Capstone Project

Machine Learning Engineer Nanodegree

Jou-ching Sung

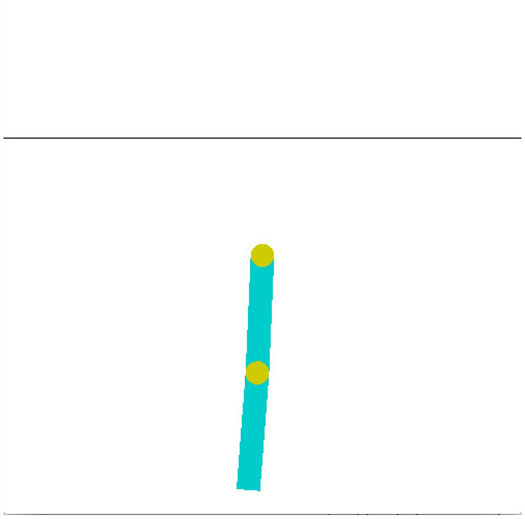
September 20, 2016

I. Definition

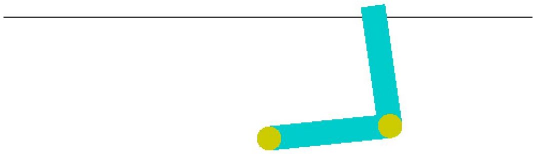
Project Overview

This project attempts to solve the OpenAI Gym "Acrobot-v1" environment (https://gym.openai.com/envs/Acrobot-v1), using deep reinforcement learning techniques. OpenAI Gym's description of the environment states: "The acrobot system includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height." The Acrobot problem is analogous to a gymnast swinging on a high bar.

Visually, the Acrobot environment starts in a state like the following:



Applying clockwise, counterclockwise, or no torque to the joint between the two links, the goal is to swing the lower link above the threshold, as such:



A video of how this may be done can be found here:
https://gym.openai.com/evaluations/eval_X4wWctnvRvqcVHy4FLDRwQ

Acrobot is a classic reinforcement learning problem, and a more detailed description of Acrobot can be found at https://webdocs.cs.ualberta.ca/~sutton/book/ebook/node110.html

Problem Statement

The problem is to solve OpenAI Gym's "Acrobot-v1" environment. The goal is to train an agent to swing the lower link above a given height, in as few time steps as possible.

Given recent advances in deep learning applied to reinforcement learning, I have chosen a deep reinforcement learning strategy to solve this problem. Using deep reinforcement learning techniques on Acrobot will allow me to run experiments quickly (as opposed to running experiments on Atari Pong, for example), which allows for greater exploration in terms of different techniques and parameter combinations.

Metrics

In Acrobot, the goal is to swing the lower link above a certain height in as few timesteps as possible. In the Acrobot-v1 environment, a reward of -1 is given at each timestep, until the agent achieves the goal. Thus we would like to maximize the total reward, i.e. the score, of the agent.

Currently, OpenAI Gym does not specify a threshold where Acrobot-v1 is considered solved. To get a baseline, I ran 100 consecutive episodes of Acrobot by uniformly sampling from the action space. By taking completely random actions, the average score over 100 episodes is -1987.17. The code to perform this is available at ‘random_avg_score.py’.

Given this baseline, **I have chosen a threshold score of at least -100 over 100 consecutive episodes, for Acrobot to be considered “solved”.** This is almost 20x better than the aforementioned baseline, and achieving this score would put us in the top half of the submitted scores on OpenAI Gym Acrobot-v1: https://gym.openai.com/envs/Acrobot-v1

II. Analysis

Data Exploration

In reinforcement learning problems, the input space for the learning agent consists of observations and rewards from the environment. The agent’s outputs are the actions at each timestep. Thus, our goal is to explore the state/observation space and action space, and explore how the rewards are given.

To explore the Acrobot-v1 environment, I ran one episode by choosing a random action at each timestep, until the episode completed. This code is available at ‘explore1.py’. I printed the state, action, and reward at each timestep. The last few lines of my print-out are as follows:

