Robot Learning

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter Semester 2016, Homework 4 (65 points + 10 bonus)
Prof. Dr. J. Peters, F. Veiga, S. Parisi

Due Date: Thursday, 9 February 2017 (before the lecture)

Problem 4.1 Expectation-Maximization [30 Points + 5 Bonus]

In this exercise your task is to control a 2-DoF planar robot to throw a ball at a specific target. You will use an episodic setup, where you first specify the parameters of the policy, evaluate them on the simulated system, and obtain a reward. The robot will be controlled with the Dynamic Motor Primitives (DMPs). The goal state of the DMPs is pre-specified and the weights of the DMP $\theta_i, i = 1 \ldots 10$ are the open parameters of the control policy. Each DoF of the robot is controlled with a DMP with five basis functions. The ball is mounted at the end-effector of the robot and gets automatically released at time step t_{rel}. We define a stochastic distribution $\pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with $\omega = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$.
Your task it to update the parameters $\boldsymbol{\omega}$ of the policy using EM to maximize the expected return. In this exercises we will not modify the low-level control policy (the DMPs) of the robot.

A template for the simulation of the 2-DoF planar robot and plotting functions can be found at the course website. For the programming exercises, attach snippets of your code.

a) Analytical Derivation [5 Points]

Using the weighted ML estimate,

$$\boldsymbol{\omega}_{k+1} = \arg\max_{\boldsymbol{\omega}} \{\sum_i w^{[i]} \log \pi\left(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\right)\}, \tag{1}$$

derive analytically the update rule for our policy $\pi(\boldsymbol{\theta}|\boldsymbol{\omega})$ for the mean $\boldsymbol{\mu}$. Show your derivations.

Name, Vorname: ______________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴

b) Programming Exercise [15 Points]

Implement the EM algorithm using the provided framework. Your goal is to throw the ball at $\boldsymbol{x} = [2, 1]m$ at time $t = 100$. Use the weighted Maximum Likelihood (ML) solutions for updating the parameters of the policy. For the mean, use the equation derived at the previous exercise. For the covariance, the update rule is

$$\boldsymbol{\Sigma}_{\text{new}} = \frac{\sum_i w^{[i]} \left(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}_{\text{new}}\right)\left(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}_{\text{new}}\right)^T}{\sum_i w^{[i]}}. \tag{2}$$

To calculate the weights $\boldsymbol{w}_i$ for each sample, transform the returned rewards by

$$\boldsymbol{w}^{[i]} = \exp\left(\left(\boldsymbol{R}^{[i]}(\boldsymbol{\theta}) - \max(\boldsymbol{R})\right)\beta\right) \tag{3}$$

where the vector $\boldsymbol{R} \in \mathbb{R}^{N\times 1}$ is constructed from the rewards of the N samples. The parameter β is set to

$$\beta = \frac{\lambda}{\max(\boldsymbol{R}) - \min(\boldsymbol{R})}, \tag{4}$$

where $\lambda = 7$. Start with the initial policy

$$\pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \mathcal{N}\left(\boldsymbol{\theta}|\boldsymbol{0}, 10^6\boldsymbol{I}\right) \tag{5}$$

and calculate the average reward at each iteration. Iterate until convergence or for a maximum number of 100 iterations. We assume that convergence is achieved when the average return does not change much at every iteration, i.e.,

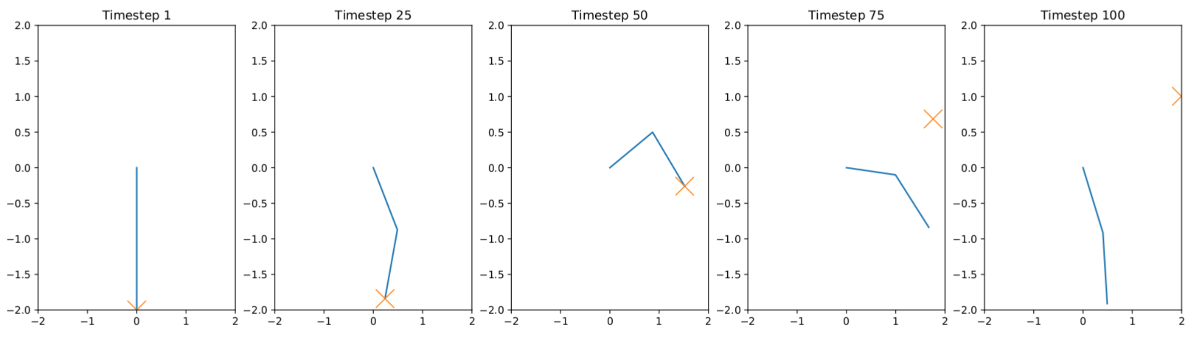
$$| < \boldsymbol{R}_{i-1} > - < \boldsymbol{R}_i > | < 10^{-3},$$

where $< \cdot >$ denotes the average operator. At each iteration use $N = 25$ samples. In order to avoid getting stuck to a local optimum, we force the algorithm to explore a bit more by adding a regularization factor to the covariance of the policy,

$$\boldsymbol{\Sigma}'_{\text{new}} = \boldsymbol{\Sigma}_{\text{new}} + \boldsymbol{I}. \tag{6}$$

