# pLearn: A Reinforcement Learning Platform for MOOS-IvP

**Spring 2018**

Arjun Gupta, argupta@mit.edu
Department of Computer Science
MIT, Cambridge MA 02139



## 1 pLearn: A Reinforcement Learning Platform for MOOS-IvP

The goal of the `pLearn` platform 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 platform can be used for a much wider array of applications.

# 2 pLearn Dependencies

Prior to using the pLearn Learning platform, the following dependencies must be installed.

## 2.1 MAC OS X and Ubuntu

This platform 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.python.org/downloads/release/python-2714/

After Python has been successfully installed, run the following commands in order to install the python dependencies:

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

The platform also requires a valid `moos-ivp` and `moos-ivp-aquaticus` tree for the relevant utilities. The `moos-ivp` tree can be acquired by running:

```
$ svn co svn+ssh://{username}@oceanai.mit.edu/home/svn/repos/moos-ivp
$ svn co svn+ssh://{username}@oceanai.mit.edu/home/svn/repos/moos-ivp-aquaticus/trunk aquaticus
```

Make sure that the following directories are included in your `PATH` environment variable:

```
{path to}/moos-ivp/bin
{path to}/moos-ivp-argupta/bin
{path to}/tensorflow/bin
{path to}/aquaticus/bin
```

Include the following directories in your `IVP_BEHAVIOR_DIRS` environment variable:

```
{path to}/moos-ivp-argupta/lib
{path to}/aquaticus/lib
```

Finally, include the following directories in your `PYTHONPATH` environment variable:

```
{path to}/moos-ivp-argupta/pLearn/learning_code
{path to}/moos-ivp-argupta/src/lib_python
.
```

# 3 pLearn QuickStart

This section is meant to see that everything is setup and working correctly. By the end of the section, you will load an already trained model into BHV Input and run a simulation with it. These files are all in the `moos-ivp-argupta/pLearn/learning_code` folder

1. **Modify the File Paths -** Change the `user_path` variable in `Constants.py` to the path to your moos-ivp-argupta tree.

2. **Setup Files to Load the Model -** We will load the model in the

   `moos-ivp-argupta/pLearn/learning_code/examples/Simple_Opponent_BHV/topModel`

   folder. First, overwrite *Constants.py* with

   `moos-ivp-argupta/pLearn/examples/Simple_Opponent_BHV/environment.py`

   which is a snapshot of the `Constants.py` file used when the `Simple_Opponent_BHV` Model was trained. Then, reset the `self.load_model_dir` variable in the new `Constants.py` to be:

   ```
   self.load_model_dir = user_path +
           "moos-ivp-argupta/pLearn/examples/Simple_Opponent_BHV/topModel"
   ```

   to tell the program that that is the model we are loading.

3. **Load and Run the Model -** On the command line, run:

   ```
   $ python reinforce.py test
   $ ./run.sh
   ```

   to load the model to be tested, and then run a simulation with Felix exhibiting the learned behavior.

# 4 Training a Model Using pLearn

## 4.1 Training a New Model

1. **Setup Constants.py -** Set the desired parameters in `Constants.py`, and more importantly, set `Constants.save_model_dir` to a new directory name in the `model/` folder. More specific information about `Constants.py` is in the **Making a Model** section.

2. **Begin Training -** run:

```
$ python reinforce.py new
```

   to start training a new model based on the parameters sepcified in `Constants.py`.

## 4.2 Continuing to Train an Existing Model

1. **Overwrite Constants.py -** Go into the folder where the model is located and copy the `environment.py` file to `Constants.py` in the `learning_code/` directory (overwrite the `Constants.py` file in the process). This ensures that the settings correspond with the old model.

2. **Load the Base Model -** set `Constants.load_model_dir` to the name of the directory that holds the existing model.

3. **Set the Save Directory -** set `Constants.save_model_dir` to the name of the directory you would like to save to.

4. **Begin Training -** run:

```
$ python reinforce.py load
```

   to continue training the model that is loaded. The model will still save into the directory specified by `Constants.save_dir` and will not overwrite the load directory unless specified.

# 5 Loading and Testing Existing Models

## 5.1 Loading One Specific Model

To load and test one specific model, specify the directory path to that model in `Constants.py` by setting `Constants.test_address` to the directory of the new model, then call:

```
$ python reinforce.py test
$ ./run.sh
```

to output an optimal table for that action, and then run a simulation using the loaded model.

### 5.2 Testing Multiple Models

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 that you want to test in the directory of `Constants.test_address`. The program will run `Constants.num_test_iters` simulations for each model and get data. At the end, it will output the model directories with the top 15 average rewards and then graph the average reward per iteration and percent time spent inbounds.