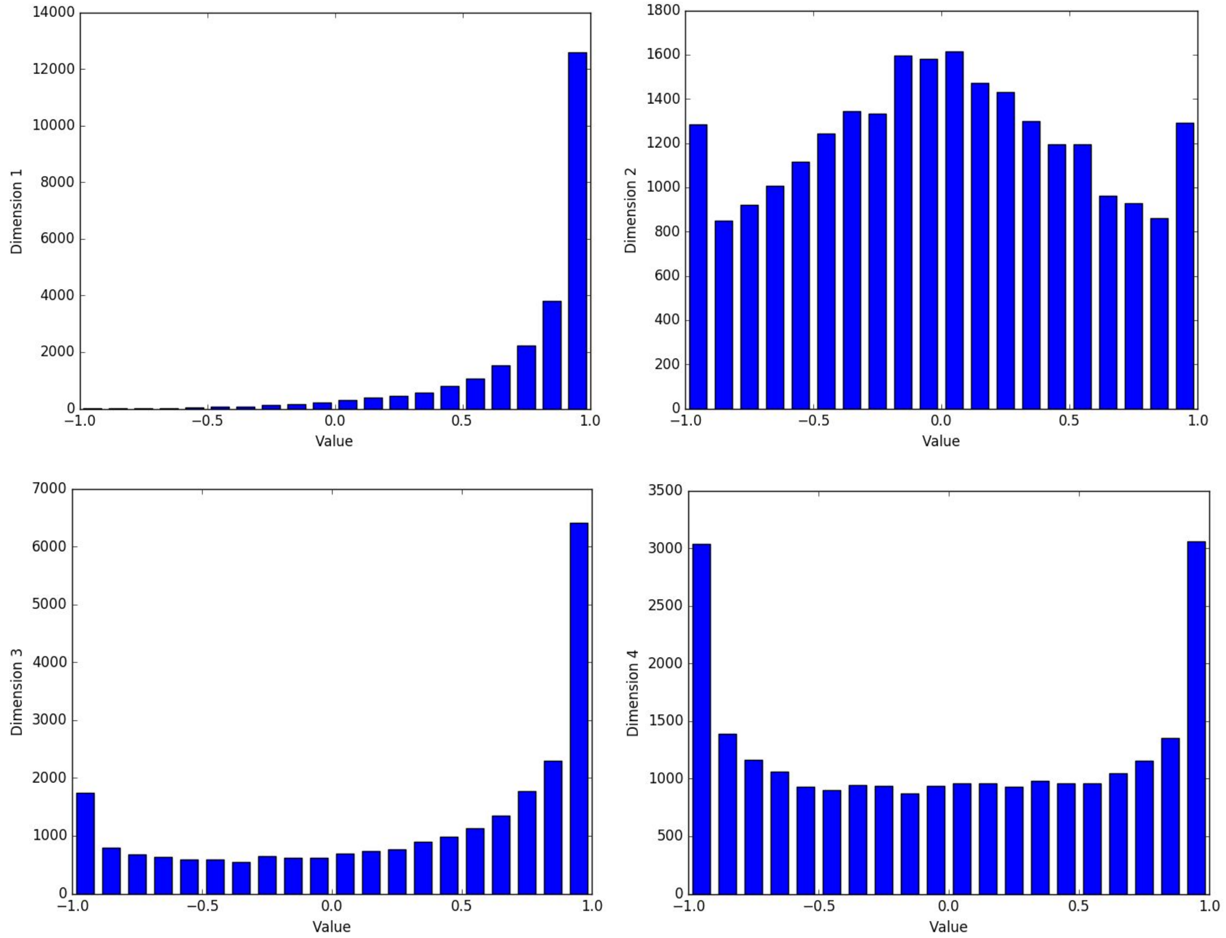
```
obs: [-0.4773868   0.87869326 -0.99994922 -0.01007757  1.44781584 -3.3079218 ], action: 1, reward: -1.0
obs: [-0.63433623  0.77305727 -0.86813646  0.49632559  0.48541698 -1.98910707], action: 0, reward: -1.0
obs: [-0.64812599  0.76153312 -0.69537648  0.71864564 -0.30452259 -0.82549355], action: 2, reward: -1.0
obs: [-0.51864307  0.85499086 -0.69380649  0.72016148 -1.32424668  0.8055575 ], action: 0, reward: -1.0
obs: [-0.1732413   0.98487941 -0.86574971  0.50047722 -2.42461072  1.97982082], action: 1, reward: -1.0
obs: [  4.25456913e-01   9.04978682e-01  -9.99994603e-01  -3.28541293e-03
  -3.68671806e+00   3.30335269e+00], action: 2, reward: -1.0
obs: [ 0.95372669  0.3006749  -0.66841324 -0.74379012 -4.41286524  5.15337878], action: 2, reward: -1.0
obs: [ 0.83538676 -0.54966259  0.50883078 -0.86086656 -4.43176723  7.52964327], action: 1, reward: -1.0
obs: [ 0.1581863  -0.98740928  0.86047626  0.50949054 -3.60913754  7.50043985], action: 0, reward: -1.0
obs: [-0.3622969  -0.93206274 -0.20724711  0.97828863 -1.70122947  5.06890232], action: 0, reward: -1.0
obs: [-0.5156939  -0.8567729  -0.86939734  0.49411362 -0.06858319  3.45360338], action: 0, reward: -1.0
obs: [-0.40993174 -0.9121162  -0.99975234 -0.02225439  1.2166262   1.92218601], action: 0, reward: -1.0
obs: [-0.06050682 -0.99816778 -0.96987188 -0.24361555  2.41178857  0.31836753], action: 0, reward: -1.0
obs: [ 0.51971536 -0.85433948 -0.98720977 -0.15942667  3.64055164 -1.12549416], action: 0, reward: -1.0
obs: [ 0.97896656 -0.20402076 -0.98410674  0.17757791  4.40398798 -2.19502491], action: 0, reward: -1.0
obs: [ 0.7909205   0.61191892 -0.76971751  0.63838464  4.04716348 -2.86476543], action: 2, reward: -1.0
obs: [ 0.22209125  0.97502588 -0.32773698  0.94476901  2.77446906 -2.43464041], action: 0, reward: 0.0
Episode finished after 5611 timesteps
```