What is the average return of the final policy? Use `animate_fig` from `Pend2dBallThrowDMP.py` to show how the final policy of the robot looks like (attach all five screenshots).

The average return of the final policy is: -39.4430848515 .



```
env = Pend2dBallThrowDMP()
```

Name, Vorname: ____________________ Matrikelnummer: ⎵⎵⎵⎵⎵⎵⎵

```
def learn(lambda_var):
        # Do your learning
        # For example, let initialize the distribution...
        Mu_w = np.zeros(numDim)
        Sigma_w = np.eye(numDim) * 1e6
        # ... then draw a sample and simulate an episode
        sample = mkSample(Mu_w, Sigma_w)
        reward = rewardSample(sample)
        beta = mkBeta(reward)
        weights = mkWeight(reward, beta)
        Mu_w = mkWeightedMean(sample, weights, reward)
        Sigma_w = mkWeightedCovariance(sample, weights ,Mu_w)
        mu = np.copy(Mu_w)
        sig = np.copy(Sigma_w)
        i = 0
        convergence = False
        av_rewards = np.zeros(maxIter)
        while(i < maxIter and not convergence):
                if i is maxIter-1: print "convergence not reached"
                old_reward = reward
                sample = mkSample(mu, sig)
                reward = rewardSample(sample)
                beta = mkBeta(reward)
                weights = mkWeight(reward, beta)
                mu = mkWeightedMean(sample, weights, reward)
                sig = mkWeightedCovariance(sample, weights ,mu) + np.eye(numDim)
                print np.mean(reward)
                av_rewards[i]=np.mean(reward)
                convergence= True if meanRewardDiff(reward, old_reward) < 1e-3 else Fals
                i += 1
        return av_rewards

def mkBeta(reward):
        return lambda_var / (np.amax(reward) - np.amin(reward))

def mkWeight(reward, beta):
        weight = np.zeros(numSamples)
        for current in range(numSamples):
                weight[current] = np.exp((reward[current] - np.max(reward)) * beta)
        return weight

def mkSample(mu, sig):
    sample = np.zeros((numDim, numSamples))
    for i in range(numSamples):
        sample[:,i] = np.random.multivariate_normal(mu, sig)
    return sample

def rewardSample(sample):
    reward = np.zeros(numSamples)
    for i in range(numSamples):
        reward[i] = env.getReward(sample[:,i])
    return reward
```

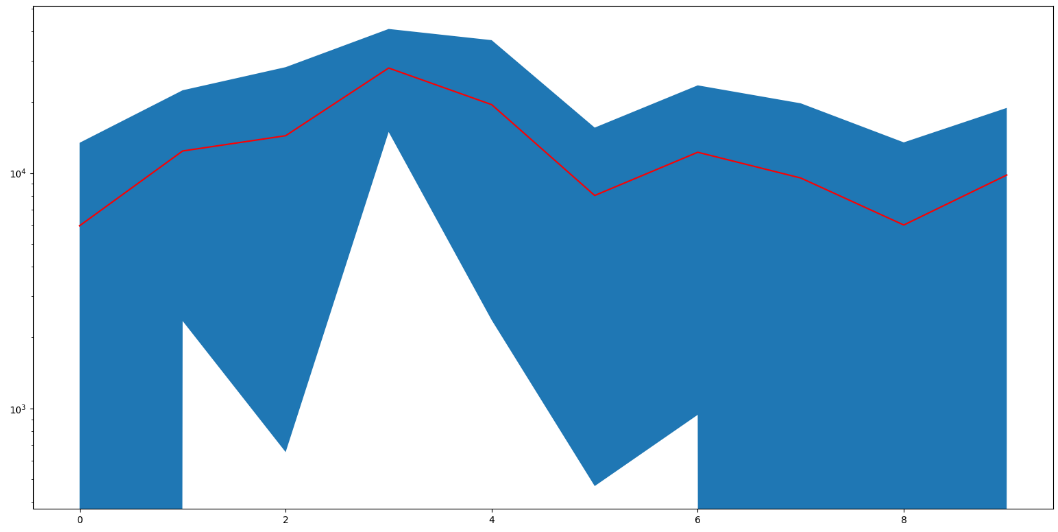
Name, Vorname: ______________________ Matrikelnummer: ␣␣␣␣␣␣␣

```
def meanRewardDiff(rew1, rew2):
    return np.absolute(np.mean(rew1) - np.mean(rew2));

def mkWeightedMean(sample, weight,reward):
    sumOfWeights = 0
    sumOfWeightedSamples = 0
    for i in range(numSamples):
        sumOfWeights += weight[i]
        sumOfWeightedSamples += weight[i] * sample[:,i]
    return sumOfWeightedSamples / sumOfWeights

def mkWeightedCovariance(sample, weights,mean):
    sumOfWeights = 0
    sumOfWeightedSample = 0
    for i in range(numSamples):
        sumOfWeights += weights[i]
        sumOfWeightedSample += weights[i] * (np.outer((sample[:,i] - mean), np.transpose(s
    return sumOfWeightedSample / sumOfWeights
```

