Machine Learning Engineer Nanodegree
Capstone Proposal
Jared Wadsworth
September 21st, 2017

Proposal

Domain Background:

Almost since the dawn of Machine Learning and Artificial Intelligence researchers and scientists have been trying to teach machines through Reinforcement Learning (RL). This field of AI was originally inspired by Behaviorist Psychology. Specifically, it deals with how an agent chooses actions in an environment to maximize some reward. Generally, this problem is captured as a Markov Decision Process which uses a (state, action reward, next_action) tuple. Historically researchers have often used RL to play games such as Chess (https://arxiv.org/abs/1509.01549), Go (https://gogameguru.com/i/2016/03/deepmind-mastering-go.pdf), or even various Atari games (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf), however some more concrete examples of some applications of RL include: Self Driving Cars, and investing in the Stock Market.

With the advancements made in Deep Learning there has been a renewed interest in RL in what has been called Deep RL. Using these techniques computers have been able to beat the Grandmasters of Chess and Go, and the best gamers at the Atari. It has also made self-driving cars a reality.

On a personal note, ever since I first learned about reinforcement learning it has always peaked my interest. To me it is the most fun side of AI. I also believe that it has the greatest potential to change our lives, and help us solve many of the problems of today.

Problem Statement:

https://gym.openai.com/envs/LunarLander-v2/

The problem which I will solve is a game called Lunar Lander. The basic idea of this game is that the lunar lander is coming in for landing, and the agent gets to control its thrusters taking one of four possible actions (left, right, main, none). The agent must land the lander in the specified location on the ground without crashing the lander. To help the agent to land without crashing, it will get certain inputs from the environment, namely its location, the orientation of the lander, and its velocity. It will also recieve rewards to know how well it is doing. Rewards for the agent include landing softly in the specified location (100 - 140 points), crashing (-100 points), landing softly (100 points), Leg ground contact (10 points each), Firing Main engine (-0.3 points per frame). The agent wins if it scores over 200 points.

Datasets and Inputs:

At each step, or frame, of the game the agent receives an array with values corresponding to its location along both the x and y axis, orientation of the lander (is the lander facing directly towards the ground?), and velocity (the speed and direction of the lander). Combined these are referred to as the "State". Each of these is necessary to allow the agent to navigate down to the landing zone. These inputs come from the OpenAI Gym environment.

At each frame the agent will take an action, and then receive a reward from the environment (as described above). Using this (state, action, reward, next_state) tuple, the agent is able to determine how effective each action was, and then optimize itself to always choose the best action each frame so that it is able to land softly in the landing zone.

Solution Statement:

There are currently many solutions to this problem, including Q Learning, which I will describe in the next section, and Deep Reinforcement Learning, which will also be described in the following sections.

Benchmark Model:

A common solution to this type of problem is Q Learning. This however doesn't work because of the continuous values used in the state array it becomes impossible to build a Q table. One solution to this problem is to discretize the state array. This can be done in many ways. A simple solution would be to round to the nearest whole number. Another solution would be to simply split the state space into 'n' buckets (where n can be any positive integer) and assign each bucket a value and use that value. By discretizing the state array the agent is able to build a proper Q table and learn a valid policy. This then can be compared to my solution through the metrics described below.

Evaluation Metrics:

An agent is said to have solved this problem if it is able to score an average of greater than 200 points over a period of 100 iterations (games).

The metric of success will be the fewest number of iterations (games) until the algorithm solves the problem. For example, if it takes algorithm A 10,000 iterations to solve it, and it takes algorithm B 9,000 iterations, then algorithm B is said to be better than algorithm A.

Project Design:

My plan is to start by coding up a simple Q Learner (described in the benchmark session) to use as my benchmark for further testing. It will use the 'bucket' method for discretizing the states, and load them into a Q table to learn an optimal policy. After building the benchmark, I will start exploring various other options to see if I can beat it.

I'll then dive into some alternate algorithms such as Deep Q Networks (DQN) which uses a deep neural network instead of a Q table. The idea is that since the state space is so large (nearly infinite since it is continuous) rather than discretizing we can build a deep neural network that will predict the Q table values rather than storing them. Next, I'll try Actor-Critic (AC) which employs two deep neural networks where one learns how well the other is actually modeling the environment and helps it to more quickly move towards correct answers. I'll also keep researching and see what other algorithms people are using and may try some of those. After creating these agents, I'll compare them to the benchmark algorithm as described in the metrics section. After testing each of them, I will select the best one and move on to the next step.

Since the agents receive rewards after each action. The Q table (or network) will be updated after each action. This, however makes it difficult to learn because only the last action will receive the rewards for landing correctly (or crashing). To help solve this, I will keep a 'memory' or previous actions, and after each action is taken I will replay these memories to the Q table / network. Since each entry is updated by the function reward = (reward + discount_factor * reward_of_next_state). That means that over time the Q values in the table / network will converge so that it will take the action that will maximize the reward gained immediately, and the discounted reward for the future.

After choosing an algorithm, I'll continue by altering the things the algorithm learns on. I'll potentially introduce some artificial rewards, or alter the rewards before the agent learns on them. Then I might change some of the items in the state to see if I can get it to learn better, each time comparing it to the previously selected agent.

Finally I'll run the new agent, and compare it to the benchmark to see how much better it performed. I will then submit my findings, and code to both Udacity and OpenAI.