## 6 pLearn File Structure

### 6.1 Directory Structure

**Learning_Code/** top level directory which holds the algorithms, data and models to do reinforcement learning.

**Constants.py** configuration file that holds the state definition, file paths and hyperparameters for learning.

**DeepLearn.py** the implementation of the reinforcement learning class and training algorithms.

**Reinforce.py** makes the action space and the reward function. It calls DeepLearn.py to create, load, and train models.

**Clean.sh** cleans the simulation engine folder to remove extraneous logs.

**results/** a folder containing raw data from simulations.

**log_converter.py** converts the logs in `results` to better formatted .csv files and places them into `processed`.

**run.sh** runs a simulation for humans to view to check on model progress.

**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 the simulation.

**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 that were in `Constants.py` when the model was initialized.

**train_and_evaluate** a script to first train a model and then run tests on it.

**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 `pMarineViewer`).

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

**m200/** m200 vehicle configuration files.

**BHV_Input_output.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.

**mokai/** mokai vehicle configuration files.

**motorboat/** motorboat vehicle configuration files.

**shoreside/** shoreside configuration files.

### 6.2 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 file is where all major program parameters can be set and will be carried through to the other learning scripts.

**Reinforce.py** - This is the main piece of code that is called to build and train the learning agent. It defines the statespace, actionspace, and the reward function to be passed to the learning agent, and then initializes and trains the agent based on the parameters given in `Constants.py`. Call `Reinforce.py` 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 for testing.

**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`. `DeepLearn.py` outputs `table.csv` to the `reinforcement/learning_code` folder to be consumed by `BHV_Input`.

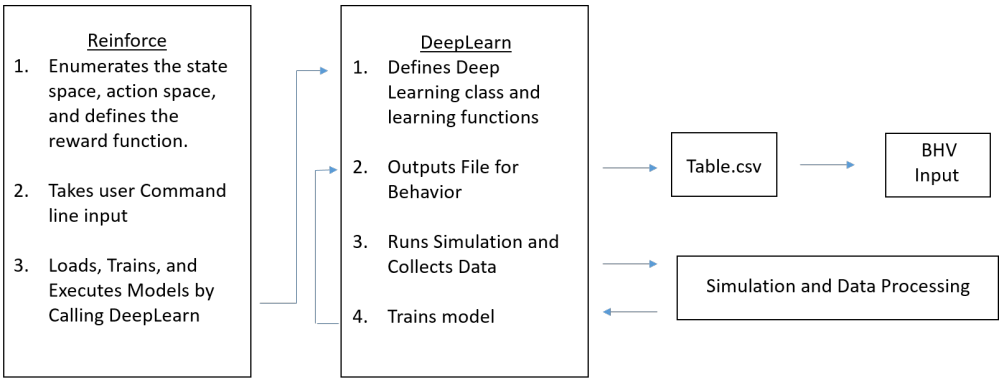


Figure 1: **Learning Script File Interaction Diagram**

## 6.3 Scripts for Running Simulations

**Tester.sh** - Runs the simulation for a set amount of time and then kills it. It then extracts notifications from `INP_STAT` in the relevant `MOOSLogs` to get raw state and action information and writes the resulting `.csv` to the `results` directory.

**launch_reinforce.sh** - Launches the simulation with Felix on the red team exhibiting the learned. behavior.

**run.sh** - A simple script to run a simulation with the current model and `pMarineViewer` running.

## 6.4 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).

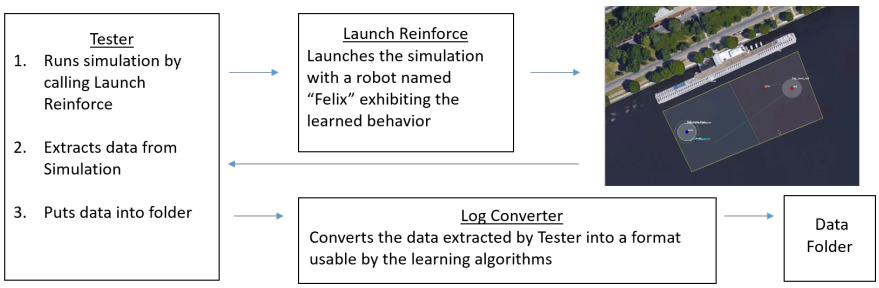


Figure 2: **Simulation and Data Processing File Interaction Diagram**

.

# 7 Types of Learning Algorithms Available

## 7.1 Fitted Q Learning

The algorithm makes one neural net per action, each of 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. The mechanics of building the neural neat and training it are provided by the `Keras` Python library.

