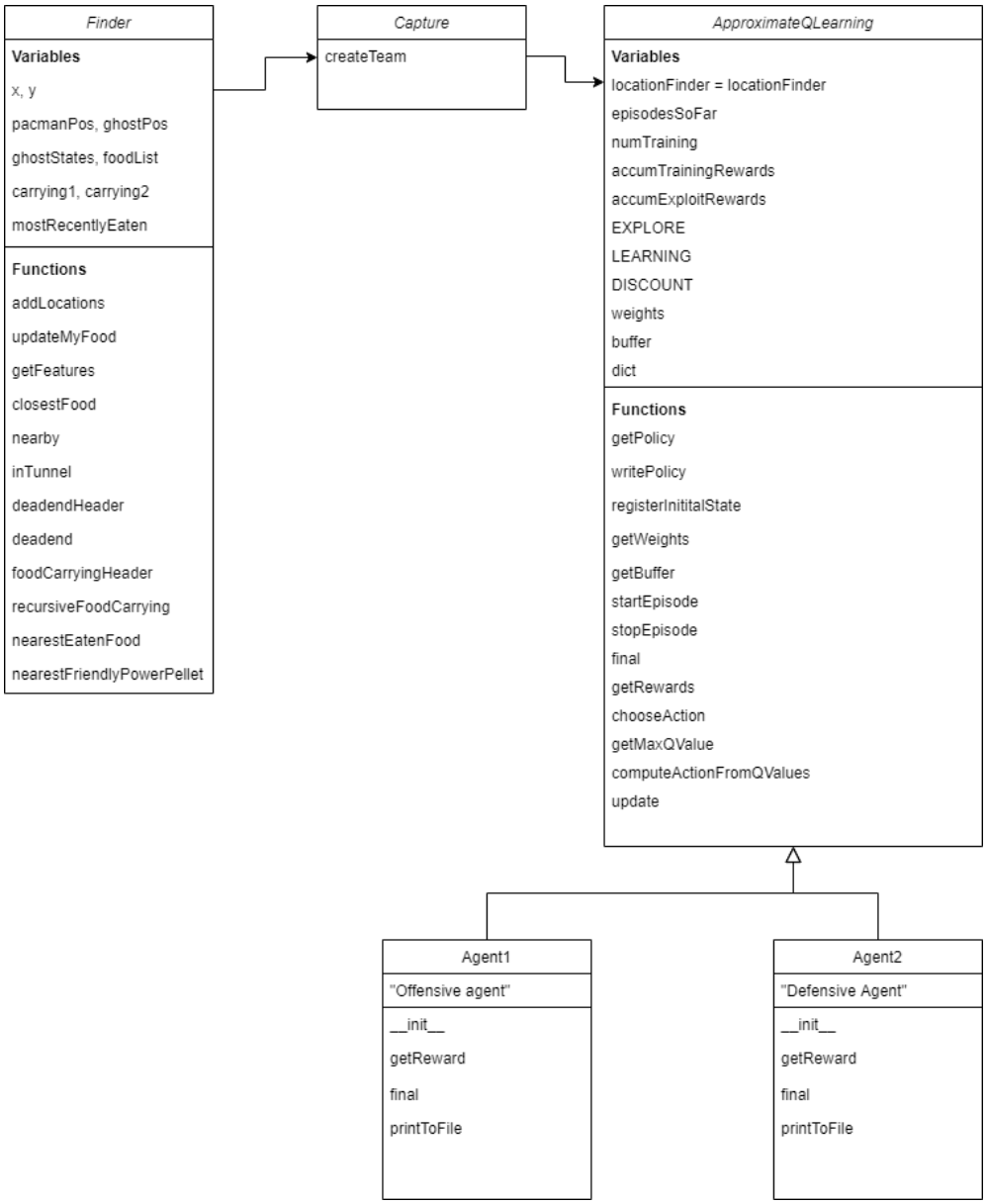
The end results of agent2 are as follows with the current policy:

- At the beginning of the game, it will typically go towards the middle of the board and hover around the nearest pellet.
- If a defending pellet is eating, the ghost will travel towards the last eaten pellet.
- If it sees a pacman it will chase the pacman until it no longer sees it or eats it.
- If scared the ghost will stay about 2 spaces away from the pacman.

Wesley DeVore, Samuel Tapia
862081656, 892097969

UML

The illustration of how two classes work with one another and how the agent child classes inherit from each other.



Wesley DeVore, Samuel Tapia
862081656, 892097969

Evaluation

Expectations

The agents are based on the rewards we set up for it to find and the features it uses. We hoped that by giving the defensive agent a high reward for eating a pac man would give it a high drive to find them. Also for the offensive agent we still allowed it to get rewarded for eating a pacman, but gave better rewards for eating food and scoring with hopes it would learn to prefer to do those activities. Additionally, we also encourage the offensive agent to seek out food, by giving small rewards as it gets closer to the food.

Observations

Our defensive agent met our expectations, the policy it generated gives it a high drive to seek out problem areas where food is missing and then hone in on the pac man to stop it from eating. The offensive agent was slightly more complicated, while we thought it should run away from ghosts, the weight it gave it was much smaller than we expected and only runs if it is incredibly close.

The agents we test against are straight forward, so many games will end with only a few points difference. The reason for this being the teams get into a back and forth with each other where neither makes a potentially freeing move because it would be sub optimal according to their own policies. This leads to games being won, but generally being boring to watch as once they gain a few points they can enter a deadlock state until the timer expires.

