Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym

Definition:

My goal is to use reinforcement learning techniques to learn solutions to the four "classic control" problems in the OpenAI Gym (which I will abbreviate OAIG from here on) with a single unified algorithm. The four environments are described in detail below. The OAIG is designed specifically for this sort of exercise, and provides an API that allows algorithms to easily access the possible actions and states available in a given environment. Some of the environments are discrete (there is a specific set of available actions) and some are continuous (there is a range of actions in a specific real number domain). I intend to implement a version of Normalized Advantage Functions, a type of Q-learning that is adapted to work in continuous domains.

Q-learning finds an optimal solution to any control problem that can be defined as a Markov Decision Process. An MDP has a discrete set of states the world can be in, a discrete set of actions possible to take from those states, and a reward and transformation for each state-action pair. NAF adapts this to work with a continuous state set and a continuous action set.This is a useful enhancement because Q-learning isn't applicable to problems such as robotics control where there are essentially infinite states and actions. By using Normalized Advantage Functions, I hope to be able to solve all four classical control problems without having to excessively tailor the algorithm to each state/action space.

Environments:

CartPole-v0: The cart-pole problem is one in which the algorithm has to learn to balance a pole on a cart that can only move is two dimensions. The cart has a position along a single axis with a velocity dimension, and the pole is attached by one end to the center of the cart and has a rotational dimension and a rotational velocity dimension. At each time-step, the agent can apply force of +1 or -1 to the cart. The agent receives a reward of +1 for every time step that the pole is standing upright up to a maximum of 200. The threshold for success is receiving an average reward greater than 195.0 across 100 consecutive runs.

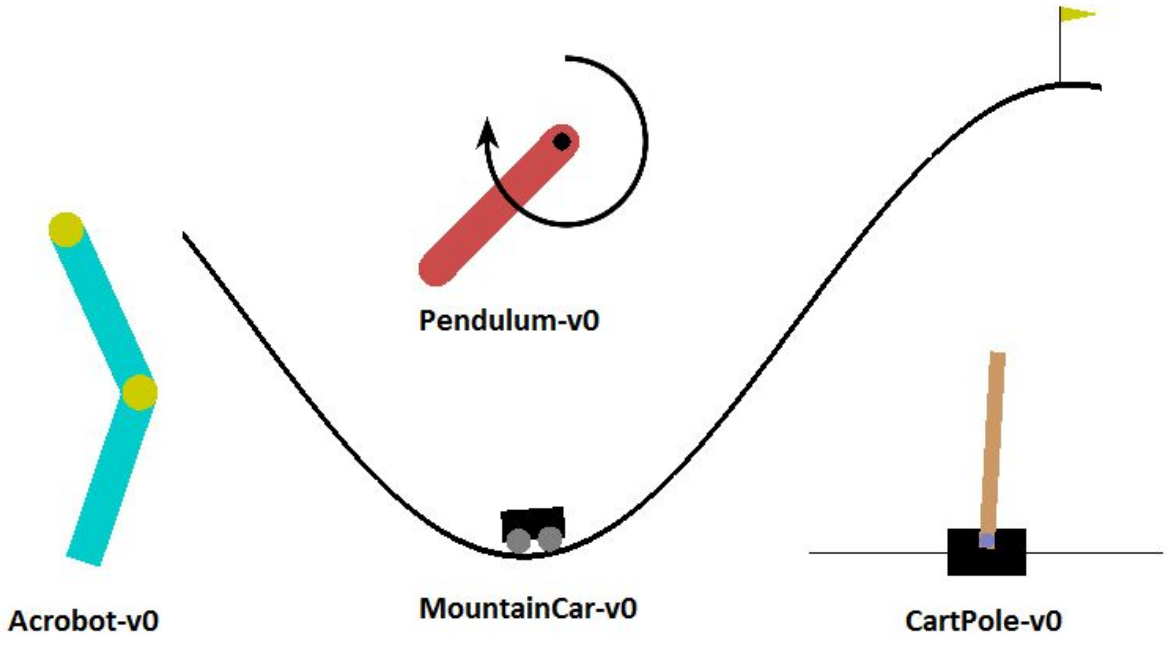
Acrobot-v0: The acrobot problem consists of two poles of equal length attached end-to-end and hanging from an immovable central point. Each pole has a rotational velocity and each joint has a rotational angle. Each time-step, the agent can apply a torque of +1, 0, or -1 to the attachment point. If the tip of the second

point doesn't exceed a certain height, the algorithm receives a reward of -1. This is solved if the agent receives an average reward greater than -100.0 over 100 consecutive runs.

