Reinforcement Learning Library Documentation

Arjun Gupta

January 26, 2018

Contents

0.1 Preface

The goal of this library is to allow for automatic intelligent behavior creation to aid in the Aquaticus project of capture the flag. However, it can also be used for many other marine applications. The current infrastructure provides the ability to train arbitrarily complex behaviors by setting a reward function, defining an action space, and then providing a simulation where one robot is exhibiting the learned behavior. While crafted with the intention of doing reinforcement learning on a game of capture the flag, with some minor modifications (outlined in later sections), this library can be used for a much wider array of applications.

0.2 Dependencies

This library requires Python2.7 (comes pre-installed on recent MacOS) as well as a number of Python packages. If you are not running on MacOS and do not already have Python2.7 installed, you can download it here: https://www.pyt hon.org/downloads/release/python-2714/. After Python has been successfully installed, run the following commands in order to install the dependencies.

```
$ pip install numpy
$ pip install matplotlib
$ pip install tensorflow
$ pip install keras
```

0.3 File Overview

0.3.1 Directory Structure

Learning_Code/ - algorithms, data and models to do reinforcement learning

- **Constants.py** - holds state definition, file paths and hyperparameters
- **DeepLearn.py** - class to do reinforcement learning
- **Reinforce.py** - makes action space and reward function, calls DeepLearn.py to do reinforcement learning
- **Clean.sh** - cleans the simulation engine folder to remove extraneous logs
- **log_converter.py** - converts logs/ from results to better formatted csv files and places them into processed/
- **run.sh** - Call test simulation to check on model progress
- **table_generator.py** - trial table generator file
- **interpreted.csv** - output from BHV_Input to test that it is running correctly
- **table.csv** - output from DeepLearn.py, read in by BHV_input to get setup information and location of the model
- **models/** - holds the saved models that have been trained, each model directory has a file called environment.txt that is a snapshot of the fields in Constants.py when the model was initialized
- **paths/** - holds the information about previous paths that we have seen in simulation as a check to see that we are getting the right data
- **processed/** - holds processed data from simulation
- **results/** - raw grepped data from simulations

Simulation_Engine - files and folders to run MOOS

- **plug_*** - plug files shared between mokai, m200, and motorboat meta.moos files
- **launch_model.sh** - launches simulation with pMarineViewer enabled to test simulation (called by run.sh)
- **launch_reinforce.sh** - launches simulation for reinforcement learning (no pMarine viewer)

launch_simulation.sh - launches simulation with two m200's and two mokais per team

m200/ - m200 vehicle configuration files

mokai/ - mokai vehicle configuration files

motorboat/ - motorboat vehicle configuration files

shoreside/ - shoreside configuration files

0.3.2 File Interaction Diagram

...fill in later

0.3.3 Scripts for Learning

Constants.py - contains all major constants used by the learning module.This includes indeces for where values are in the state representation, speeds, headings, reference files for scripts, as well as files to read and write data to. This is where major program parameters can be set and will be carried through to the other scripts (although for some changes the other scripts will also have to be modified)

Reinforce.py - This is the main piece of code that is called to build and train the learning agent. It makes the statespace, actionspace, and the reward function to be passed to the learning agent. It then initialize and trains the agent based on the parameters given in *Constants.py*. Call with *load* flag to load in parameters from the designated directory in *Constants.py*, *new* to make a new model, and *test* to output an optimal table of values, and call *run.sh* to see how the model is doing (make sure to clean the simulation_engine folder afterwards)

DeepLearn.py - Defines the *deep_learn* class which provides functions for initializing and training a deep reinforcement learning algorithm using keras. Constants used throughout the code are derived from *Constants.py*. Requires that the computer has Keras, Tensorflow, and Matplotlib installed. Outputs *table.csv* to the *reinforcement/learning_code* folder to be consumed by *BHV_Input*.

0.3.4 Scripts for Running Simulations

Tester.sh - Runs the simulation for a set amount of time and then kills it. Then aloggreps the relevant MOOSLogs for INP_STAT to get state and action information and writes the resulting .csv to results/

launch_reinforce.sh - Launches the simulation with Felix on the red team exhibiting the behavior that implements the outputted table.

run.sh - simple script to run a simulation with the current table and pMarineViewer running

0.3.5 Scripts for Parsing Simulation Data

log_converter.py - Extracts important information from files in the results/ directory and writes it to a new file in the processed directory. (these can be changed in the Constants.py file to be different directories)

0.4 Algorithm Types

0.4.1 Fitted Q Learning

The algorithm makes one neural net per action which takes in a state as input and outputs the predicted reward of doing that action at that state. The goal is to correctly approximate the Q-Value function which maps state, action pairs to their expected reward. Once trained, the robot acts by choosing the action for which the expected reward is greatest given the state that it is currently in.