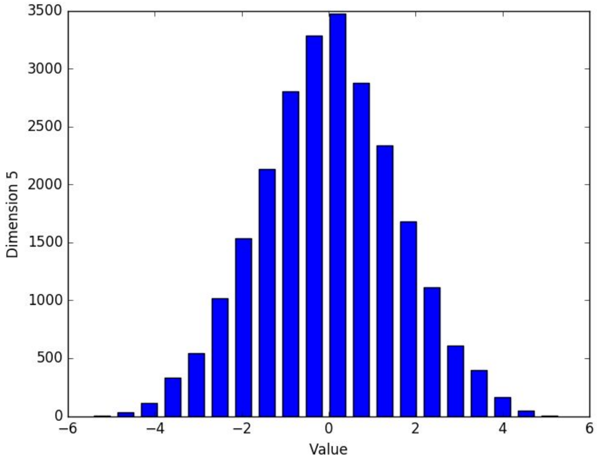
Based on the above exploration, we see that the observation space is 6-dimensional and continuous value, and the action space is 1-dimensional with three possible values: 0, 1, 2. The environment gives a reward of -1 for each timestep the Acrobot does not achieve its goal.

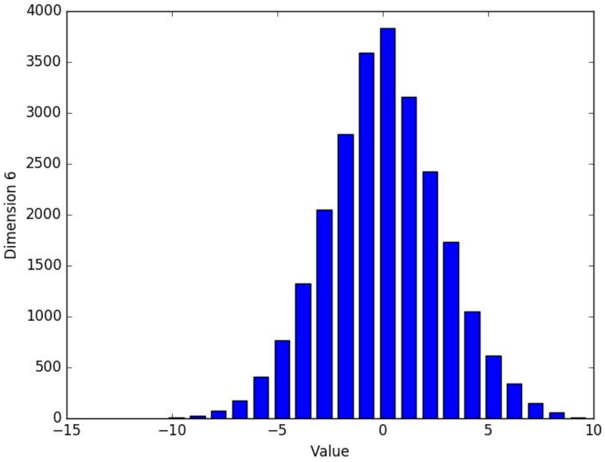
Regarding what each dimension of the state space represents, and what each action value represents, OpenAI Gym’s documentation does not give any explanation. This is possibly intentional, as we would like our learning agent to “learn” the environment with minimal domain expertise to start.

Exploratory Visualization

To further explore the input observation space, I ran 10 episodes of Acrobot by uniformly sampling random actions. The code is available at ‘explore2.py’. I plotted the histograms of each of the 6 observation dimensions:







Looking at the figures above, all 6 observation dimensions appear to be centered around 0, thus we do not need to subtract the mean from any dimension. Dimensions 1 through 4 are all within the range of -1 to 1.

Looking at dimensions 1 and 3, the input data is heavily skewed towards 1. However, since we are taking random actions at each timestep, this could be a reflection of the fact that the Acrobot is stuck in an undesirable state for a very long time. For example, dimensions 1 and 3 may represent the height of the two joints, and the Acrobot may be spending a lot of time with its joints in at a low height, since our agent is taking random actions. After our agent learns a better policy to solve Acrobot, this input distribution may change.

Looking at dimensions 5 and 6, we see the value range is much larger than dimensions 1 through 4. Dimensions 1 through 4 all have a range between -1 and 1, whereas dimension 5 has a range between -6 and 6, and dimension 6 has a range between -10 and 10 (minus outliers). However, as mentioned previously, the input distribution may change as the agent better learns how to solve the environment, so we cannot decisively say that we must scale dimensions 5 and 6 as a data preprocessing step. Even if it would benefit the learning process to scale these dimensions, it is not clear the magnitude by which to scale, since the distribution may change. Even so, the neural networks may be able to learn the appropriate scaling factor in its first or second layer automatically.

As an aside, if I were to guess what each observation dimension represents, I would guess that dimensions 1 and 3 are the vertical positions of the two joints, as discussed above. Dimensions 2 and 4 appear to be the horizontal positions of the two joints, since they are relatively evenly distributed, which may be a result of applying random torque to the actuated joint. It is hard to guess what dimensions 5 and 6 represent, but one possibility is that dimensions 5 and 6 represent the angular velocity of the two joints, since that information is important in learning how to achieve the goal.

Algorithms and Techniques

Deep reinforcement learning approach:

I have chosen a deep reinforcement learning approach to solve Acrobot. Specifically, I have chosen to use policy gradient methods with actor/critic networks. Policy gradient methods use deep neural networks to learn both the optimal policy ("actor" network) and the Q-value of a state ("critic" network).

The critic learns to estimate the Q-value given a state (optionally given a state/action pair instead). It treats the calculated/empirical Q-value as the target value, and the network output as the predicted value. Thus, we attempt to solve a supervised learning regression problem for the critic network. However, for reinforcement learning problems, we are only able to calculate the empirical Q-values at the end of each episode, via the reward discounting process.