MountainCar-v0: There is a car in the valley of a sinusoidal hill. It has a velocity and displacement. At each time-step, the algorithm can apply a force of +1, 0, or -1 to the car. The agent receives -1 reward at each timestep until it reaches the peak of the hill to the right. This is solved if the agent averages greater than -110.0 reward over 100 consecutive runs.

Pendulum-v0: There is a single pole attached by one end to an immovable pivot. It has a rotational velocity and displacement. At each time-step, that agent can apply a torque in the continuous range -2.0 to 2.0. The agent receives a reward related to how "upright" it is after each timestep. The goal is to maximize time spent upright, and I am targeting a score greater that -150.0 over 100 consecutive runs.

Each environment has a corresponding visualization:



Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym

Analysis:

Each of the four environments has a continuous state-space. This would be a problem with traditional Q-learning, but should not be an issue with NAF. The action spaces are all discrete except for Pendulum-v0, which will make NAF an excellent choice for all of these environments.

The API to OAIG makes determining the boundaries of each state space very easy to do:

	CartPole-v0 [4 state dimensions]	Acrobot-v0 [4 state dimensions]	MountainCar-v0 [2 state dimensions]	Pendulum-v0 [3 state dimensions]
State High	4.8 3.402e+38 0.419 3.402e+38	3.142 3.142 12.566 28.274	0.6 0.07	1. 1. 8.
State Low	-4.8 -3.402e+38 -0.419 -3.40e+38]	-3.142 -3.142 -12.566 -28.274	-1.2 -0.07	-1. -1. -8.
Actions	0 or 1	0, 1, or 2	0, 1, or 2	-2. to 2.

The action available in each environment are representative of specific actions taken by the agent, so the action “2” in the MountainCar-v0 environment corresponds to force applied to the right, “0” to the left, and “1” represents no force applied.

To compare the working agents after training, I calculated averages over 100 runs for an agent that takes a random sample of the action space at every time-step up to the 200-step cut-off:

Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym

Run	CartPole-v0	Acrobot-v0	MountainCar-v0	Pendulum-v0
1	16	-200	-200	-875.3389506
2	41	-200	-200	-1288.919575
3	19	-200	-200	-1473.86944
4	19	-200	-200	-1182.085518
5	19	-200	-200	-974.1607737
...	...	...	...	...
96	15	-200	-200	-883.1205064
97	15	-200	-200	-879.9719266
98	13	-200	-200	-1670.598111
99	27	-200	-200	-1514.20523
100	11	-200	-200	-1576.855572
Average:	**22.56**	**-200**	**-200**	**-1291.606194**

None of these even came close to succeeding even once, let alone for the entire run. See this document for the full data: Link.

Algorithm:

I am going to use Normalized Advantage Functions as described in the linked journal article. NAF are a form of Q-learning that use a form of deep neural network in place of the Q function. First, several structures are initialized: a "replay buffer" to store state-action -> new state, reward transitions in initialized to null, the normalized Q network (a neural network) is initialized to random values, and a target network is initialized to an identical set of weights. At each time-step in the simulation, the agent will use the observed state as the input to the target network and add some random noise to select an action. The agent will get a new observed state and a reward. The original state, new state, action, and reward will be saved in a replay buffer. The agent then randomly samples the replay buffer and runs training epochs on the Q network, minimizing mean squared error, and finally uses that result to update the the target network weights and biases using a discounted value (τ * normalized Q network + (1 - τ) * target network).

Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym

Algorithm 1 Continuous Q-Learning with NAF

Randomly initialize normalized Q network $Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q)$.
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$.
Initialize replay buffer $R \leftarrow \emptyset$.
for episode=1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1)$
  for t=1, T **do**
    Select action $\boldsymbol{u}_t = \mu(\boldsymbol{x}_t|\theta^\mu) + \mathcal{N}_t$
    Execute $\boldsymbol{u}_t$ and observe r_t and $\boldsymbol{x}_{t+1}$
    Store transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ in R
    for iteration=1, I **do**
      Sample a random minibatch of m transitions from R
      Set $y_i = r_i + \gamma V'(\boldsymbol{x}_{i+1}|\theta^{Q'})$
      Update θ^Q by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(\boldsymbol{x}_i, \boldsymbol{u}_i|\theta^Q))^2$
      Update the target network: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    end for
  end for
end for

Q(x, u) is built from A(x, u) + V(x):
V(x) is a single linear output node from the deep neural network at the heart of the NAF.
A(x, u) is calculated as $-\frac{1}{2}(u - \mu(x))^T P(x)(u - \mu(x))$
$\mu(x)$ is the max output for Q(x) across all u.
P(x) is $L(x)^T L(x)$, where L(x) is a lower-triangular matrix with entries from a linear output layer of the NAF DNN. The diagonal elements of L(x) are exponentiated.

Benchmarks:

The benchmarks are different for each environment in OAIG and are described above. I hope to achieve these benchmarks with fewer than 1000 runs in each case. I will submit agent results to the OAIG leaderboards as well.

Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym

Methodology:

The initial step will be to create a structure in which I can run repeated trials of different agents to compare results. Since the states, rewards, and actions are all provided by the environment, most of the data is already in a useable format. The more difficult task will be in building the normalized Q network infrastructure. I will use TensorFlow to implement the NAF algorithm.

Implementation:

I created a setting to compare agents and created RandomAgent which takes a random action at every time step. I benchmarked it across the four environments to get a baseline to compare my future NAFAgent against. The average scores over 100 runs are included in a table above.

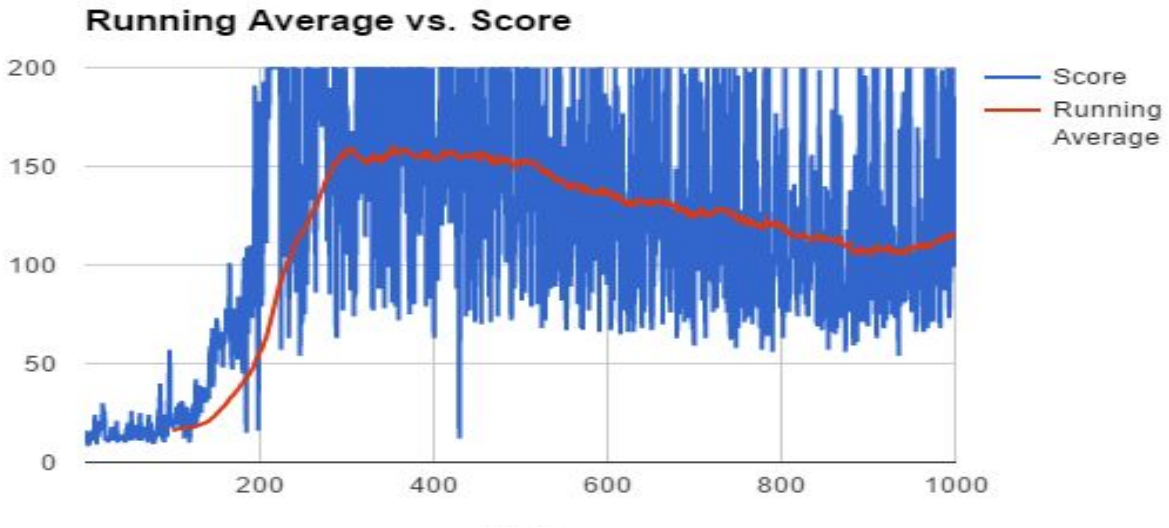
Next, I implemented NAFAgent in Tensorflow. Initially, I targeted my code at solving the CartPole-v0 environment, since it is the simplest and most straight-forward. Building the network had some challenges: I implemented the algorithm for a discrete domain first, so I structured u as a one-hot vector of possible actions. This meant that μ(x) needed to output a one-hot vector as well. When I represented this, the values converged to either 0.0 for all actions or NaN for all actions. I used softmax to normalize μ(x) output, but I ended up converging to 0.5 for all actions. Building L(x) was also difficult, as I had to create a linear output with the correct number of elements, then convert it to the correct form (lower-triangular matrix with diagonal elements exponentiated). Once everything was functioning as it should, the output was either the same as or worse than RandomAgent.

