ESE650 Project 5: Policy Gradient Methods

Code Due Date: **04/06/2017 at 1:20pm** on Canvas, <pennkeyID>_project4.zip
Report Due Date: **04/07/2017 at 11:59pm** on Canvas, <pennkeyID>_project4.pdf

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there is an international frisbee shortage, so it is absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend...

In this project, you will implement several methods for dealing with your Frozen Lake problem. The situation can be represented via a Markov Decision Process (MDP) and a strategy for retrieving the frisbee can be obtained using value iteration (VI), policy iteration (PI), and policy gradient optimization (PGO). Once you learn to retrieve your frisbee, you can use your PGO implementation as well as a more sophisticated proximal policy optimization to learn how to balance a pendulum or to play Atari games.

Data Download: Now available at https://upenn.box.com/v/ese650pj5
Upload: on Canvas
(1) **Code** (<pennkeyID>_project5.zip)
(2) **Write-up** (<pennkeyID>_project5.pdf) : Write a project report including the following sections: Introduction, Problem Formulation, Technical Approach, Results and Discussion. Make sure your results include proper visualization of your gesture classification.

*Clearly presenting your approach in the form of report and presentation and having good algorithm performance are equally important.

Project Overview: The project consists of 3 parts (1_MDP, 2_PGO, and 3_ATARI) but you need to work on only two of those. The **first part (1_MDP) is required of everyone** and will be the main basis for grading. **Once you are done with 1_MDP, you can pick either 2_PGO or 3_ATARI depending on which one is more interesting and try to make some progress**. If you have no ideas for the **final project**, you can consider continuing to work on 2_PGO or 3_ATARI or both. Below is an outline of the three tasks.

1. **1_MDP**: Frozen Lake MDP. In this part, we will consider a discounted infinite-horizon MDP with finite state and action spaces, representing the Frozen Lake problem. You need to implement the following algorithms in **1_MDP/todo/todo.py**:
   - **Value Iteration** (vstar_backup): You will implement the Bellman backup value operator:

$$V_{k+1}(x) = \max_u \; r(x,u) + \gamma \sum_{x'} p(x'|x,u) V_k(x')$$

This update is called a **backup** because we are updating the state x based on possible future states x', i.e. we are propagating the value function *backwards in time*. The function is called vstart_backup because this update converges to the optimal value function V^*.

- **Policy Iteration** (policy_evaluation_v and policy_evaluation_q): Policy iteration initializes a policy π_0 and then performs the following two steps on the *n*th iteration:
  - Compute state-action value function: $Q^{\pi_{n-1}}(x,u)$
  - Compute new policy: $\pi_n(x) = \arg\max_u Q^{\pi_{n-1}}(x,u)$

  First, you will write a function (policy_evaluation_v) to compute the state-value function V^π for an arbitrary policy π. This can be done by solving the following linear system:

$$V^\pi(x) = r(x,\pi(x)) + \gamma \sum_{x'} p(x'|x,\pi(x)) V^\pi(x')$$

  Next, you will write a function (policy_evaluation_q) to compute the state-action value function Q^π, defined as:

$$Q^\pi(x,u) = r(x,u) + \gamma \sum_{x'} p(x'|x,u) V^\pi(x')$$

- **Policy Gradient Optimization** (softmax_policy_gradient, policy_gradient_step): Finally, you will implement a vanilla policy gradient method that parametrizes the policy π_θ via parameters θ, computes a gradient estimate $\hat{g} \approx \nabla_\theta E[\sum_t r_t(x_t, \pi(x_t))]$, and takes gradient ascent steps $\theta \leftarrow \theta + \varepsilon \hat{g}$. The policy for our discrete Frozen Lake MDP will be encoded by a matrix parameter $\theta =$ f_sa. The action probabilities are defined by exponentiating this matrix (element-wise) and then normalizing across the action dimension so that the probabilities add to , i.e. the *softmax* function. You need to compute the gradient of the function softmax_policy_checkfunc in util.py. To verify the gradient computation we will compute the gradient numerically as well using the **numdifftools** python module. Once you have a function that computes the policy gradient, you will write a function (policy_gradient_step) that simply updates the policy parameters. Note that the perplexity of your computed policy should go down to 1, corresponding to a deterministic policy. You should try this method on the larger frozen lake in frozen_lake.py as well.

2. **2_PGO**: Cart-Pole and Pendulum Balancing: In this task you are provided with a policy gradient method for solving a discrete-action cart-pole balancing task. Your task is to modify to work on a Pendulum balancing task, which has a continuous action space. Take a look at the README file for dependencies and make sure that the provided code (run_pgo.py) runs

on your computer. The implementation uses a neural network to encode the policy and outputs "logits" (i.e., log probabilities plus or minus a constant) that specify a discrete distribution. In contrast, for the pendulum task, your neural network should output the mean of a Gaussian distribution and a second parameter vector for the log standard deviation:

```
mean_na = dense(h2, ac_dim, weight_init=normc_initializer(0.1)) # Mean control output
logstd_a = tf.get_variable("logstdev", [ac_dim], initializer=tf.zeros_initializer) # Variance
sy_sampled_ac = YOUR_CODE_HERE
sy_logprob_n = YOUR_CODE_HERE
```

You should also compute differential entropy (replacing **sy_ent**) and KL-divergence (**sy_kl**) for the Gaussian distribution. The code computes several useful diagnostics, which you may find useful:

- **KL[policy before update || policy after update]**: Large spikes in KL divergence mean that the optimization is taking large steps and may become unstable.
- **Entropy of the policy**: If the entropy goes down too quickly, then you may not be exploring enough, while if it goes down too slowly, you probably won't reach optimal performance.
- **Explained variance of the value function**: If the function perfectly explains the returns, then it will be 1; if you get a negative result, then it is worse than predicting a constant.

The Pendulum problem is slightly harder and using a fixed stepsize does not work reliably. Instead, you should use an adaptive stepsize, adjusted based on the KL divergence between the new and old policy (code is provided). You can plot your results using the script plot_learning_curves.py. **Deliverables**:

- Show a plot with the pendulum converging to EpRewMean of at least **-300**. Include EpRewMean, KL, Entropy in your plots.
- Describe the hyperparameters used and how many timesteps your algorithm took to learn.

3. **3_ATARI**: A fun application of reinforcement learning is to learn policies for playing Atari games. OpenAI Gym provides a convenient interface for simulating a large range of problems including many games. As an extension to this project, install OpenAI Gym and explore the Atari application. To get you started we have provided the file run_atari.py which will simulate a policy of random actions. For example:

```
python run_atari.py --game space_invaders --image 0 --n_iter 1000
# Run Space Invaders with random actions for 1000 iterations while observing
the memory state of the game.

python run_atari.py --game pong --image 1 --n_iter 100
```

```
# Run Pong with random actions for 100 iterations while observing the image
output of the game.
```

This code should work with pong, breakout, enduro, beam_rider, space_invaders, seaquest, and qbert. Pick a game and experiment with the Atari environment. Can you learn a policy which does better than random? Some ideas are provided in the README file.

A large portion of this assignment was developed by John Schulman for the CS 294 Deep Reinforcement Learning course at UC Berkeley. We have his gracious permission to use this material for our class.