The actor, on the other hand, learns to predict the optimal probability distribution over all actions, given the state. In other words, the actor network learns the optimal policy. The learning leverages the "advantage function", which is how much better the empirical/observed Q-value is versus the Q-value predicted by the critic, and the policy gradient formula.

For more detailed descriptions and derivations of policy gradients, please refer to Andrej Karpathy's blog post at http://karpathy.github.io/2016/05/31/rl/. Using policy gradient methods with actor/critic networks is known as "Deep Deterministic Policy Gradients", and the original paper is available at https://arxiv.org/pdf/1509.02971v5.pdf.

An alternative approach is to use tabular Q-learning to solve Acrobot. However, given the continuous nature of the observation space, a tabular Q-learning approach would be unfeasible. A fine-grained binning of the input space would be required, which means the space and time complexity to store and process the Q-value table would be impractical.

Stabilizing the learning process:

In general, running an episode of Acrobot will produce strongly temporally correlated data, e.g. the data obtained in the final timesteps will be strongly correlated to the data in prior timesteps. Thus, running a training step after each episode is likely to cause unstable learning.

One approach to alleviate this issue is the use of "experience replay". The idea is to have a replay buffer to store experiences (tuples of state, action, advantage, empirical Q-value) across multiple episodes, then randomly sample a subset of the replay buffer on which to perform training. A disadvantage to this approach is that the algorithm is very serial, and a more parallel approach would be desirable, as it may scale better with appropriate compute resources.

Recently, a parallel approach was proposed a paper titled “Asynchronous Methods for Deep Reinforcement Learning”: http://arxiv.org/pdf/1602.01783v2.pdf. The idea is to run multiple environments in parallel, and perform training on the data from all parallel environments. This approach is more scalable, and I have implemented this for the learning the Acrobot environment.

Benchmark

As mentioned in the “Metrics” section, the benchmark score I have chosen is -100 over 100 consecutive episodes. The goal for my learning agent is to achieve a score higher than the benchmark. In particular, I calculate the learning agent’s final score as its average total reward over 100 episodes, in a single environment (no parallel environments), and no training steps during those 100 episodes.

The benchmark score itself is roughly 20x better than taking random actions, and beating the benchmark score would put my learning agent in the top half of the Acrobot leaderboard on OpenAI Gym: https://gym.openai.com/envs/Acrobot-v1

III. Methodology

Data Preprocessing

As mentioned in the “Exploratory Visualization” section, no manual data preprocessing is required.

Implementation

First, the following dependencies are required to implement my learning agent:

- Python 2.7
- NumPy
- Matplotlib
- OpenAI Gym
- TensorFlow 0.10.0

As a starting point, I used Kevin Franz’s TensorFlow-based python code, which implemented a policy gradient actor/critic method to solve OpenAI Gym’s Cartpole environment. Kevin’s code is available at:
https://github.com/kvfrans/openai-cartpole/blob/master/cartpole-policygradient.py

I enhanced and modified the starter code to fit the more complicated Acrobot environment, implemented more complex neural networks to learn the more complicated environment (3-layer fully-connected actor network, 4-layer fully-connected critic network), added the ability to run and train over multiple parallel environments (I chose 5 environments), and more. I also added “bookkeeping” enhancements, such as the metrics described in the “Metrics” section, the ability to modify parameters, the ability to save a trained model, and the ability to resume training from a previously saved model. My code is located in the file ‘learning_agent.py’.

Other things to note about my implementation:

- To create the actor/critic neural networks, I used TensorFlow-Slim, which abstracts neural network creation
    - TensorFlow-Slim is included in TensorFlow 0.10.0
    - More information about TensorFlow-Slim is available at: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim
- I implemented a modified version of “experience replay” (see ‘replay_buffer’ variable in ‘learning_agent.py’), where I populate a replay buffer with experiences from multiple episodes, randomly sample a subset of those experiences, run training, and repeat. However, I found that the parallel environment approach gave better results, so I did not use my modified experience replay feature. Note that my code for experience replay is still there, but I set the parameters such that the learning agent does not use experience replay at all.
    - The experience replay mechanism used in the “Playing Atari with Deep Reinforcement Learning” paper (https://arxiv.org/pdf/1312.5602v1.pdf) sampled from a replay buffer that contained experiences from previous neural network weights. My modified version only sampled experiences from the current neural network weights. Currently I do not understand how learning can be achieved by sampling experiences from previous neural network weights -- maybe it only works for deep Q-learning and not policy gradient methods, I’m not sure.
- I included “from __future__ import …” statements at the beginning of the file, to future-proof my Python 2.7 code, so it may run in Python 3 if needed

Throughout my implementation process, I encountered numerous complications:

- Making the parallel environments work with the reinforcement learning process was complicated
    - I had to add an extra dimension to multiple tensors throughout my code, to support the parallel environments
    - Stochastically, some environments will finish earlier than others, and accounting for this required complicated logic
- When the algorithm does not learn anything, e.g. when I see rewards decrease over time, the debugging process was complicated, since there are 2 neural networks to consider, and the reward discounting process could also be a culprit
    - Methodically checking different variables and printing/plotting them over time was helpful

Refinement

Initially, when implementing the code in ‘learning_agent.py’ (described in the “Implementation” section), I manually set the parameters without much refinement. I initially settled on the following parameters:

```
ACTOR_LR = 0.005  # actor network learning rate
CRITIC_LR_SCALE = 0.5  # scaling factor of critic network learning rate, relative to actor
REWARD_DISCOUNT = 0.97  # reward discount factor
A_REG_SCALE = 0.0005  # actor network regularization strength
C_REG_SCALE = 0.0005  # critic network regularization strength
```

After training over 1000 episodes, the final score for the agent was -216.639. To reiterate from the “Benchmark” section, the final score is the learning agent’s average total reward over 100 episodes, in a single environment (no parallel environments), and no training steps during those 100 episodes.