Refinement:

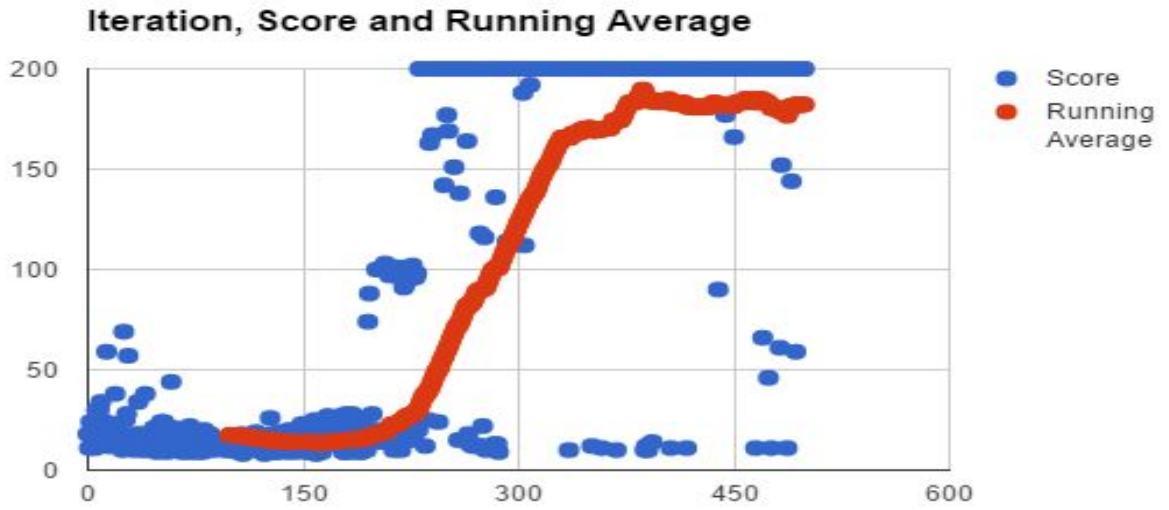
I looked for errors in my logic, compared it to another implementation of NAF (https://github.com/carpedm20/NAF-tensorflow, which targets only continuous output environments), and tried different methods of constructing the core A(x, u) and V(x) functions. I was still unable to get a useful algorithm for CartPole-v0. I altered the reward function to give greater rewards the longer the simulation went on (the longer it kept the pole up), but this meant I was putting my own knowledge of the system into the algorithm.

David Timm
https://github.com/dtimm/mlnd-openai-gym

The next stage I tried was to implement a more basic DNN learning algorithm, abandoning NAF. I created a network with three fully-connected layers with 10, 20, and 10 nodes, respectively. At each time interval, I performed several steps of Adaptive Moment Estimation (Adam optimization) with a learning rate of 0.001, minimizing $(Q(x, u) - r + \gamma * \text{argmax } Q(x'))^2$ and the current transition and a random sample of previous state-action-transitions. Running 5 training steps per iteration with 200 random samples, the agent learned a policy that averaged a score of 150 across 100 episodes after 500 training episodes, much better than the score of RandomAgent:



After that, the running average dipped back down. I changed the initial learning rate to 0.005, and now the algorithm performs like RandomAgent until it hits on a very good policy:



