# Setup

Obtain a basic installation of OpenAI Gym as follows:

```
pip install pyglet # dependency, for visualization
git clone https://github.com/openai/gym
cd gym
pip install -e . # do a local installation
```

# 1. CEM Implementation

Implement the cross-entropy method. Starter code is provided below.

Test it out on the classic_control Environments in OpenAI Gym: CartPole-v0, Pendulum-v0, Acrobot-v0, MountainCar-v0.

CEM sometimes reduces the covariance too fast, causing premature convergence to a local optimum. A heuristic fix is to artificially increase it according to a schedule; see [SzitaLorincz06] for details.

## 2. CEM Analysis

Let's introduce some notation to help with the analysis of CEM. The optimization problem solved by CEM can be written as follows:

$$\underset{\theta}{\text{maximize}}\, E_\zeta[f(\theta, \zeta)]$$

Let $\psi \in \mathbb{R}^d$ parameterize the distribution over $\theta$ (e.g., via mean and variance of a Gaussian), so in each iteration of CEM, we collect $N$ samples $\theta_i \sim p_\psi(\cdot)$ and function values $f_1, f_2, \ldots, f_N$. Each iteration, we update $\psi$ by solving the subproblem

$$\text{maximize}_\psi \sum_{i \in \text{elite}} \log p_\psi(\theta_i).$$

(1) Explain why CEM does not always reach a local maximum of $E_\zeta[f(\theta, \zeta)]$, even in the limit of infinite samples ($N \to \infty$). (Hint: CEM seeks $\theta$ where the $\text{Var}_\zeta[f(\theta, \zeta)]$ is high.)

(2) Let's modify CEM so that each iteration, it solves a subproblem of the form

$$\text{maximize}_\psi \sum_{i=1}^{N} f_i \log p_\psi(\theta_i)$$

I.e., samples $\theta_i$ are weighted by the return $f_i$. (CEM corresponds to weighting by 1 or 0, depending on whether the sample is in the elite set). Show that the resulting algorithm does reach a local maximum of the objective, in an appropriate limit as $N \to \infty$.

# 3. Score Function Gradient Estimation

It's sometimes illuminating to see the functional form of the score function gradient estimators, for different distributions and parameters in those distributions.

This exercise asks you to determine if the score function gradient estimator can be used for several parameterized probability distributions, and if it can, compute $\nabla_\theta \log p(x|\theta)$.

Here's some notation:

$\text{Ber}(p; a, b)$ is the distribution that takes values $a, b$ with probility $(p, 1-p)$, respectively.
$\text{Poi}(\lambda)$ is the Poisson distribution with mean $\lambda$
$U(a, b)$ is the uniform distribution between $a, b$
$\text{Exp}(\lambda)$ is the exponential distribution, $p(x|\lambda) = \lambda e^{-\lambda x}$

Distributions:

1. $\text{Ber}(\theta, a, b)$
2. $\text{Ber}(p, \theta, b)$
3. $\text{Poi}(\theta)$
4. $N(\theta, \sigma^2)$
5. $N(\mu, \theta^2)$
6. $U(0, \theta)$
7. $\text{Exp}(\theta)$

# CEM Starter Code

```
import numpy as np
import gym
from gym.spaces import Discrete, Box

# ========================================================================
# Policies
# ========================================================================

class DeterministicDiscreteActionLinearPolicy(object):

    def __init__(self, theta, ob_space, ac_space):
        """
        dim_ob: dimension of observations
        n_actions: number of actions
        theta: flat vector of parameters
        """
        dim_ob = ob_space.shape[0]
        n_actions = ac_space.n
        assert len(theta) == (dim_ob + 1) * n_actions
        self.W = theta[0 : dim_ob * n_actions].reshape(dim_ob, n_actions)
        self.b = theta[dim_ob * n_actions : None].reshape(1, n_actions)

    def act(self, ob):
        """
        """
        y = ob.dot(self.W) + self.b
        a = y.argmax()
        return a

class DeterministicContinuousActionLinearPolicy(object):

    def __init__(self, theta, ob_space, ac_space):
        """
        dim_ob: dimension of observations
        dim_ac: dimension of action vector
        theta: flat vector of parameters
        """
```

```
        self.ac_space = ac_space
        dim_ob = ob_space.shape[0]
        dim_ac = ac_space.shape[0]
        assert len(theta) == (dim_ob + 1) * dim_ac
        self.W = theta[0 : dim_ob * dim_ac].reshape(dim_ob, dim_ac)
        self.b = theta[dim_ob * dim_ac : None]

    def act(self, ob):
        a = np.clip(ob.dot(self.W) + self.b, self.ac_space.low, self.ac_space.hi
gh)
        return a

def do_episode(policy, env, num_steps, render=False):
    total_rew = 0
    ob = env.reset()
    for t in range(num_steps):
        a = policy.act(ob)
        (ob, reward, done, _info) = env.step(a)
        total_rew += reward
        if render and t%3==0: env.render()
        if done: break
    return total_rew

env = None
def noisy_evaluation(theta):
    policy = make_policy(theta)
    rew = do_episode(policy, env, num_steps)
    return rew

def make_policy(theta):
    if isinstance(env.action_space, Discrete):
        return DeterministicDiscreteActionLinearPolicy(theta,
            env.observation_space, env.action_space)
    elif isinstance(env.action_space, Box):
        return DeterministicContinuousActionLinearPolicy(theta,
            env.observation_space, env.action_space)
    else:
```

```
        raise NotImplementedError

# Task settings:
env = gym.make('CartPole-v0') # Change as needed
num_steps = 500 # maximum length of episode
# Alg settings:
n_iter = 100 # number of iterations of CEM
batch_size = 25 # number of samples per batch
elite_frac = 0.2 # fraction of samples used as elite set

if isinstance(env.action_space, Discrete):
    dim_theta = (env.observation_space.shape[0]+1) * env.action_space.n
elif isinstance(env.action_space, Box):
    dim_theta = (env.observation_space.shape[0]+1) * env.action_space.shape[0]
else:
    raise NotImplementedError

# Initialize mean and standard deviation
theta_mean = np.zeros(dim_theta)
theta_std = np.ones(dim_theta)

# Now, for the algorithm
for iteration in xrange(n_iter):
    # Sample parameter vectors
    thetas = YOUR_CODE_HERE
    rewards = [noisy_evaluation(theta) for theta in thetas]
    # Get elite parameters
    n_elite = int(batch_size * elite_frac)
    elite_inds = np.argsort(rewards)[batch_size - n_elite:batch_size]
    elite_thetas = [thetas[i] for i in elite_inds]
    # Update theta_mean, theta_std
    theta_mean = YOUR_CODE_HERE
    theta_std = YOUR_CODE_HERE
    print "iteration %i. mean f: %8.3g. max f: %8.3g"%(iteration, np.mean(reward
s), np.max(rewards))
    do_episode(make_policy(theta_mean), env, num_steps, render=True)
```

[SzitaLorincz06] Szita and Lorincz, *Learning Tetris with the Noisy Cross-Entropy Method*.