Name, Vorname: ____________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴

c) Programming Exercise [5 Points]

Repeat the learning 10 times and plot the mean of the average return of all runs with 95% confidence. Use the logarithmic scale for your plot.



Iterations on x-axis and negative reward on y-axis.

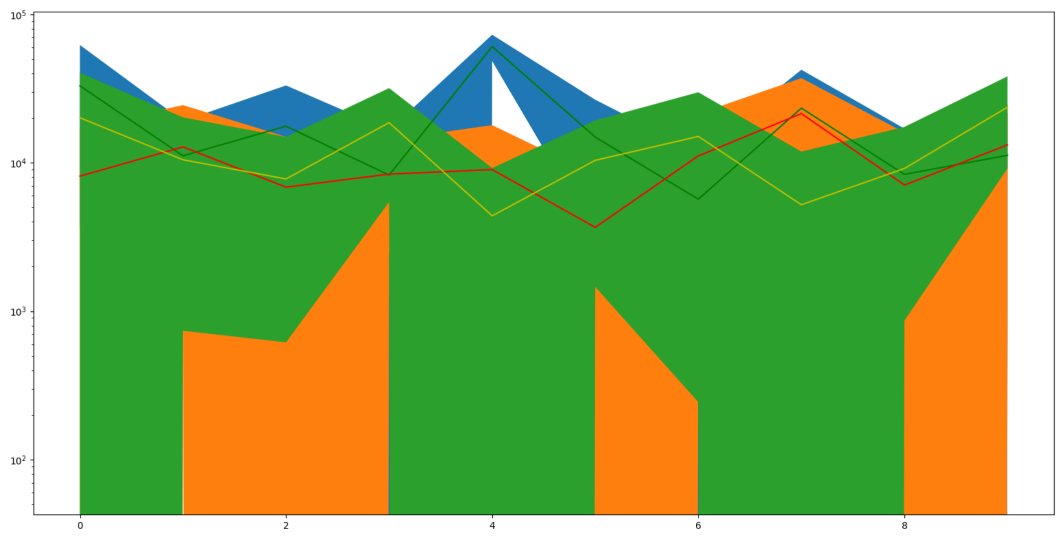
```
average_rewards7 = np.zeros((10,100))
for i in range(10):
    average_rewards7[i,:] = learn(7)

average_rewards3 = np.zeros((10,100))
for i in range(10):
    average_rewards3[i,:] = learn(3)

average_rewards25 = np.zeros((10,100))
for i in range(10):
    average_rewards25[i,:] = learn(25)

def mean_confidence_interval(data, confidence=0.95):
        a = 1.0 * np.array(data)
        n = len(a)
        m,se = np.mean(a), scipy.stats.sem(a)
        h = se * sp.stats.t._ppf((1+confidence)/2.,n-1)
        return m, h

a3, a7, a25 = [np.ones((10,2)) for i in range(3)]
for i in range(10):
    a = average_rewards3[i]
    a3[i,:] = np.array(mean_confidence_interval(a[a != 0]))
```

```
for i in range(10):
    b = average_rewards7[i]
    a7[i,:] = np.array(mean_confidence_interval(b[b != 0]))

for i in range(10):
    c = average_rewards25[i]
    a25[i,:] = np.array(mean_confidence_interval(c[c != 0]))
```

d) Tuning The Temperature [5 Points]

Repeat the previous exercise with temperature $\lambda = 25$ and $\lambda = 3$. Plot in one figure the mean of the average returns for all $\lambda = 3$, $\lambda = 7$, $\lambda = 25$ with 95% confidence. How does the value of λ affects the convergence of the algorithm in theory? Use the logarithmic scale for your plot.