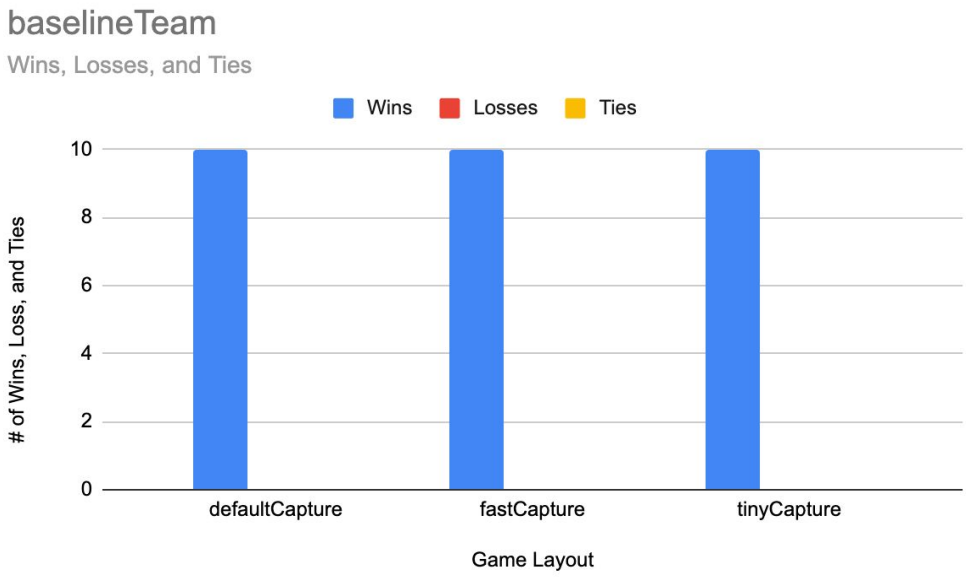
One of the aspects of our project that is unlike other reinforcement learning is that approximate q-learning does not converge to an optimal policy. It was up to us to test good policies that were generated by the training to find one that worked well enough to call our final policy.

Wesley DeVore, Samuel Tapia
862081656, 892097969

Results

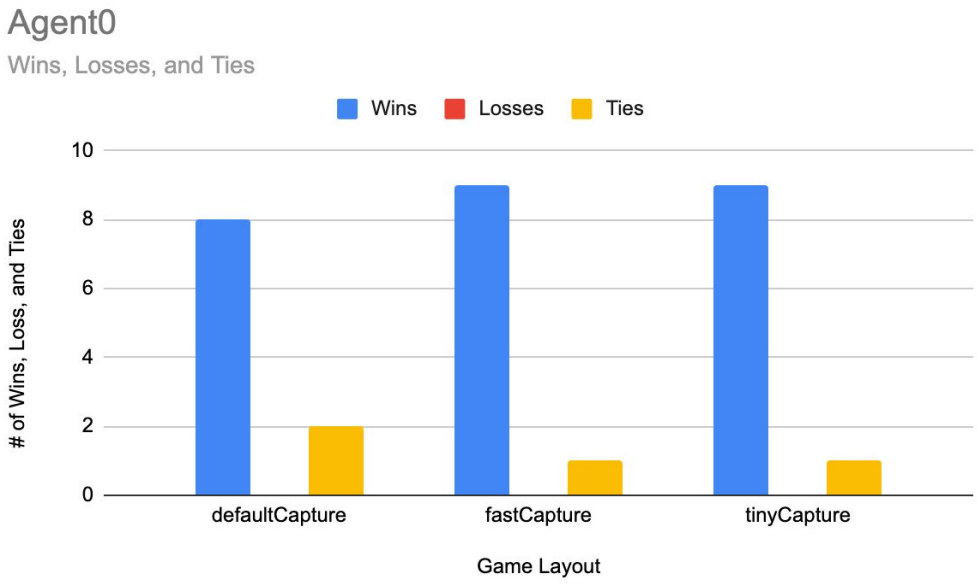
The current learned policies from both our agents performed very well against our previous design team, *myTeam.py (offensiveAgent, DefensiveDummyAgent)*, and the default team, *baselineTeam*. Turning off our team's learning functionality yields the results below, performing well above our expectations in the initial trials.

Against the baselineTeam we are performing at a 100% win rate, with no losses or ties over 10 rounds.



Against our original non-ai agent *Agent0*, we are performing at a 90% win rate, with no losses over 10 rounds.

Wesley DeVore, Samuel Tapia
862081656, 892097969



References

1. UC Berkeley's CS188 Slides: http://ai.berkeley.edu/slides/Lecture%2011%20--%20Reinforcement%20Learning%20II/SP14%20CS188%20Lecture%2011%20--%20Reinforcement%20Learning%20II.pptx
2. Reinforcement Learning Research Paper By: Abeynaya Gnanasekaran, Jordi Feliu Faba, Jing Anhttp://cs229.stanford.edu/proj2017/final-reports/5241109.pdf
3. UC Berkeley's Reinforcement Learning Python Template: http://ai.berkeley.edu/projects/release/reinforcement/v1/001/docs/qlearningAgents.html
4. UC Berkeley's CS188 Lab: http://ai.berkeley.edu/reinforcement.html#Q8