Robot Learning

Winter Semester 2017/2018, Homework 4

Prof. Dr. J. Peters, D. Tanneberg, M. Ewerton

TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 60 + 10 bonus
Due date: Wednesday, 31 January 2018 (before the lecture)

Name, Surname, ID Number Markus Lamprecht, 2424163 Moritz Knaust, 2430801

Problem 4.1 Expectation-Maximization [25 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 $\theta_i, i = 1 \dots 10$ 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 $\boldsymbol{\omega} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$.
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.

$$\frac{\partial \boldsymbol{w}_{k+1}}{\partial \boldsymbol{\mu}} = 0 \tag{7}$$

$$\pi\left(\theta^{[i]}; w\right) = N(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} exp\left(-\frac{1}{2}(\boldsymbol{\mu} - \boldsymbol{\theta})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} - \boldsymbol{\theta})\right) \tag{8}$$

$$log\left(\pi\left(\theta^{[i]}; w\right)\right) = -log\left(\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}\right) - \frac{1}{2}(\boldsymbol{\mu} - \boldsymbol{\theta})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} - \boldsymbol{\theta}) \tag{9}$$

$$\frac{\partial}{\partial \boldsymbol{\mu}} log\left(\pi\left(\theta^{[i]}; w\right)\right) = -\frac{1}{2}\left(2\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} - 2\boldsymbol{\Sigma}^{-1}\boldsymbol{\theta}\right) \tag{10}$$

$$\frac{\partial \boldsymbol{w}_{k+1}}{\partial \boldsymbol{\mu}} = 0 \rightarrow \sum_i w^{[i]} \boldsymbol{\Sigma}^{-1} \mu_i = \sum_i w^{[i]} \boldsymbol{\Sigma}^{-1} \theta_i \rightarrow \mu_i = \frac{\sum_i w^{[i]} \theta_i}{\sum_i w^{[i]}} \tag{11}$$

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

$$\boldsymbol{\Sigma}_{\text{new}} = \frac{\sum_i w^{[i]} \left(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}_{\text{new}}\right)\left(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}_{\text{new}}\right)^T}{\sum_i w^{[i]}}. \tag{12}$$

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

$$\boldsymbol{w}^{[i]} = \exp\left(\left(\boldsymbol{R}^{[i]}(\boldsymbol{\theta}) - \max(\boldsymbol{R})\right)\beta\right) \tag{13}$$

and normalize them, 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

$$\beta = \frac{\lambda}{\max(\boldsymbol{R}) - \min(\boldsymbol{R})}, \tag{14}$$

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

$$\pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \mathcal{N}\left(\boldsymbol{\theta}|\boldsymbol{0}, 10^6\boldsymbol{I}\right) \tag{15}$$

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.,

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

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,

$$\boldsymbol{\Sigma}'_{\text{new}} = \boldsymbol{\Sigma}_{\text{new}} + \boldsymbol{I}. \tag{16}$$

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).

