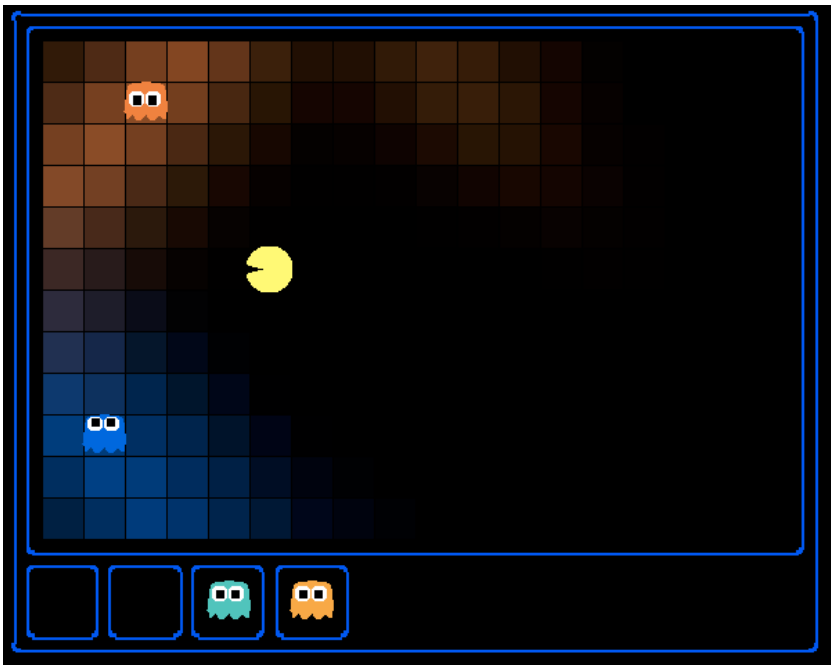
Project 4

Ghostbusters

Due Date: 2. November 2017, 11:59 PM



I can hear you, ghost.
Running won't save you from my
Particle filter!

Introduction

Pacman spends his life running from ghosts, but things were not always so. Legend has it that many years ago, Pacman's great grandfather Grandpac learned to hunt ghosts for sport. However, he was blinded by his power and could only track ghosts by their banging and clanging.

In this project, you will design Pacman agents that use sensors to locate and eat invisible ghosts. You'll advance from locating single, stationary ghosts to hunting packs of multiple moving ghosts with ruthless efficiency.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's judgements -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know.

These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, classes, and the discussion forum are there for your support; please use them. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Discussion: Please be careful not to post spoilers.

Questions: We will post for each project a document like this containing all questions you have to work on. Although our course is based upon the CS188 AI class at Berkeley, **we might change, add or remove tasks**! Thus, make always sure to read this document.

Files to Edit and Submit: You will fill in portions of `bustersAgents.py and inference.py` during the assignment. You should submit this file with your code and comments. Please *do not* change the other files in this distribution or submit any of our original files other than this file. Submit the file via Blackboard.

Please submit your Pacman files (Task 1-7) in one container (.zip) and your DRL files (Task 8) in another container (.zip). In total, you should add two containers to your submission on Blackboard.

Files you'll edit:

`bustersAgents.py`	Agents for playing the Ghostbusters variant of Pacman.
`inference.py`	Code for tracking ghosts over time using their sounds.
`analysis.py`	A file to put your answers to questions given in the project.

Files you will NOT edit:

`busters.py`	The main entry to Ghostbusters (replacing Pacman.py)
`bustersGhostAgents.py`	New ghost agents for Ghostbusters
`distanceCalculator.py`	Computes maze distances
`game.py`	Inner workings and helper classes for Pacman
`ghostAgents.py`	Agents to control ghosts

`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents
`util.py`	Utility functions
`autograder.py`	Project autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question

Ghostbusters and BNs

In the CSE571 version of Ghostbusters, the goal is to hunt down scared but invisible ghosts. Pacman, ever resourceful, is equipped with sonar (ears) that provides noisy readings of the Manhattan distance to each ghost. The game ends when Pacman has eaten all the ghosts. To start, try playing a game yourself using the keyboard.

```
python busters.py
```

The blocks of color indicate where the each ghost could possibly be, given the noisy distance readings provided to Pacman. The noisy distances at the bottom of the display are always non-negative, and always within 7 of the true distance. The probability of a distance reading decreases exponentially with its difference from the true distance.

