Udacity Machine Learning Nanodegree

Capstone Proposal - Mateusz Bednarski

Domain Background

For my capstone project, I have selected robot motion planning. This requires either a physical robot or simulation engine. For purposes of this project I decided do make use of OpenAI Gym - https://gym.openai.com/ . Gym is a set of environments simulating various tasks. It provides ready to use simulators and framework for comparing algorithms. From available environments, I selected "LunarLander-v2" - https://gym.openai.com/envs/LunarLander-v2. For a few reasons:

- For first, reinforcement learning interested me the most, so I want to get deeper into this.
- I love space and landing on a planet/moon is interesting problem for me.
- It has (approximately – more details in dataset section) continuous state space, so basic tabular Q-learning will not work – I need to examine more sophisticated techniques

I want to explore approximating Q-values using a function. At this moment I think about Logistic Regression, SDG, simple neural net – but they need investigation, which one (if any) will work.

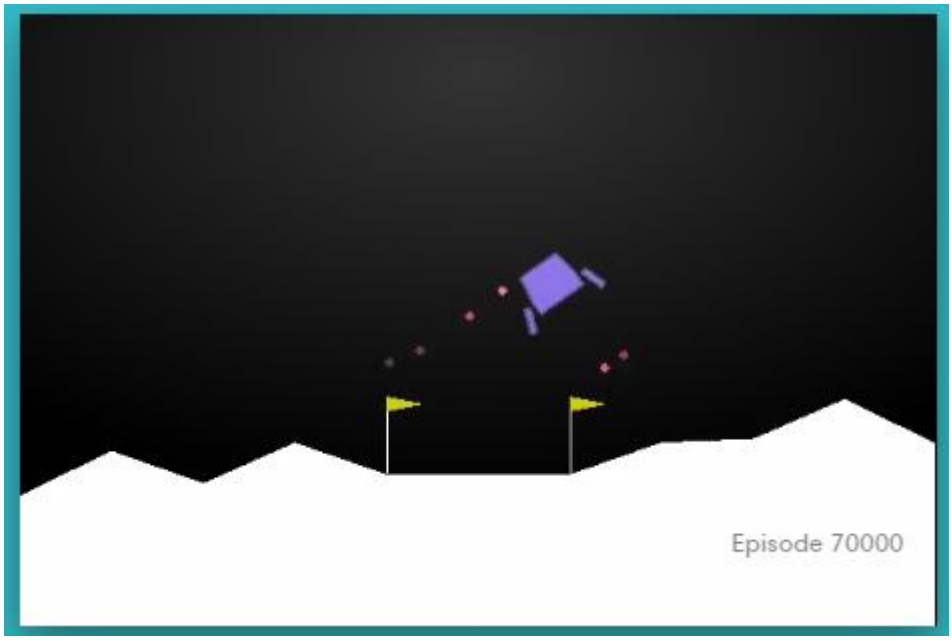

Figure 1 LunarLander-v2

Problem statement

There are a landing pad and a lander. Lander has two legs, and three engines – thrust directed down, left and right. Possible actions are one of do nothing/fire main engine/fire left engine/fire right engine. Task is to land on the landing pad. Lander starts above the pad and is affected by gravity. Simulation is finished either lander lands or crashes. Landing outside pad is also possible. Fuel is unlimited and there is no time penalty.

Rewards and penalties are already defined in the environment:

- Moving towards pad with zero speed: from +100 to +140
- Crash: -100
- Successful landing: +100
- Leg with ground contact: +10
- Firing main engine (not side engine): -0.3
- When landing outside pad, an additional penalty is given.

Datasets & input

Because it is reinforcement learning problem, there is no dataset understood in a classic way. Data available for the algorithm are as follows:

Action space is discrete, finite set:

$$A = \{0,1,2,3\}$$
$$|A| = 4$$

State space is a vector of 8 real numbers describing lander position, velocity and orientation.

$$s \in \mathbb{R}^8$$

To be specific there are not real numbers, because of limited floating point precision. So technically it is finite set. However it is very big:

$$|S| = (2^{52})^8 \approx 1.69 \times 10^{125}$$

And Q-space for that would be

$$|Q| = |S| \times |A| \approx 6.77 \times 10^{125}$$

If I would use tabular Q-learning (as in course) full table would take about 2.7×10^{114} TB of memory. For this problem, we can safely assume, that S-space (and consistently Q-space) is infinitely, uncountable (because of real, not integer numbers) large.