Basic Algorithm

...fill in later

0.4.2 Deep Q Learning

The algorithm makes one neural net that takes in a state representation as input and outputs a the expected reward for each possible action. The neural net has one output node for each possible action in the action space. Then, like with Fitted Q Learning, the algorithm acts by picking the action with the maximum expected reward. The benefits of DQN over the fitted algorithm is that it is faster to train due to the fact that it has a single neural net instead of multiple and when it learns the weights for the correct output for one action for some state, it may help in learning the correct weights for other actions for that state. However, it also has less expressibility in terms of how complex a function it can approximate since all but the last layer of the neural network have shared weights for all the actions.

Basic Algorithm

...fill in later

0.5 Making a Model

Change the *user_path* variable to the path to your moos-ivp-extend tree, and change the m_infile parameter in *BHV_Input.cpp* to be the path to *table.csv* on your local machine (the same path as $Constants.out_address$). Change the parameters in Constants.py to form the model as desired. More specifics on the parameters in Constants.py are below. Once the parameters are set, the model is ready to be trained.

0.5.1 Constants.py

Constants.py contains all the important information for the training and testing algorithms and is the primary file that needs to be edited to customize the machine learning model. Before editing any other portions of Constants.py, make change the *user_path* variable to the path to your moos-ivp-extend tree. Constants.py has four main parts listed below.

State Definition

The state class is a way of holding together all information about the state, it has the fields:

index location in the input array that the parameter resides

type the type of information the state contains. This can be either "binary", "distance", "angle", or "raw". "raw" can only be used to select for data about the vehicle it is on for now

standardized A boolean flag of whether or not to scale the variable to be between 0 and 1 (based on the rang parameter). It is recommended that it be set to true for better neural net performance

range A tuple that is used for variables that are not of type "binary." Sets the minimum and maximum value that this variable can have and is used to standardize the variable between 0 and 1 if the *standardized* flag is set to true

vehicle the vehicle who's info we are using. This is used when finding the distance or the angle to another vehicle. Set this parameter to the name of the vehicle whose distance or angle is desired, and set *var* to "player." The default value is "self."

var The variable which is providing the information. Can be any of the following:

- **flag** can be used to get the distance or angle from the vehicle to the flag. Needs to define *var_mod* as either "self" (for own flag) or "enemy" (for enemy flag)
- **team** binary variable that is 1 if blue team and 0 if red team.
- **tagged** binary variable that is 1 if the vehicle is tagged and 0 if not tagged
- **has_flag** binary variable that is 1 if enemy flag captured and 0 otherwise
- **player** used when vehicle is not self. Returns the distance or angle (as specified in type) from the vehicle to the robot running the current behavior
- **x/y/heading** these can be used with "raw" state parameters. They will provide the raw x coordinate, y coordinate, or *heading* of the vehicle running BHV_Input

var_mod Additional variable information such as "self" or "enemy" which are used for deciding which flag to use if asking for flag info.

bucket The size chunks that this variable is discretized into. Set to 1 as default (not discretized).

Constants.py creates a dictionary (hashmap) called *state* which contains all the definitions for different parts of the state for easy access. These definitions are automatically passed to all the other files, including BHV_Input, to make the correct state definition and pass it through the files automatically.

Hyper Parameters

The second section of Constants.py contains the terms that control how the neural net is structured, and how it is trained. These parameters are important for making a neural net that is good at approximating the Q-value, and thus generating a robust behavior. Documentation for these parameters can be found in Constants.py

Reinforcement Learning Parameters

These are parameters that are used in the reward function and learning functions, define the action space that the robot uses, and decide how often to save snapshots of the model. Documentation for these parameters can be found in Constants.py

File Paths

The last section of Constants.py defines the system paths to all the important files that are being used by the learning algorithms. These parameters need to be edited whenever a new model is created, or a model is loaded. More specific documentation is available in Constants.py.

0.5.2 Defining the Reward Function

The reward function is defined in reinforce.py. As is, it checks whether the robot is tagged, and gives it a negative reward if that is the case. Otherwise it gives the robot a reward proportional to its distance away from the enemy flag. If these parameters no longer exist in *Constants.state* or a new reward function is needed, just redefine the function in reinforce.py as desired.

0.6 Training the Model

To train a new model, set the desired parameters in Constants.py, and more importantly, set *Constants.save_model_dir* to a new directory name in the *model/* folder. Then run *python reinforce.py new* to start training a new model. To continue training a previous model, set *Constants.load_model_dir* to the name of the directory that holds that model. Then run *python reinforce.py load* to continue training the model that is loaded.

0.7 Loading and Testing Existing Models

To load and test one specific model, specify the directory path to that model in Constants.py in the *Constants.load_model_dir*, then call *python reinforce.py test* to output an optimal table for that action. Call *./run.sh* to see how that model performs. To see how multiple models are doing and graph their relative scores, specify the directory where the model iteration folders reside in *Constants.test_address* and call *python reinforce.py test num* where *num* is the number of different iteration directories there are. The program will run 5 simulations for each model and report the average score. At the end, it will output the model with the highest average score and display a graph of the scores for the different models.