101 | 202 | 480

- Instructor Info
- Projects
- Tile Driver I
- Tile Driver II
- Biogimmickry
- Mine Shafted
- Board Stupid
- Moonlander II
- Course Details
- Description
- Assessment
- Resources
- Calendar
- Policies
- Enrollment
- Code Style
- Student Conduct
- Accommodation
- Guides
- Casey Submission System
- Getting Started
- Secure Shell (SSH)
- Vim Editor

Project VI: Moonlander II - Crash & Learn

Due Wed, Aug 18 by 9PM Pacific Time

To begin, use the following file: http://users.csc.calpoly.edu/~dkauffma/480/moonlander.py

- To download locally, right-click link and choose "Save link as...".
- To download and unzip on a CSL server run:

```
wget http://users.csc.calpoly.edu/~dkauffma/480/moonlander.py
unzip moonlander.py
```

Description

This project is based off "Moonlander", assigned in Cal Poly's CPE 101.

Eagle ir
photographe

You are tasked with training an agent in a simulation to land the Lunar Module (LM) from the Apollo space program on the moon. The simulation starts when the retro-rockets cut off, using a predetermined amount of fuel and altitude (with an initial velocity of zero meters per second). With the thrusters off and the LM in free-fall, lunar gravity is causing the LM to accelerate toward the surface. The agent must control the rate of descent using the thrusters of the LM by selecting a rate of fuel flow each second. A fuel rate of 0% means free-fall, 50% maintains the current velocity, and 100% means maximum thrust. To make things interesting (and to reflect reality) the LM has a limited amount of fuel. If your agent runs out of fuel before touching down, it of the LM, which then free-falls to the surface. The goal is to land the LM on the surface with a than `-1` meters per second, using the least amount of fuel possible. Note that a negative ve movement toward the moon, while a positive velocity indicates movement away from it.

While this simulation is inspired from the moon landing, you may simulate landing on other celestia One of the simulator initialization values is the g-force applied to the module. The following table force at (or near) the surface of various objects.

Object	G-force
Pluto	0.063
Moon	0.1657
Mars	0.378
Venus	0.905
Earth	1.0
Jupiter	2.528

Q-Learning

Reinforcement learning is an iterative process using a system of rewards to develop a policy tha action to apply in any given state. That is, it approximates a function `π(S) -> A` that maps s optimal) actions to apply in those states. Each state in the environment has a (usually unknown) ut approximated throughout the learning process, so that ultimately actions can be chosen that lea higher utilities. The utility of a state is based on predefined rewards received in that state ar rewards of successor states.

As with problems discussed previously, learning environments use a transition model `T(S, A) -` a state-action pair to a successor state. However, many environments exist in which this model is to the learning process. A well-established approach to [model-free]() reinforcement learning is [Q-]() observes transitions from exploring the environment to approximate a function `Q(S, A) -> U` t action pairs to the estimated utility of that successor state. With this Q-function, the policy func equivalent to $\pi(S) = \text{argmax}_A\ Q(S, A)$, which means return the action `A` that results in th returned from `Q(S, A)`.

While a more precise approximation of the Q-function would use a sophisticated model like a neu small environments a Q-table (matrix) indexed by state (row-wise) and action (column-wise) to a be sufficient and is much faster to train and easier to implement.

During the training phase, table updates are performed using the following formula,

$$(1 - \alpha) * Q(s, a) + \alpha(R(s) + \gamma * \max_{a'}\{Q(s', a')\})$$

where α is the learning rate and γ is the discount factor, both determined using experimen starts high and decreases throughout the learning process). `R` is the reward function, which is problem. Note that since every call to the `Q` function is simply a table look up, this formul [memoization]() rather than recursion.

Reward Function

Every reinforcement learning process involves a reward function, which specifies the immediate va particular state. These functions may be very simple or contain a complex sequence of condition the environment. In the context of the LM, a reward function would need, at a minimum, to detect state is reached (i.e. when the altitude is zero) and provide a value based on the LM's velocit landings are more highly rewarded. In addition, the LM should be encouraged to move toward (mainly so it does not try to hover in place until fuel runs out), which can be done in various ways t using the altitude, fuel, as well as imposing a small penalty for being in a non-terminal sta appropriate reward function is one of the many challenges of implementing a reinforcement learnir

ε-Greedy Selection

During the training phase of reinforcement learning, actions must be selected to produce succ would seem natural to select actions greedily by choosing the action in a given state that, acco function, results in the successor state with the highest utility. This greedy approach is **ex** knowledge built thus far, but since the utilities during training are still changing, it is possible th action sequences exist. A simple way to encourage **exploration** is to allow a small ε chance of ch at random instead of the current best one. This chance may start higher at the beginning of trainin decrease over time, which has the effect of encouraging a lot of exploration early on and reinforcin action sequences near the end.

Implementation

Allowed Modules: random

The provided starter file contains the `ModuleState` class to represent states in the simulation. Th initialized with a starting fuel, altitude, velocity (which is always zero for the first state), the g-forc module, and a transition function. Successor `ModuleState` objects are generated by calling method, which takes a fuel rate as its argument and uses the transition function to calculat acceleration, which in turn impacts the altitude and velocity of the successor state. The transition any callable that accepts two floats: the acceleration due to gravity (in m/s^2) and the fuel rate s reasonable transition function is provided in the file, you may use any (unrealistic) function with t your tests (e.g. a function that just returns a constant value).

At each state, the module can use one of its available fuel rates to move to a successor state by to `use_fuel`. For example, a module with `5` available rates would have the following corresponding amounts of thrust.

Action	Fuel Rate
0	0%
1	25%
2	50%
3	75%
4	100%

Note that larger action sets offer more control (higher sensitivity to differences in rate changes), b Q-tables and thus more training time.

```
learn_q(state: ModuleState) -> Callable[[ModuleState, int], float]
```

Return a Q-function that maps a state-action pair to a utility value. This function must be a c signature shown. When this Q-function is used by an agent to make a policy, the lander must a target surface with an impact velocity greater than `-1`. Assume that the g-force and transition fu to the given `ModuleState` object will be the same one used to test the Q-function generated (bu assumption about the other initial values). For both the training and testing phases, assume t altitude is never above `100` meters.

The function returned may be a closure as in the following example:

```
q = lambda s, a: table.get((s, a))
```

This example assumes a variable named `table` exists and its value has a `get` method. Your may be different so long as it returns a function object with the signature `(ModuleState, int)` simplicity, it is recommended that this function use a table lookup (i.e. a dictionary) to map state utilities. Care must be taken to appropriately discretize the state representations (which may inclu current fuel, altitude, and velocity) so that the table does not become too large.

The action set may be changed from the default of `(0, 1, 2, 3, 4)` by calling the `set_actions` `ModuleState` class. Doing so is optional, but must only be done once before the training process

```
state.set_actions(8)  # sets the action set to be (0, 1, 2, 3, 4, 5, 6, 7), with
```

Changing the action set can be useful for implementations that seem to benefit from more or le the decision-making process.

Submission

On a CSL server with `moonlander.py` in your current directory:

Instructor	Command
Daniel Kauffman	`/home/dkauffma/casey 480 moonlander`