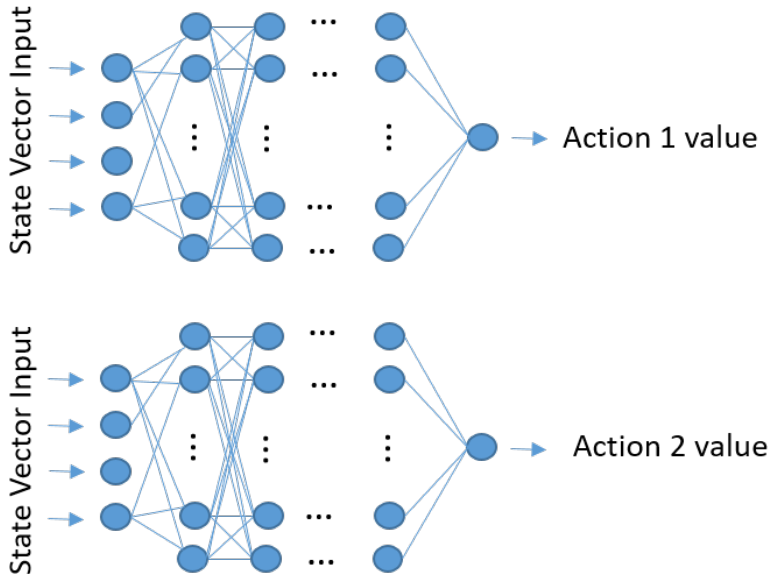


Figure 3: **Fitted Neural Net Structure:** Seperate neural nets for each action. They all take in the same state vector as input, and we select the action which has the neural net with the highest output for that state vector.

*Listing 7.1: Basic Algorithm for Fitted Q Learning.*

```
1  ================================================================
2  Fitted Q Learning Algorithm PseudoCode
3  ================================================================
4
5  initialize model_NN_dict
6  initialize target_NN_dict
7
9  //setup the neural nets
8  for action in actionspace:
9       model_NN_dict[action] <- new neural net with 1 output node
11      target_NN_dict[action] <- new neural net with 1 output node
12
13 //get data and train neural nets
14 for iters <- 0 ; iters < total_iterations:
15      data <- data from running a simulation with model
```

```
16      store data into memory
17      training_set <- Random sample of experiences from memory
18
19      for experience in training_set:
20          expected <- Caculated expected reward for experience based on target_NN
16          //seperate experience into its state, action, nextstate, reward pairs
17          (state, action, newstate, reward) <- experience
18          train model_NN_dict[action] on state input to target output "expected"
22
23      //update target network
24      target_NN_dict <- model_NN_dict
```

## 7.2 Deep Q Learning (DQN)

The algorithm makes one neural net that takes in a state representation as input and outputs 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.

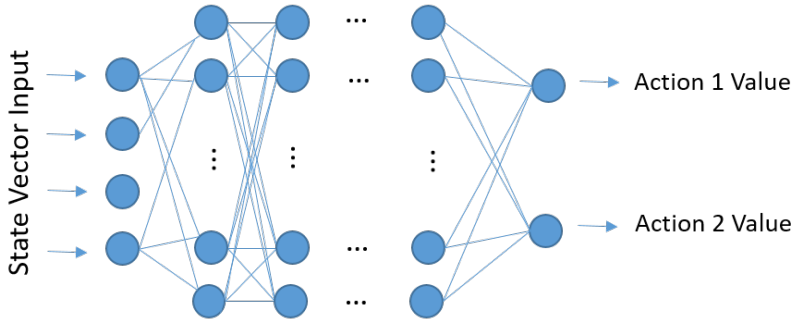


Figure 4: **Deep Q Neural Net Structure:** One neural net for all the actions, with one output per action. Takes in a state vector as input, and we select the action which has the highest output from the neural net.

*Listing 7.2: Basic Algorithm for Deep Q Learning.*

```
1  ================================================================
2  Deep Q Learning Algorithm PseudoCode
3  ================================================================
4
5  initialize model_NN <- new neural net with 1 output node per action
6  initialize target_NN <- new nueral net with 1 output node per action
7
8  //get data and train neural nets
9  for iters <- 0 ; iters < total_iterations:
```

```
    data <- data from running a simulation with model
    store data into memory
    training_set <- Random sample of experiences from memory

    for experience in training_set:
        expected <- Caculated expected reward for experience based on target_NN
        //seperate experience into its state, action, nextstate, reward pairs
        (state, action, newstate, reward) <- experience
        train model_NN on state input to target output "expected"

    //update target network
    target_NN <- model_NN
```

# 8 Making A Model

Change the `user_path` variable to the path to your moos-ivp-argupta tree. 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.

## 8.1 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`, change the `user_path` variable to the path to your `moos-ivp-argupta` tree. `Constants.py` has four main sections listed below.