To refine the model, a methodical approach to find the optimal parameter combination is required. In the file ‘search_params.py’, I have created a script to execute a random search across the following parameter values:

```
ACTOR_LR = [0.05, 0.01, 0.005, 0.001, 0.0005]
CRITIC_LR_SCALE = [1.25, 1.0, 0.75, 0.5, 0.25]
REWARD_DISCOUNT = [0.99, 0.97, 0.95, 0.93]
A_REG_SCALE = [0.005, 0.0005, 0.00005]
C_REG_SCALE = [0.005, 0.0005, 0.00005]
```

Due to time constraints, an exhaustive grid search across all possible parameter combinations is impractical. Thus, I have opted for random search across 20 iterations. In each iteration, the script randomly chooses a parameter combination, without replacement, runs the reinforcement learning algorithm for 1000 episodes, and records the final score. Note for each parameter combination, I run the reinforcement learning algorithm 3 times from scratch. This is because the random neural network weight initialization combined with random initial environment state influences the learning process -- when the agent/environment starts in a “bad” state, the entire learning process suffers. The best score out of the 3 runs is recorded.

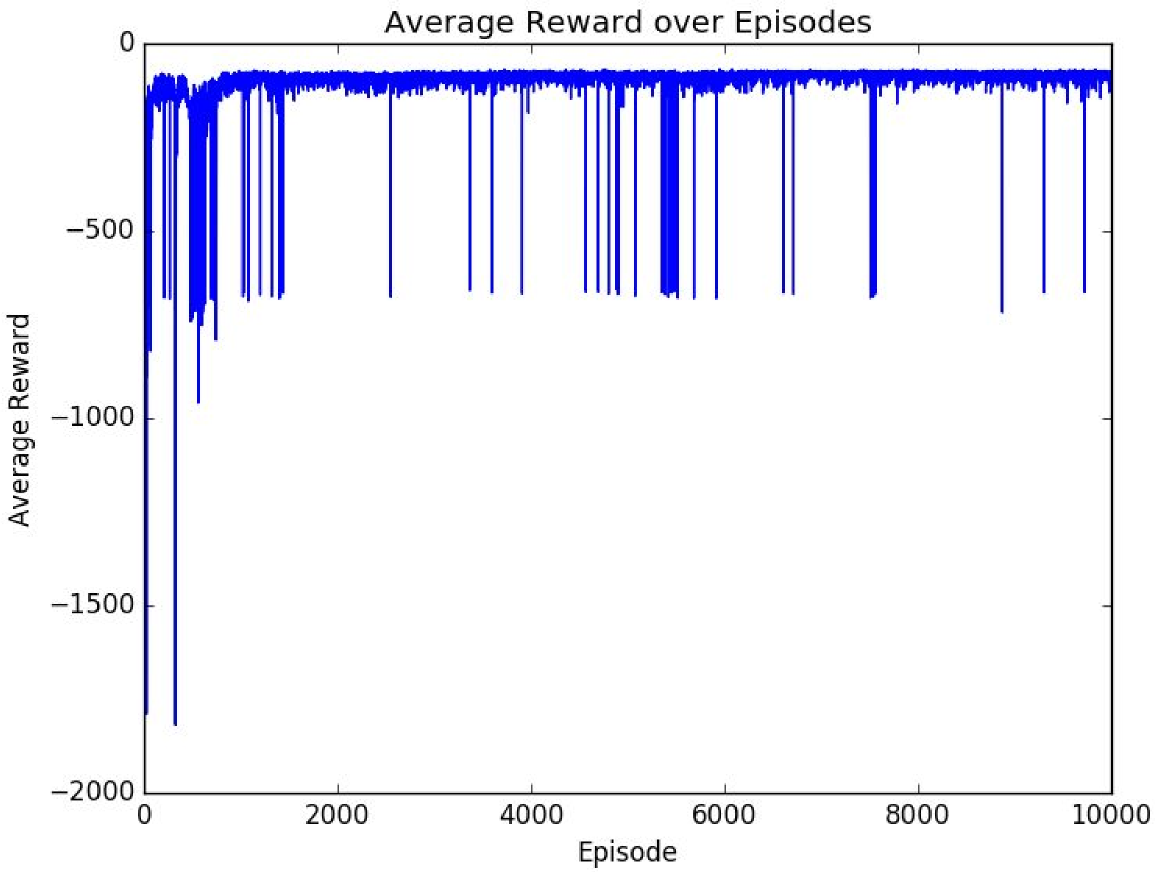
The following table shows the top 5 parameter combinations from the parameter search. The complete results of the parameter search are available in the file ‘search_params.csv’. Note the results in ‘search_params.csv’ are unsorted.