Average Return of the final policy: -38.89

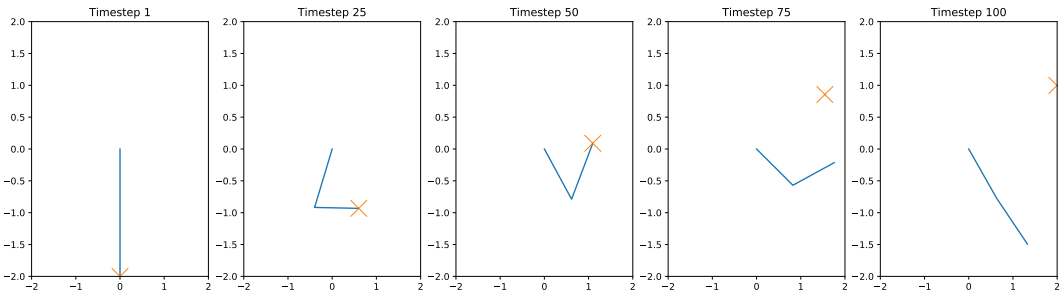

Figure 2: Policy of the Robot's arm (blue) and the Ball's position (yellow cross)

```
numDim = 10
numSamples = 25
maxIter = 100
numTrials = 10

def EM(lam = 7):
```

```
    # For example, let initialize the distribution...
    Mu_w = np.zeros(numDim)
    Sigma_w = np.eye(numDim) * 1e6

    R_mean_old = 0
    # Do your learning
    R_avg = np.zeros(maxIter)
    for i in xrange(maxIter):
        # reward
        R = np.zeros((numSamples, 1))
        theta = np.zeros((numSamples, numDim))
        for j in xrange(numSamples):
            theta[j, :] = np.random.multivariate_normal(Mu_w, Sigma_w)
            R[j] = env.getReward(theta[j, :])
        R_avg[i] = np.mean(R)

        # weights
        beta = lam / (np.max(R) - np.min(R))
        w = np.exp((R - np.max(R)) * beta)

        Mu_w = np.sum(w * theta, axis=0) / np.sum(w)

        temp = np.zeros((numDim, numDim))

        # for k in xrange(numDim):
        #     diff = theta - Mu_w[k]

        for j in xrange(numSamples):
            diff = (theta[j, :] - Mu_w).reshape((numDim, 1))
            temp += w[j] * np.dot(diff, diff.T)
        Sigma_w = temp / np.sum(w) + np.eye(numDim)

        if i>1:
            if np.abs(R_avg[i] - R_avg[i-1]) < 1e-3:
                break
    print 'end'
    return Mu_w, Sigma_w, R_avg
```

Listing 2: EM algorithm

c) Tuning The Temperature [5 Points]

Repeat the learning 10 times and plot the mean of the average return of all runs with 95% confidence for temperatures $\lambda = 25$, $\lambda = 7$ and $\lambda = 3$. How does the value of λ affect the convergence of the algorithm in theory and what do you observe in the results? Use the logarithmic scale for your plot.

In the results we observe, that for $\lambda = 25$ the average return converges faster than for smaller lambdas. Moreover it should be noted that for $\lambda = 7$ we have the smallest average return at episode=100.

In Theory: With the temperature λ one can select how the maximum of the weights is selected:

$$\lim_{\lambda \to \infty} w^{[i]} : \text{weights are selected deteministically} \tag{19}$$

$$\lim_{\lambda \to 0} w^{[i]} : \text{weights are selected stochastically} \tag{20}$$

In the plots we can prove that for small lambdas the convergence is slower because the weights are selected stochastically. However the mean of the average reward for the last episode should be the smallest if the lambda is very small. This theoretical fact differs from the plot above.

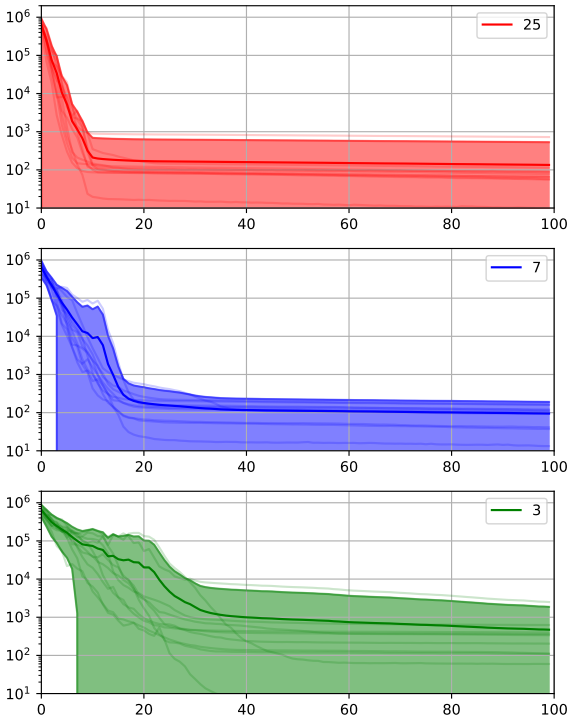

Figure 3: The average return of different λ over the episodes of the EM-Algorithm.

d) 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?

RL problem: Policy Search Methods using Policy Gradients finding the right weights scaling factor.
Algorithm that selects automatically the right lambda: Reward Weighted Regression.

The Reward Weighted Regression
is an algorithm that generalizes the EM algorithm. This algorithm reduces the problem of learning with immediate rewards to a reward-weighted regression problem with an adaptive, integrated reward transformation for faster convergence. The resulting algorithm is efficient, learns smoothly without dangerous jumps in solution space, and works well in applications of complex high degreeof-freedom robots.