Solution statement

Because Q-space ($S \times A$) is infinite in size, it is impossible to use Q lookup table. Instead, I will use $h(s, a) \rightarrow R$ function to approximate Q(s, a) values. I hope, linear model will be sufficient.

The idea is to replace Q table with a function, that will approximate Q(s, a) values. Therefore, the goal is to find a function, that will return a value for a given pair state (s,a).

$$Q(s, a) = \sum_{i}^{n} f_i(s, a) w_i$$

There is a need to describe f_i. As we can see, Q(s,a) is a linear combination of n f_i functions. Each f_i function can represent a different feature. Feature in the most simple cases, those can be just observations. For example, if Q-space is defined as above we could create following f_i functions:

$$f_1(s,a) = s_1$$

$$f_2(s,a) = s_2$$

$$f_3(s,a) = s_3$$

$$\vdots$$

$$f_8(s,a) = s_1$$

$$f_9(s,a) = a_1$$

$$\vdots$$

$$f_{12}(s,a) = a_4$$

Therefore, update rule will be following:

$$err = \big(R(s,a,s') + \gamma V(s',a')\big) - Q(s,a)$$

$$w_i \leftarrow w_i + \alpha[err] f_i(s,a)$$

Where V is defined as:

$$V(s',a') = \max_{a'} Q(s',a')$$

After each step, we are updating weights in function approximating values.

Another possibility is to use 4 separate Q(s,a) functions. In our example that would be separate Q_a for each action:

$$Q(s,a) = Q_a(s)$$

$$Q_1(s) = \sum_i^n f_i(s) w_i$$

$$Q_2(s) = \sum_i^n f_i(s) w_i$$

$$\vdots$$

This can be useful because each action have a separate model and they do not interact with each other. I suppose linear model might be not sufficient to handle relations between all 4 possible actions. From the other hand, non-linear model will be more prone to overfitting.

This approach allows to:

- Handle near-continuous Q-space
- Generalize knowledge to unvisited states
- Provides robustness – similar Q-states will have similar Q-values

Evaluation metric

Environment's author define it as solved when the average cumulative reward for 100 following trials is over 200. I am not sure how difficult it is, so I do not aim to achieve 200 average reward. Instead, I will consider solution as acceptable, when the lander will land in consecutive 100 trials. During training, I will use moving average of cumulative reward for last 100 trials – just as OpenAI uses.

Benchmark model

Gym provides detailed information about submitted solutions (https://gym.openai.com/envs/LunarLander-v2). I will use https://gym.openai.com/evaluations/eval_Dsaitj0TdW7jejz3ePwAQ as a benchmark for my model, but comparing with other ones also can be valuable to determine quality of my solution. I did not chose top solutions because it uses deep Q-learning which I am not going to explore right now.

Project design

Project will be developed in Python 2.7 with gym library. For machine learning, I will use scikit-learn and maybe Keras if will be needed. I have specified a few steps.

For first, I want to implement simple linear function approximation and some code to grab statistics about learning and performance in order to easily compare methods.

I want to try few methods in order to check which will work. Currently I have:

- Linear function for approximation
- Stochastic Gradient Descent
- Passive-Aggressive regression
- Simple (one hidden layer) neural network
- More sophisticated neural networks (two or more layers)
- Softmax exploration/exploitation
- Various decay functions
- Normalizing features

SDG and Passive-Aggressive regression implemented in scikit-learn are able to train online (using `partial_fit` method). I am going to update model (probably four model –one per action) each step or in chunks. If I will explore neural network, the structure I want to create, is to have |S| input neurons and |A| output neurons with values in range [0;1]. Using softmax instead random choose from max Q-values can help

with exploration/exploitation issue (or not – need to try out). As S-space elements can vary in values, normalizing them before can potentially make learning faster and more reliable.

All of those methods have parameters to tune (alpha, epsilon, temperature, etc) so It will be needed to try them out. Statistics of all runs will be collected automatically in order to have full overview of various combinations. Also, I will store (s,a,s',r) tuples. In this moment I do not know how to use them, but I may find application to train some kind of supervised model on that (maybe learning T and R).