ACTOR_LR	CRITIC_LR_SCALE	REWARD_DISCOUNT	A_REG_SCALE	C_REG_SCALE	score
0.05	**0.5**	**0.97**	**0.00005**	**0.0005**	**-92.944**
0.05	0.25	0.97	0.005	0.00005	-93.812
0.01	0.5	0.95	0.005	0.0005	-108.762
0.001	0.75	0.99	0.0005	0.0005	-118.54
0.05	1	0.95	0.00005	0.005	-158.192

From the results above, we see that the optimal parameter combination achieved a score of -92.944, after 1000 training episodes.

For further refinement, taking the optimal parameter combination (the row in bold in the above table), I ran 1500 episodes of training from scratch, followed by another 8500 episodes of training with the learning rate reduced by a factor of 10 (i.e. ACTOR_LR = ACTOR_LR / 10), for a total of 10,000 training episodes. The code for this process is in the file ‘full_training.py’.

After executing the aforementioned training process, **the learning agent was able to achieve a final score of -88.580**. The figure below plots the average total rewards over episodes, illustrating how the agent learns to improve in Acrobot.



The final model was saved as a TensorFlow model, in the file ‘models/model.ckpt’.

IV. Results

Model Evaluation and Validation

From the previous “Refinement” section, I have chosen a set of optimal parameters, trained the model over 10,000 episodes, and saved the final trained model to disk.

To evaluate and validate the final model, I ran 1000 episodes of the model in inference mode, i.e. no training steps were performed. I recorded the total reward (i.e. the “score”) for each episode. With this data, I calculated the mean and standard deviation, and created a plot. Based on the metrics I defined, I also want to calculate an average 100-episode score, so I chose the final 100 episodes over which to calculate an average score. This 100-episode score is what I will use to compare to my benchmark score of -100.

Finally, I enabled OpenAI Gym's environment monitor, so that I could upload my results to the OpenAI Gym leaderboard for Acrobot: https://gym.openai.com/envs/Acrobot-v1.

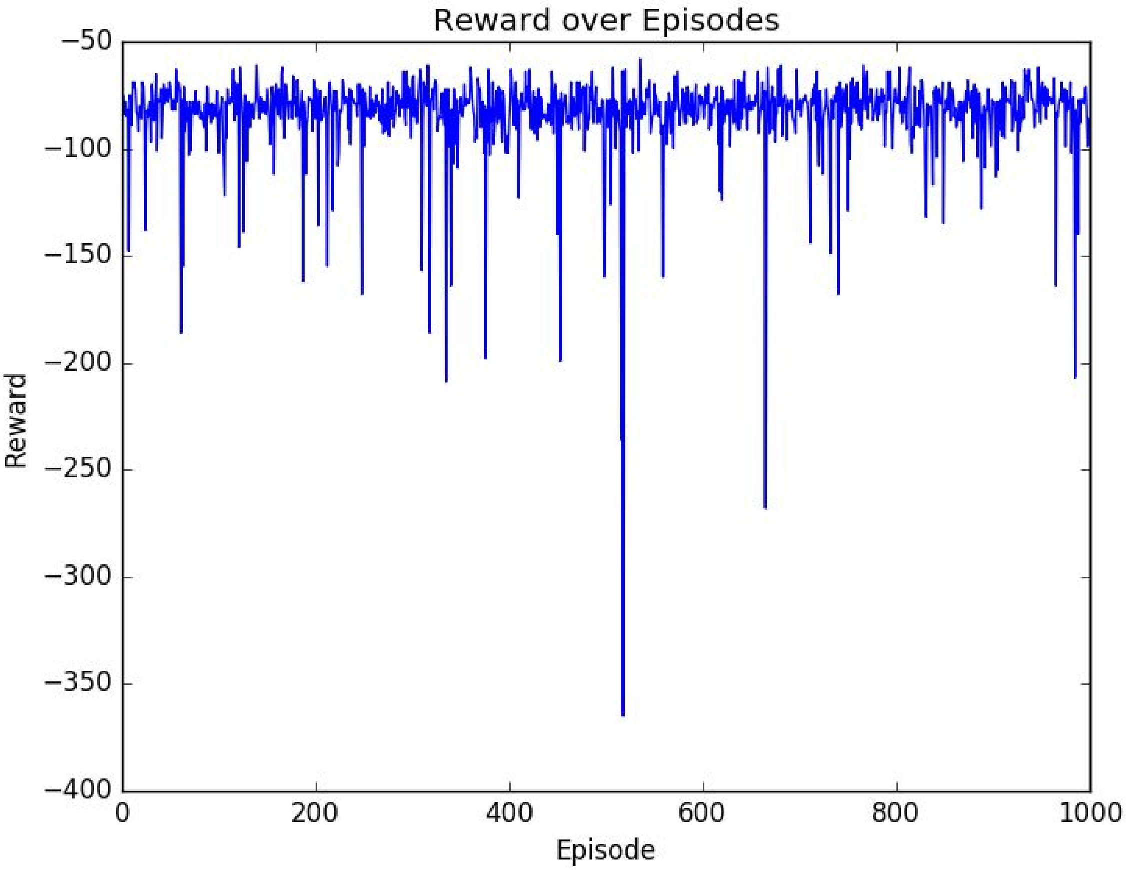
The code to accomplish the model evaluation and validation is in 'model_eval.py'. Note that most of the logic is located in the function 'model_evaluation()', in 'learning_agent.py'.