The λ coefficient (temperature) in theory affects the convergence kind of like a learning rate. So, if you choose a bigger temperature the algorithm converges faster. However, it could just stuck in a local optima, not a global



one.

e) Optimal Temperature [5 Bonus Points]

The problem of choosing λ falls under a more general RL problem. Which one? Which algorithm would allow you to automatically select the optimal λ? Explain it with your words. Does it still have hyperparameters to be set?

It falls under the problem of hyperparameter tuning.

Robot Learning - Homework 4

Name, Vorname: ______________________ Matrikelnummer: ⊔⊔⊔⊔⊔⊔⊔

Problem 4.2 Policy Gradient [25 Points + 5 Bonus]

In this exercise, you are going to solve the same task of the previous exercise but using policy gradient. For the programming exercises, you are allowed to reuse part of the EM code. Attach snippets of your code.

a) Analytical Derivation [5 Points]

You have a Gaussian policy with diagonal covariance, i.e., $\pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \mathcal{N}\left(\boldsymbol{\mu}, \text{diag}\left(\boldsymbol{\sigma}^2\right)\right)$, where $\boldsymbol{\omega} = [\boldsymbol{\mu}, \boldsymbol{\sigma}]$. Compute analytically the gradient of the logarithm of the policy with respect to the parameters $\boldsymbol{\omega}$, i.e., $\nabla_{\boldsymbol{\omega}} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega})$.

b) Programming Exercise [5 Points]

Consider the same robotic task as in the previous exercise and this time solve it with policy gradient. Use an initial mean of $\boldsymbol{\mu}_0 = [0 \ldots 0]$ and a fixed $\boldsymbol{\sigma} = \text{diag}([10 \ldots 10])$ (i.e., do not update $\boldsymbol{\sigma}$). Set the learning rate to $\alpha = 0.1$ and use the same hyperparameters as before (25 episodes sampled for each iteration and max 100 iterations).
Repeat the learning 10 times and plot the mean of the average return of all runs with 95% confidence. Use the logarithmic scale for your plot. Comment your results.

c) A Little Trick [5 Points]

How would you improve the above implementation? (Beside using a smaller or adaptive learning rate, or using the natural gradient). What is the theory behind this "trick"? Repeat the learning and discuss the results.

One could subtract a baseline. This yields to a function, with less variance, which would be easier to investigate.

d) Learning Rate [5 Points]

Repeat the optimization changing the learning rate to $\alpha = 0.4$ and $\alpha = 0.2$ (keep the trick of the previous exercise). Plot in one figure the mean of the average returns for all α with 95% confidence. How does the value of α affect the convergence of the algorithm? Use the logarithmic scale for your plot.

e) Variable Variance [5 Points]

Try to improve the optimization process by learning also the variance $\boldsymbol{\sigma}$. Is it easier or harder to learn also the variance? Why?
Without using the natural gradient, tune the learning rate and the initial variance to achieve better results. If you think it is necessary, you can impose a lower bound to avoid that the variance collapses to infinitely small values (e.g., if $\sigma(i) < \sigma_{\text{lower}}$ then $\sigma(i) = \sigma_{\text{lower}}$). In one figure, plot the learning trend with confidence interval as done before and compare it to the one achieved with $\alpha = 0.4$.

Name, Vorname: ______________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴

f) Natural Gradient [5 Bonus Points]

Write down the equation of the natural gradient. What is the theory behind it? Is it always easy to use?

$$\delta_w^{NG} J = G(w)^{-1} \delta_w J \tag{8}$$

It uses the Fisher Matrix, and is based on the idea of comparing the resulting distributions, while updating the parameters, using the KL-divergence to assure that the steps made by the NG doesn't affect the distribution too much. It can only be used working on distribution.

Problem 4.3 Reinforcement Learning [10 Points]

a) RL Exploration Strategies [10 Points]

Discuss the two different exploration strategies applicable to RL.

The two different exploration strategies in Reinforcement Learning consist of exploitation and exploration. To exploit means to make the best decision, given the current information wheras exploration is gathering more information. Given a set of actions in a given state, we can choose from, exploitation would mean to choose the best action suited to the state, thus the action resulting in the biggest reward. Exploration here, would mean to e.g. choose random, uniform on the actions to discover new information, through maybe a better return can be obtained.