### 8.1.1 State Definition

The state class is a way of holding all the state information together. It has the fields:

**Index** location of this parameter in the input vector.

**Type** the type of information the state contains. This can be either `"binary"`, `"distance"`, `"angle"`, or `"raw"`. `"Binary"` type states can only be 0 or 1. `"Distance"` type states represent the distance between `vehicle` (defined below) and the robot running the learned behavior. `"Angle"` type states represent the angle betwwn `vehicle` and the robot running the learned behavior. Finally, `"raw"` type states represent raw values. These specifications are used by `BHV_Input` to insert the correct type of value into the state vector.

**Normalized** A boolean flag of whether or not to scale the variable to be between 0 and 1 (based on the range 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 machine learning model.
- **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.

**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 vector 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.

### 8.1.2 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 capable of approximating the Q-value accurately, as well as deciding how the neural net is trained.

**num_layers** The number of layers in the neural network.

**num_units** number of units per layer in network.

**num_traj** number of times a simulation is run per iteration to make dataset.

**iters** number of iterations for training (can be arbitrarily high since model is saved on each iteration and training can be stopped any time).

**lr** learning rate for the Adam Optimizer used to train the neural net.

**training_type** whether trained with `"stochastic"` gradient descent or `"batch"` gradient descent.

**eps_min** the minimum rate at which actions are picked at random (exploration).

**eps_init** the starting rate ar which actions are picked at random.

**eps_decay** the rate at which the randomness of actions diminishes per iteration.

**epochs** how many times the the optimizer is run on a batch of training data.

**batch_size** the number of examples randomly picked from memory when training.

**batches_per_training** the number of times we pick out and train on batch of size `batch_size` per iteration.

**alg_type** selects which algorithm to use, can be either `"DQL"` or `"fitted"`.

### 8.1.3 pLearn 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.

**speeds** possible speeds that the action can have.

**relative** switch to actions that add or subtract from heading rather than set an absolute heading to take. In practice, absolute headings tend to work better.

**rel_headings** possible headings for relative actions (only relevant if `relative` is `true`).

**thets_size_act** the block size for how spaced out the possible thetas are in the action space.

**discount_factor** the rate at which future rewards are discounted (each timestep ahead that a reward is, it gets multiplied by the discount factor).

**max_reward** the maximum reward possible at the goal state.

**neg_reward** the negative reward used as punishment for bad actions (such as going ouut of bounds or getting tagged).

**reward_dropoff** the rate at which reward diminishes as you move away from the goal.

**max_reward_radius** the radius around the goal that has `max_reward`.

**save_iteration** boolean for whether to have the model automatically save checkpoints every `save_iter_num` iterations.

**save_iter_num** how often the model is saved.

**players** a list of the other players in the simulation. Allows `BHV_Input` to keep track of them.

**mem_type** whether the memory is a `"set"`, a `"deque"`, or a `"memory per action"`. A set makes an unbounded memory with no repeating experiences. A deque makes a memory of fixed length (`self.mem_length`) that may have multiple of the same experiences, but keeps track of more recent experiences. Memory per action makes a seperate deque for each action with fixed length (`self.mem_length`), and samples are taken equally from each action deque.

**mem_length** the length of the memory if using `"deque"` memory type.

**mem_thresh** the threshold at which the program goes from sampling one batch per iteration to sampling `batches_per_training` batches per iteration.

**end_at_tagged** boolean that dictates whether tagged states count as terminal states (i.e. there are no transitions recorded that go from a tagged state to another state).

**num_test_iters** The number of simulations the test script runs per model in the testing directory.

#### 8.1.4 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 the `Constants.py` file.

**sim_cmd** path to the simulation running command.

**process_path** path to folder to get the data to be processed.

**process_cmd** path to script to process the raw data at `process_path`.

**read_path** path to the processed data folder.

**out_address** path to where `table.csv` should be output so that it can be read by `BHV_Input`. This should be the path to the folder containing the `.bhv` file running `BHV_input`.

**load_model_dir** path to the folder where an existing model should be loaded from.

**save_model_dir** path the folder where we should save the model that is being trained.

**mem_address** path to the folder where a `memory.txt` file is so that we can load it into the program. Make sure that the memory file has the same state and action definitions as the current model. If no such file, leave the address as an empty string.

**test_address** path to the folder holding the models that need to be tested (with sub folders labeled `iteration_#`).

### 8.2 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.

### 8.3 Modifying the Simulation Scripts

#### 8.3.1 Changing the Time For the Simulation

In `Tester.sh`, edit the `while` loop to change the counts for how many times the while loop sleeps before killing the simulation. To change the time warp edit the number next to the `./launch_reinforce.sh` command.

### 8.3.2 Editing the Simulation

Edit files in the `simulation_engine/` folder as needed to edit the nature of the simulation. To edit the simulation run by the training scripts, edit `launch_reinforce.sh`. To add more players to the game, add the appropriate m200 `launch_*` script and set a robot name not yet selected and the team that you would like the robot to be on. To remove a player, just delete or comment out the line with its launch script in `launch_reinforce.sh`.