After running 'model_eval.py', I obtained the following results for the final model:

- Average score over 1000 episodes: -83.769
- Standard deviation of scores over 1000 episodes: 20.433
- Average score for final 100 episodes: -82.710

The final model has met our goal of beating the benchmark score, to achieve an average 100-episode score greater than -100. This is evidenced by the fact that the average score over the final 100 episodes is -82.710. It is further supported by the fact that the average score over 1,000 episodes is still less than -100.

Below is a plot of the total reward (score) over episodes:



In term of model robustness, the average score over 1,000 episodes is very close to the final 100-episode score. This is further illustrated by the fact that the standard deviation of the scores is relatively small, at 20.433.

I also uploaded my model evaluation results to OpenAI Gym's Acrobot leaderboard, at https://gym.openai.com/envs/Acrobot-v1. My username is "georgesung", and I placed 3rd on the leaderboard as of September 20, 2016. Note that the leaderboard shows my *best* 100-episode performance over the 1,000 episodes (-80.69), whereas I only recorded my *final* 100-episode performance in my results above (-82.71).

To upload my results to OpenAI Gym's leaderboard, I ran the following in the command line:

```
python -c 'import gym; gym.upload("openai_data", api_key="my_api_key")'
```

Justification

From the previous section, “Model Evaluation and Validation”, the final model has beaten the benchmark score of -100 over 100 consecutive episodes. To reiterate:

Average score over 1000 episodes:	-83.769
Standard deviation of scores over 1000 episodes:	20.433
Average score for final 100 episodes:	-82.710

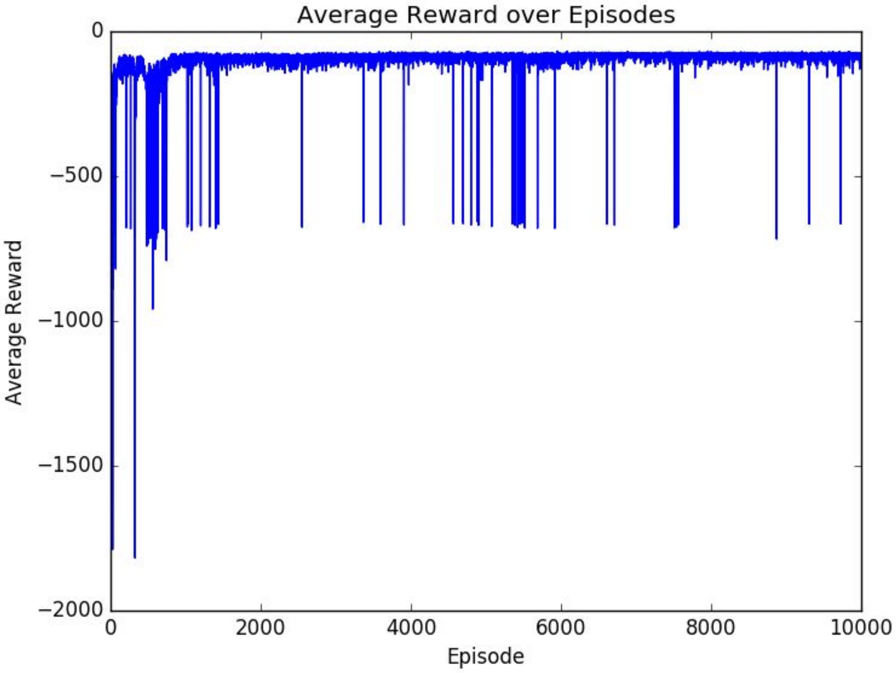
I believe these results are statistically significant enough to consider the problem *solved*. From the figure presented in the previous section, “Reward over Episodes”, it looks like most of the score’s variation is contributed by corner cases where the score/reward is below approximately -130, thus we saw a standard deviation of 20.433. However, these appear to be rare corner cases where the random environment initialization combined with the stochastic action sampling likely resulted in certain sub-optimal behavior. For the most part, the final model was able to achieve a score greater than -100, as seen in the figure.