Hyperparameters:
A transformed Hyperparameter still has to be set in RWR. However this parameter is automatically updated. With this parameter the convergence speed can be selected.

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., $\nabla_{\boldsymbol{\omega}} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega})$. (Hint: consider the properties of the diagonal covariance matrix.)

$$\nabla_{\boldsymbol{\omega}} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \nabla_{\boldsymbol{\omega}} \left(-log\left(\sqrt{(2\pi)^k} |diag\left(\sigma^2\right)| \right) - \frac{1}{2} (\theta - \mu)^T \, diag\left(\sigma^2\right)^{-1} (\theta - \mu) \right) \tag{28}$$

$$\nabla_{\boldsymbol{\omega}} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \nabla_{\boldsymbol{\omega}} \left(-log\left(\sqrt{(2\pi)^k} \left(\sigma^{2k}\right) \right) - \frac{1}{2\sigma^2} \sum_{i=1}^{k} (\theta_i - \mu_i)^2 \right) \tag{29}$$

$$\nabla_{\boldsymbol{\omega}} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \begin{bmatrix} \frac{\partial}{\partial \mu} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) \\ \frac{\partial}{\partial \sigma} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) \end{bmatrix} \tag{30}$$

$$\frac{\partial}{\partial \mu} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \frac{1}{\sigma^2} \sum_{i=1}^{k} (\theta_i - \mu_i) \tag{31}$$

$$\frac{\partial}{\partial \sigma} \log \pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = -\frac{2k}{\sigma} + \frac{1}{\sigma^3} \sum_{i=1}^{k} (\theta_i - \mu_i)^2 \tag{32}$$

Note that:

$$\mathcal{N}\left(\boldsymbol{\mu}, \text{diag}\left(\boldsymbol{\sigma}^2\right)\right) = \frac{1}{\sqrt{(2\pi)^k} |diag\,(\sigma^2)|} exp\left(-0.5\,(\theta - \mu)^T \, diag\left(\sigma^2\right)^{-1} (\theta - \mu) \right) \tag{33}$$

$$(\theta - \mu)^T \, diag\left(\sigma^2\right)^{-1} (\theta - \mu) = \frac{1}{\sigma^2} \sum_{i=1}^{k} (\theta_i - \mu_i)^2 \tag{34}$$

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 $\boldsymbol{\mu}_0 = [0 \dots 0]$ and a fixed $\boldsymbol{\sigma} = \text{diag}([10 \dots 10])$ (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.

$$\omega_{k+1} = \omega_k + \alpha \nabla_\omega J_\omega \tag{38}$$

Update rule for θ:

$$\sigma_{k+1} = \sigma_k + \alpha \frac{\partial}{\partial \sigma} J(\sigma_k) \tag{39}$$

Update rule for μ:

