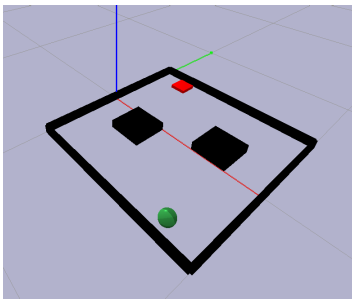
This assignment does not count toward the final grade.

Project 1

Due Apr 13 by 11:59pm **Points** 15

This project aims to demonstrate how classical machine learning methods can be used in robotics setting. In this project, we will working on a navigation agent that navigates inside a simple 2D maze.



The image above shows the simulation world. The "robot" (also called "agent") is shown by the green dot. The goal location is shown by the red square. The aim of the agent is to navigate to the goal.

The ultimate goal in this projects is to learn an appropriate behavior for the agent by imitating demonstrations from an expert user. These demonstrations have been collected by a human controlling the agent via a keyboard, and will be provided to you as training data.

Note that in this project we are explicitly not allowing the use of Deep Learning or Reinforcement Learning. We will be using these in future projects, which will allow us to see what significant benefits these technologies bring. Until then, this project is restricted to "traditional" supervised learning.

Part 0. Project Setup

Check out Project 1 from the SVN server:

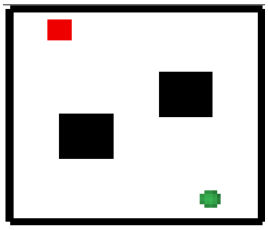
```
svn co svn+ssh://YOUR_UNI_HERE@roamopenserver.me.columbia.edu/home/YOUR_UNI_HERE/assignments/project1
```

Install the virtual environment for the project:

```
cd project1
pipenv install --dev
```

Important: DO NOT install any other dependencies or a different version of already provided package as we will be grading you against the virtual environment created by the provided Pipfile.lock

Part I. Inferring the position of an agent with RGB images



The first task is to learn to infer where the agent is inside the maze based on RGB image observations like the one shown above. Each such observation will consist of an RGB image of size [64, 64] for each color channel, so the total size of each observation is [64, 64, 3].

The maze has its own coordinate system, in which the agent's location must be expressed. You will be provided with RGB image observations in this environment, as well as the groundtruth location of the agent in each image, expressed in the maze coordinate system. The task is to learn a model that can predict the location of an agent given this RGB observation.

Note that this can be seen as a regression problem (if the location of the agent is a continuous variable) or a classification problem (if we discretize the output space to a finite number of possible locations).

In this part, you will need to implement a class that inherits from base.Regressor. Your class will need to implement two methods to get a score. The methods to implement are the following:

```
base.Regressor.train(self, data): A method that train a regressor with given data

"""
Args:
    data: a dictionary that contains images and the groundtruth location of an agent.
Returns:
    Nothing
"""

base.Regressor.predict(self, Xs): predict the output values given a batch of X's and return the outputs.

"""
Args:
    Xs: a batch of data (in this project, it is in shape [batch_size, 64, 64, 3])
Returns:
    The fed-forward results (predicted y's) with a trained model.
"""
```

We will test the performance of your model in this part using mean square error between the predicted positions and the groundtruth. We will perform this evaluation on both the training data which is provided to you, and which your model will be training with, and on some additional testing data that is held out.

Please implement your class in the file solutions/pos_regressor.py.

Part II. Behavioral cloning with low dimensional data

In this part, your model is asked to determine what action the agent should take, based on an observation from its environment. The action can be one of three choices: go up, go left, or go right. The goal of the agent is to reach the goal squre, shown in red in the images above.

Note that, in general terms, what you are providing here is a "policy" - a model that selects an action based on observations from the world. Numerous methods are available for training policies, and we will cover many of them in the Reinforcement Learning part of the class.

However, learning a policy can also sometimes be a Supervised Learning problem: you will be provided with labeled examples from an "expert". Each labeled example *i* will contain a tuple of the form (**o**, a)^{i}, where **o** represents an observation and a represents the action taken by the expert given that observation. You must simply learn to imitate the expert, a process also known as behavioral cloning. If the action space is discrete, then behavioral cloning is a classification problem and if the action space is continuous, it will be a regression problem.

In this project, we will be working on an environment with discrete action space, so we can see behavioral cloning as a classification problem with three output classes (go up, go left, go right). While the action space is the same in Parts II and III, the nature of the observation used in each case will be different.

In Part II, the observation will consist of the ground truth position of the agent in the maze coordinate system. Training data will thus contain tuples $(\mathbf{o}, a)^i$ where **o** is the agent's location in the maze, and a is the action taken by the expert at that location. You can use any classification method from Scikit-learn to learn the mapping between observations and actions.

You will be asked to implement a class that inherits from base.RobotPolicy. You need to implement two abstract methods from this abstract class. The methods you need to implement are the followings:

```
base.RobotPolicy.train(self, data): train the policy with data provided.

"""
Args:
    data: a dictionary that contains X (observations) and y (actions).
Returns:
    This method does not return anything. It will just need to update the
    property of a RobotPolicy instance.
"""

base.RobotPolicy.get_actions(self, observations): get actions given a batch of observations.

"""
Args:
    Xs: a batch of data (in this project, it is in shape [batch_size, 64, 64, 3])
Returns:
    The fed-forward results (predicted y's) with a trained model.
"""
```

In this part, we will evaluate your model by simply having the robot follow the commands that it provides, or, in other words, "rolling out your policy" in the environment. After 20 steps, we will evaluate how close to the goal the robot has ended up. Formally, the score for a single run will be calculated based on the minimum distance between your agent and the target location achieved over a trajectory of 100 steps. We will run your agent for 20 times in the environment and use the following formula to calculate your score: score = (Init_dist - min_dists/20) / Init_dist.

Please implement your class in solutions/pos_bc_robot.py

Part III. Behavioral cloning with visual observations

In this part, you asked to do a similar task as Part II, but the observations will be a lot more challenging to use. Instead of being provided with the actual robot location, your model will receive as input RGB image observations of the world, similar to the ones you used to do localization in Part I.

All requirements from your code, as well as the evaluation method, are unchanged compared to Part II. The only difference is the nature of the observation that is provided to you.

Please implement your class in solutions/rgb_bc_robot.py

Testing:

We will be using score_policy.py to auto-generate your score for this project. To see how you are doing with the project, simply run:

```
pipenv run python score_policy.py
```

If you would like to visualize your policy just pass --gui flag. This does take longer so feel free to terminate with Ctrl-C

```
pipenv run python score_policy.py --gui
```

Note that the solution files provided to you return zeros. You are required to update it with your solution.