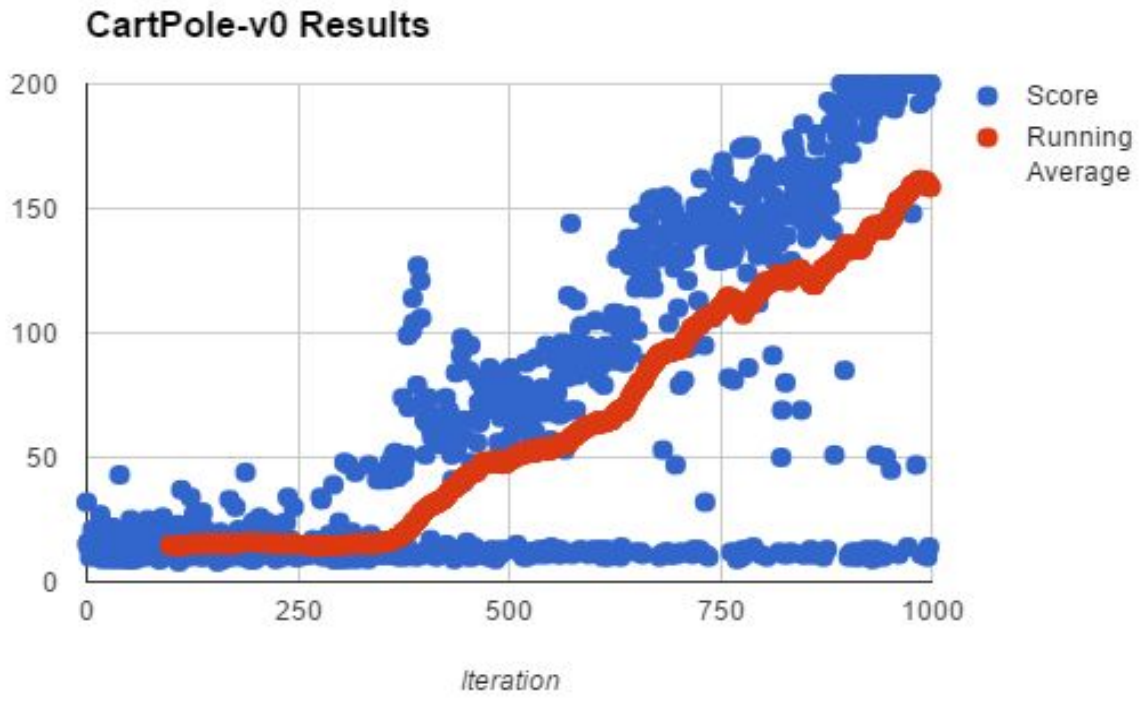
David Timm
https://github.com/dtimm/mlnd-openai-gym

Unfortunately, this policy still sometimes crashes the cart directly to one side, resulting it the very low scores. I tried giving the Adam optimizer a separately decaying alpha score (it already decays its learning rate) but the results were similar. I decided to generalize it to the other models' state-action space. Since NAF didn't workout , it will be impossible to calculate argmax Q(x) for Pendulum-v0, since u is continuous in that environment.

I tried to give the algorithm some incentive to explore by positively weighting a random, distant state. This took a lot of extra time to run, but ultimately decreased training steps.