Your primary task in this project is to implement inference to track the ghosts. For the keyboard based game above, a crude form of inference was implemented for you by default: all squares in which a ghost could possibly be are shaded by the color of the ghost. Naturally, we want a better estimate of the ghost's position. Fortunately, Bayes' Nets provide us with powerful tools for making the most of the information we have. Throughout the rest of this project, you will implement algorithms for performing both exact and approximate inference using Bayes' Nets. The lab is challenging, so we do encouarge you to start early and seek help when necessary.

While watching and debugging your code with the autograder, it will be helpful to have some understanding of what the autograder is doing. There are 2 types of tests in this project, as differentiated by their `*.test` files found in the subdirectories of the `test_cases` folder. For tests of class `DoubleInferenceAgentTest`, your will see visualizations of the inference distributions generated by your code, but all Pacman actions will be preselected according to the actions of the staff implementation. This is necessary in order to allow comparision of your distributions with the staff's distributions. The second type of test is `GameScoreTest`, in which your `BustersAgent` will actually select actions for Pacman and you will watch your Pacman play and win games.

As you implement and debug your code, you may find it useful to run a single test at a time. In order to do this you will need to use the -t flag with the autograder. For example if you only want to run the first test of question 1, use:

```
python autograder.py -t test_cases/q1/1-ExactObserve
```

In general, all test cases can be found inside test_cases/q*.

---

1. Question (3 points): Exact Inference Observation

In this question, you will update the `observe` method in `ExactInference` class of `inference.py` to correctly update the agent's belief distribution over ghost positions given an observation from Pacman's sensors. A correct implementation should also handle one special case: when a ghost is eaten, you should place that ghost in its prison cell, as described in the comments of `observe`.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q1
```

As you watch the test cases, be sure that you understand how the squares converge to their final coloring. In test cases where is Pacman boxed in (which is to say, he is unable to change his observation point), why does Pacman sometimes have trouble finding the exact location of the ghost?

Note: your busters agents have a separate inference module for each ghost they are tracking. That's why if you print an observation inside the `observe` function, you'll only see a single number even though there may be multiple ghosts on the board.

Hints:

- You are implementing the online belief update for observing new evidence. Before any readings, Pacman believes the ghost could be anywhere: a uniform prior (see `initializeUniformly`). After receiving a reading, the `observe` function is called, which must update the belief at every position.
- Before typing any code, write down the equation of the inference problem you are trying to solve.
- Try printing `noisyDistance`, `emissionModel`, and `PacmanPosition` (in the `observe` function) to get started.
- In the Pacman display, high posterior beliefs are represented by bright colors, while low beliefs are represented by dim colors. You should start with a large cloud of belief that shrinks over time as more evidence accumulates.
- Beliefs are stored as `util.Counter` objects (like dictionaries) in a field called `self.beliefs`, which you should update.
- You should not need to store any evidence. The only thing you need to store in `ExactInference` is `self.beliefs`.

---

2. Question (4 points): Exact Inference with Time Elapse

n the previous question you implemented belief updates for Pacman based on his observations. Fortunately, Pacman's observations are not his only source of knowledge about where a ghost may be. Pacman also has knowledge about the ways that a ghost may move; namely that the ghost can not move through a wall or more than one space in one timestep.

To understand why this is useful to Pacman, consider the following scenario in which there is Pacman and one Ghost. Pacman receives many observations which indicate the ghost is very near, but then one which indicates the ghost is very far. The reading indicating the ghost is very far is likely to be the result of a buggy sensor. Pacman's prior knowledge of how the ghost may move will decrease the impact of this reading since Pacman knows the ghost could not move so far in only one move.

In this question, you will implement the `elapseTime` method in `ExactInference`. Your agent has access to the action distribution for any `GhostAgent`. In order to test your `elapseTime` implementation separately from your `observe` implementation in the previous question, this question will not make use of your `observe` implementation.

Since Pacman is not utilizing any observations about the ghost, this means that Pacman will start with a uniform distribution over all spaces, and then update his beliefs according to how he knows the Ghost is able to move. Since Pacman is not observing the ghost, this means the ghost's actions will not impact Pacman's beliefs. Over time, Pacman's beliefs will come to reflect places on the board where he believes ghosts are most likely to be given the geometry of the board and what Pacman already knows about their valid movements.

For the tests in this question we will sometimes use a ghost with random movements and other times we will use the GoSouthGhost. This ghost tends to move south so over time, and without any observations, Pacman's belief distribution should begin to focus around the bottom of the board. To see which ghost is used for each test case you can look in the .test files.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q2
```

As an example of the GoSouthGhostAgent, you can run

```
python autograder.py -t test_cases/q2/2-ExactElapse
```

and observe that the distribution becomes concentrated at the bottom of the board.