V. Conclusion

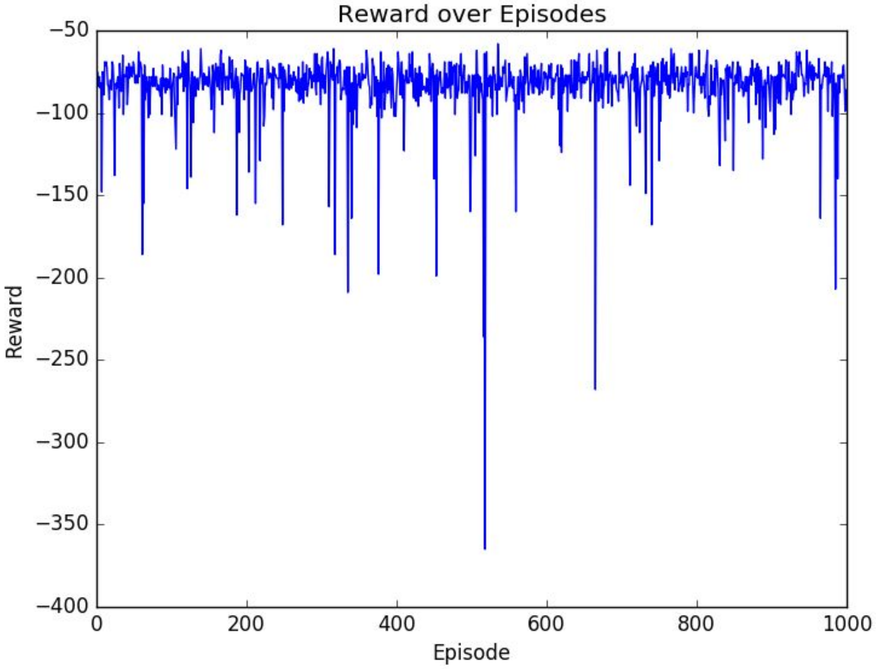
Free-Form Visualization

Throughout this project, visualizations have been very helpful.

Visually, I was able to see the agent's learning progress through training (figure from "Refinement" section):



It was also great to see the final trained agent performing consistently well (figure from "Model Evaluation and Validation" section):



Reflection

The journey to solve Acrobot using deep reinforcement learning has been a great learning experience for me. Before starting this project, I had a strong interest in both deep learning and reinforcement learning, and this project gave me great experience in both. Further, the recent availability of OpenAI Gym provided a great framework to experiment with reinforcement learning algorithms, especially in Python.

Initially, I researched different methods and techniques for deep reinforcement learning. I learned that policy gradients with actor/critic networks were the current state-of-the-art, so it would be great for me to gain experience with this technique.

Even though I was able to leverage existing starter code to implement actor/critic networks, I ran into complications with learning stability, since the starter code was used to solve a simpler Cartpole environment. This gave me the opportunity to learn about techniques to stabilize the deep reinforcement learning process, namely experience replay and parallel environments. I implemented the parallel environments technique, and enhanced/modified the starter code in numerous ways, to fit the needs of this project.

The code to perform model refinement, and model evaluation/validation were written entirely from scratch. I modified the starter code to allow for variable parameters, to enable the model refinement/evaluation/validation code. Visualizations were also very important, and I implemented the code for appropriate visualizations.

There were many interesting aspects of this project. Simply learning about the different approaches to deep reinforcement learning was interesting. Moreover, actually implementing and experimenting with policy gradients with actor/critic networks was eye-opening. If I were to utilize deep reinforcement learning in a future problem, I would be comfortable in doing so. Placing 3rd on the OpenAI Gym Acrobot leaderboard was a nice bonus.

I ran into various difficulties in this project as well. First, there are many moving parts in my reinforcement learning algorithm (two separate neural networks, reward discount algorithm, parallel environments, etc.), so it was hard to debug when things went wrong. Ultimately the debug effort involved printing out the shapes and/or values of different tensors, and trying to find the first point of failure. Another difficulty was the randomness of the initial learning process. If the neural network's random weight initialization did not "play well" with the initial random state in the environment, what I observed was that the learning agent took significantly longer to converge on a solution. Sometimes, the learning agent would *diverge*, i.e. the total rewards would *decrease* after training.

The final model does fit my expectations of the solution. As discussed in the “Justification” section, the final model beats the benchmark I have set, in a statistically significant way. Thus the final model can be used in general settings to solve Acrobot, provided the Acrobot environment matches that of OpenAI Gym’s “Acrobot-v1” environment. However, the final model would *not* be able to solve reinforcement learning environments other than Acrobot.

Improvement

For improving in the Acrobot environment, a major area of improvement would be in model refinement. Due to time constraints, in my parameter search, I executed 1,000 training episodes per parameter combination (see “Refinement” section for more details). This limits the learning rates I could explore. For example, if I had more time and/or computing power, I may modify the parameter search to execute 100,000 training episodes per parameter combination. This would allow me to feasibly explore lower learning rate values, which may ultimately lead to better scores. For example, looking at the top-ranked Acrobot submission on the leaderboard, https://gym.openai.com/evaluations/eval_X4wWctnvRvqcVHy4FLDRwQ, we see *ceobillionaire*’s learning curve appears to use a lower learning rate than mine. My learning curve is in the “Refinement” section, the plot is titled “Average Rewards over Episodes”.

Of course, another improvement would be to leverage my current code to solve other reinforcement learning problems, such as Pong (https://gym.openai.com/envs/Pong-v0), Doom (https://gym.openai.com/envs#doom), robotics, self-driving cars, etc. Even if my existing code may not be leveraged, the skills and knowledge I gained from this project will certainly be applicable.