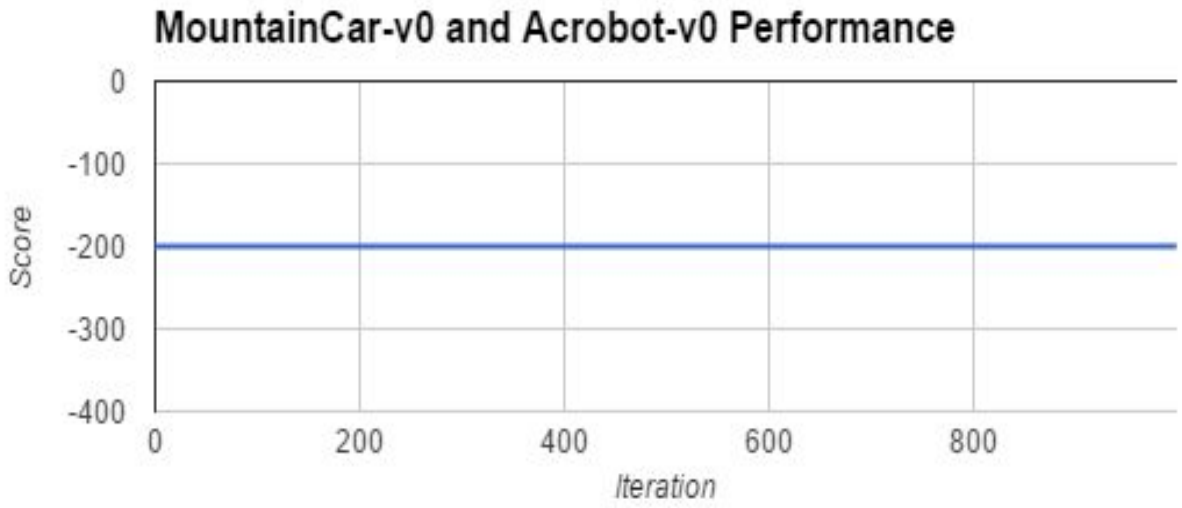
Results and Conclusion:

After converting the algorithm to use a one-hot vector for u and verifying that it still functioned correctly for CartPole-v0, I ran MountainCar-v0 and Acrobot-v0 for 1000 steps each, but neither learned a functional policy. I decreased the epsilon decay, hoping that they might get a random hit and learn what to do from there, but it didn't work. CartPole-v0 had a smoother learning process with new decay, however.



Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym



The results I achieved did not remotely match the benchmarks I laid out. I was only able to solve one of the four environments, and it took twice as many episodes as I wanted to find that solution. The model that is does find, while it can technically solve the challenge, still gives inconsistent results. Sometimes the pole falls over within 10 time steps, which is as fast as it possibly can.

I had nothing but trouble with Normalized Advantage Functions. There were several components: one was that there is only a single paper written about the subject, and another is that there is only one existing implementation available as source. It may have been over-ambitious to select such a specific method for solving these problems. The model needed careful attention paid to initial weights and biases on the hidden and output layers.

The NAF algorithm itself was absolutely fascinating, and it would have been even more interesting if I'd managed to get it working. There were many things I had never been exposed to before, and reading a research paper gave a wonderful excuse to dive deep into the literature of deep reinforcement learning. The indirect training method (optimizing the output of a function that you never use directly) was very interesting, and the basis of the NAF algorithm, deep decision policy gradients and policy gradients as a whole were interesting (and long) asides to this project.

Once I switched algorithms to a more traditional Q learning approach (still with a deep network), the results were much better. While I was unable to solve MountainCar-v0 and Acrobot-v0, solving CartPole-v0 was a significant milestone.

Udacity Capstone Project: OpenAI Gym

David Timm
https://github.com/dtimm/mlnd-openai-gym

It was beneficial to have a framework to test my ideas against in the form of the OpenAI Gym. There were downsides as well: the reward function was not optimal for Q learning-based algorithms, which value immediate reward. At every time step in each environment, there is an identical reward, so it is hard for the function to learn a meaningful policy.

I'm disappointed that NAF didn't work as I had expected, but I still learned an enormous amount about Q learning with continuous states and actions. It was also a good introduction to academic literature, which I feel is important is such a fast-moving field. Most of the cutting-edge components of machine learning need to be implemented from research papers like the one I used, and it encouraged me to push my own limits.

Improvement:

Solving NAF would be a great start!

For the final DNN Q learner, there are a number of improvements that might help with the issues that I had. Writing an independent scoring function would make all of the problems solvable, and by using some form of binning, it might be possible to solve the Pendulum-v0 environment. I would like to try DDPG to solve these environments, and there are several other journal papers that I came across in researching Normalized Advantage Functions the introduced various forms of actor-critic networks for reinforcement learning problems.