As you watch the autograder output, remember that lighter squares indicate that pacman believes a ghost is more likely to occupy that location, and darker squares indicate a ghost is less likely to occupy that location. For which of the test cases do you notice differences emerging in the shading of the squares? Can you explain why some squares get lighter and some squares get darker?

Hints:

- Instructions for obtaining a distribution over where a ghost will go next, given its current position and the `gameState`, appears in the comments of `ExactInference.elapseTime` in `inference.py`.
- We assume that ghosts still move independently of one another, so while you can develop all of your code for one ghost at a time, adding multiple ghosts should still work correctly.

---

3. Question (3 points): Exact Inference Full Test

Now that Pacman knows how to use both his prior knowledge and his observations when figuring out where a ghost is, he is ready to hunt down ghosts on his own. This question will use your `observe` and `elapseTime` implementations together, along with a simple greedy hunting strategy which you will implement for this question. In the simple greedy strategy, Pacman assumes that each ghost is in its most likely position according to its beliefs, then moves toward the closest ghost. Up to this point, Pacman has moved by randomly selecting a valid action.

Implement the `chooseAction` method in `GreedyBustersAgent` in `bustersAgents.py`. Your agent should first find the most likely position of each remaining (uncaptured) ghost, then choose an action that minimizes the distance to the closest ghost. If correctly implemented, your agent should win the game in `q3/3-gameScoreTest` with a score greater than 700 at least 8 out of 10 times. *Note:* the autograder will also check the correctness of your inference directly, but the outcome of games is a reasonable sanity check.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q3
```

Note: If you want to run this test (or any of the other tests) without graphics you can add the following flag:

```
python autograder.py -q q3 --no-graphics
```

Hints:

- When correctly implemented, your agent will thrash around a bit in order to capture a ghost.
- The comments of `chooseAction` provide you with useful method calls for computing maze distance and successor positions.
- Make sure to only consider the living ghosts, as described in the comments.

4. Question (3 points): Approximate Inference Observation

Approximate inference is very trendy among ghost hunters this season. Next, you will implement a particle filtering algorithm for tracking a single ghost.

Implement the functions `initializeUniformly`, `getBeliefDistribution`, and `observe` for the `ParticleFilter` class in `inference.py`. A correct implementation should also handle two special cases. (1) When all your particles receive zero weight based on the evidence, you should resample all particles from the prior to recover. (2) When a ghost is eaten, you should update all particles to place that ghost in its prison cell, as described in the comments of `observe`. When complete, you should be able to track ghosts nearly as effectively as with exact inference.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q4
```

Hints:

- A particle (sample) is a ghost position in this inference problem.
- The belief cloud generated by a particle filter will look noisy compared to the one for exact inference.
- `util.sample` or `util.nSample` will help you obtain samples from a distribution. If you use `util.sample` and your implementation is timing out, try using `util.nSample`.

---

5. Question (4 points): Approximate Inference with Time Elapse

Implement the `elapseTime` function for the `ParticleFilter` class in `inference.py`. When complete, you should be able to track ghosts nearly as effectively as with exact inference.

Note that in this question, we will test both the `elapseTime` function in isolation, as well as the full implementation of the particle filter combining `elapseTime` and `observe`.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q5
```

For the tests in this question we will sometimes use a ghost with random movements and other times we will use the GoSouthGhost. This ghost tends to move south so over time, and without any observations, Pacman's belief distribution should begin to focus around the bottom of the board. To see which ghost is used for each test case you can look in the .test files. As an example, you can run

```
python autograder.py -t test_cases/q5/2-ParticleElapse
```

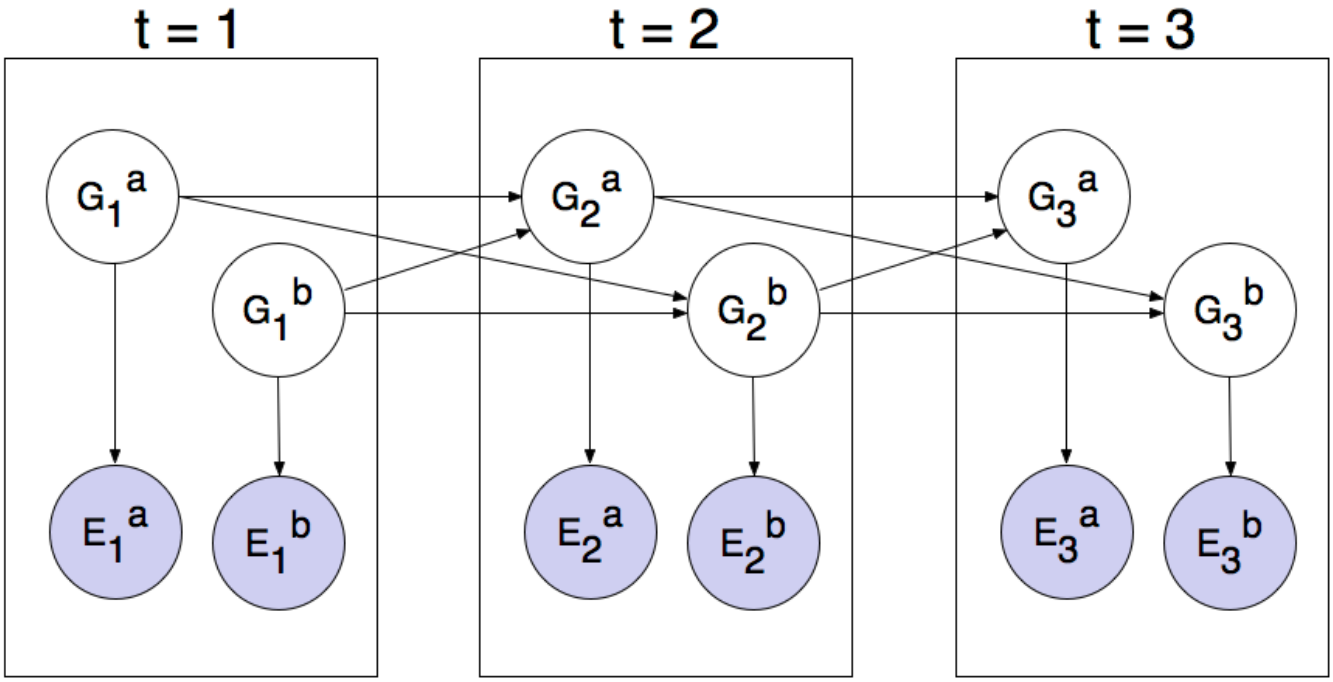
and observe that the distribution becomes concentrated at the bottom of the board.

---

6. Question (4 points): Joint Particle Filter Observation

So far, we have tracked each ghost independently, which works fine for the default `RandomGhost` or more advanced `DirectionalGhost`. However, the prized `DispersingGhost` chooses actions that avoid other ghosts. Since the ghosts' transition models are no longer independent, all ghosts must be tracked jointly in a dynamic Bayes net!

The Bayes net has the following structure, where the hidden variables G represent ghost positions and the emission variables E are the noisy distances to each ghost. This structure can be extended to more ghosts, but only two (a and b) are shown below.



You will now implement a particle filter that tracks multiple ghosts simultaneously. Each particle will represent a tuple of ghost positions that is a sample of where all the ghosts are at the present time. The code is already set up to extract marginal distributions about each ghost from the joint inference algorithm you will create, so that belief clouds about individual ghosts can be displayed.

Complete the `initializeParticles`, `getBeliefDistribution`, and `observeState` method in `JointParticleFilter` to weight and resample the whole list of particles based on new evidence. As before, a correct implementation should also handle two special cases: (1) When all your particles receive zero weight based on the evidence, you should resample all particles from the prior to recover. (2) When a ghost is eaten, you should update all particles to place that ghost in its prison cell, as described in the comments of `observeState`.

You should now effectively track dispersing ghosts. To run the autograder for this question and visualize the output:

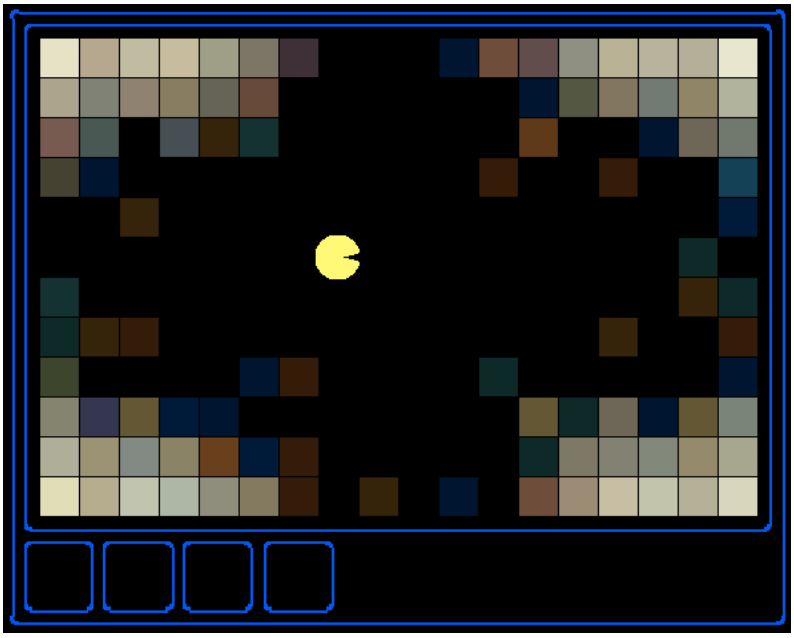
```
python autograder.py -q q6
```

7. Question (4 points): Joint Particle Filter with Elapse Time

Complete the `elapseTime` method in `JointParticleFilter` in `inference.py` to resample each particle correctly for the Bayes net. In particular, each ghost should draw a new position conditioned on the positions of all the ghosts at the previous time step. The comments in the method provide instructions for support functions to help with sampling and creating the correct distribution.

Note that completing this question involves removing the call to util.raiseNotDefined(). This means that the autograder will now grade both question 6 and question 7. Since these questions involve joint distributions, they require more computational power (and time) to grade, so please be patient!

As you run the autograder note that `q7/1-JointParticleElapse` and `q7/2-JointParticleElapse` test your `elapseTime` implementations only, and `q7/3-JointParticleElapse` tests both your `elapseTime` and `observe` implementations. Notice the difference between test 1 and test 3. In both tests, pacman knows that the ghosts will move to the sides of the gameboard. What is different between the tests, and why?



To run the autograder for this question use:

```
python autograder.py -q q7
```

Congratulations!

Question 8 (14 + 3 points): Deep Reinforcement Learning

Note that the autograder will not grade question 8. For this task, you need to work unsupervised. The same submission guidelines apply as for the other projects. Please submit your Pacman files (Task 1-7) in one container (.zip) and your DRL files (Task 8) in another container (.zip). In total, you should add two containers to your submission on Blackboard.

5.1 (8 Points) Complete the implementation of the Deep Q-Learning algorithm in TensorFlow to solve the LunarLander environment from OpenAI gym. We have provided a shell in which to implement your code. For reference, our basic DQN implementation achieves a score of about 100 in just a few minutes of training with relatively little tuning and continues to improve. A score over 200 is very good.

Some tips:

- You must use a function approximation method that predicts Q-values. This can be a linear approximator or a neural network.

- You may implement any of the extensions proposed by DeepMind, referenced in the lecture or other ideas you may have.

- Reinforcement learning algorithms, especially Q-learning with function approximation, can be unstable. Fix a random seed to remove some variability. Setting a random seed helps to debug implementation problems as well.

Grading:

- The starter code also saves periodic snapshots of any variables created in the TensorFlow session. You will be required to submit these with your code.

- Your grade will be determined by the reward achieved by your agent on a set of random seeds.

-The project code is currently configured to run an evaluation episode after a fixed period. This function will be used to determine your score. Note that this calls *select_action(obs, evaluation_mode=True).* Make sure that this function call works properly.

- We are familiar with many of the available implementations on GitHub. While these can be good references, do not copy the code. We have seen that code, too, and won't be fooled.

5. 2 (4 Points) After solving (part 1), examine the effect that exploration has on your algorithm's convergence to a quality policy. For example, if using epsilon greedy exploration, what happens when epsilon is set very low (say 0.05) from the start of training? What happens when epsilon is set very high (say 0.95) and never decreased? Plot the learning curves from both runs.

If epsilon is low, then the learning rate will reduce.
This is because no new scenario is encountered.

5.3 (2 Points) For a couple states, look at the Q-values that your trained network predicts. Do they appear accurate? Suppose they are not accurate, what property must the set of Q-values have in order for the optimal actions to still be chosen?

5.4 (Bonus: 3 Points) Solve the same LunarLander task using image input instead of the low dimensional state vector.

Some tips:

-An example wrapper is provided to setup the environment to train from pixels.

- To solve this environment using DQN you should use a convolutional network

- Note: without adding any extensions, such as prioritized experience replay, this will likely take at least a full day to train.

- You may change the input processing as long as the input remains an image. To do so, edit the environment wrapper so that the observation is return as usual from env.step(action).

- Please submit your code, a trained model, a comment detailing the intuition behind your decisions, and the estimated time to run your code.