$$\mu_{k+1} = \mu_k + \alpha \frac{\partial}{\partial \mu} J(\mu_k) \tag{40}$$

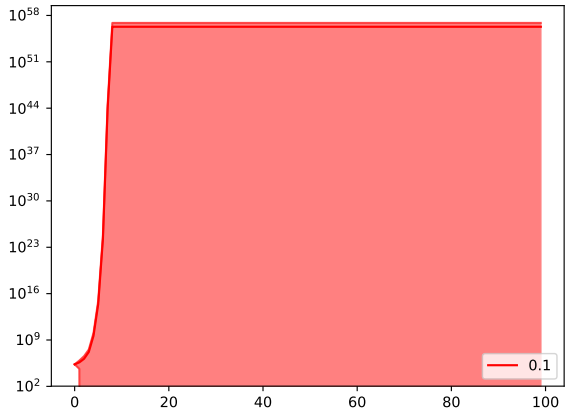

Figure 5: The average return over the episodes for $\lambda = 0.1$

You can see, that the reward explodes and one do not get an stable behavior. This depends on the high variance of the gradient.

```
numDim = 10
numSamples = 25
maxIter = 100
numTrials = 10

# Do your learning
def PolicyGradient(alpha = 0.1):
    # For example, let initialize the distribution...
    Mu = np.zeros(numDim)
    Sigma = np.eye(numDim) * 10

    # Do your learning
    R_avg = np.zeros(maxIter)
    for i in xrange(maxIter):
        # reward
        R = np.zeros((numSamples, 1))
        theta = np.zeros((numSamples, numDim))
        for j in xrange(numSamples):
            theta[j, :] = np.random.multivariate_normal(Mu, Sigma)
```

```
            R[j] = env.getReward(theta[j, :])
        R_avg[i] = np.mean(R)
        #print 'R_mean_{}: {}'.format(i, R_avg)

        temp = np.zeros(theta.shape)
        for j in xrange(numSamples):
            temp[j, :] = np.dot(np.linalg.inv(Sigma), theta[j] - Mu) * R[j]
                # gradient
        grad_mu = 1./numSamples * np.sum(temp, axis=0)

        Mu = Mu + alpha*grad_mu

        if i>1:
            if np.abs(R_avg[i] - R_avg[i-1]) < 1e-3:
                break
    print 'R_avg_end: {}'.format(R_avg[-1])
    return Mu, Sigma, R_avg
```

Listing 4: policy gradient algorithm

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.

$$\nabla_\omega J_\omega \sum_{i=1}^{N} \nabla_\omega log\, \pi(\theta;\omega)(R_i - b) \tag{42}$$

The baseline has to be subtracted. Thereby the possibly high gradient of the variance can be reduced. It is still unbiased. As baseline we use the average reward.

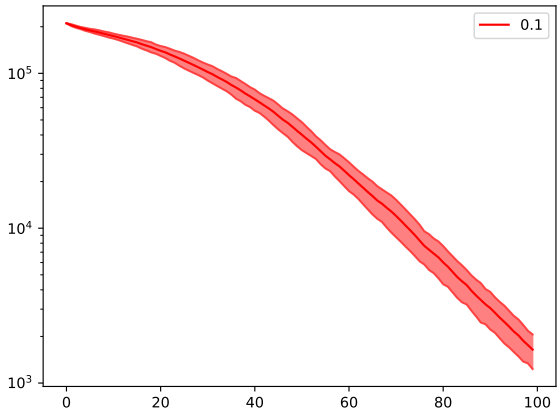

Figure 7: The average return over the episodes of for $\lambda = 0.1$ with baseline subtracted of the PG algorithm

```
numDim = 10
numSamples = 25
maxIter = 100
numTrials = 10

# Do your learning
def PolicyGradient(alpha = 0.1):
    ...
    for i in xrange(maxIter):
        ...

        temp = np.zeros(theta.shape)
        for j in xrange(numSamples):
            temp[j, :] = np.dot(np.linalg.inv(Sigma), theta[j] - Mu) * (R[j] - R_avg[i
    ])
                # gradient
        grad_mu = 1./numSamples * np.sum(temp, axis=0)

        Mu = Mu + alpha*grad_mu
        ...
    print 'R_avg_end: {}'.format(R_avg[-1])
    return Mu, Sigma, R_avg
```

Listing 6: policy gradient algorithm with subtracted baseline

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.

The higher the α the faster the algorithm converges.

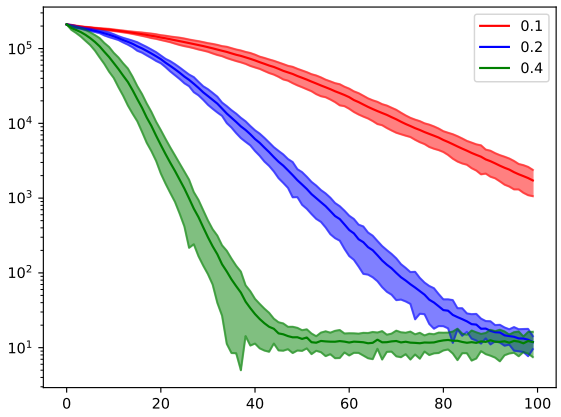

Figure 9: The average return over the episodes for different learning rates α

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 process 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 $\alpha = 0.4$ before.

Now use:

$$\sigma_{k+1} = \sigma_k + \alpha \frac{\partial}{\partial \omega} J(\sigma_k) \tag{44}$$

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?

The Natural Gradient: (with G the Fischer Information Matrix)

$$\nabla_{\omega} J = G(\omega)^{-1} \nabla_{\omega} J \tag{46}$$

Theory of the Natural Gradient:
KL divergence is a measure of how close a distribution is to another distribution.This brings us to the natural gradient. If we blindly update our network in the direction of its gradients, there are no guarantees the distribution of the new network will be similar to the old one.

To fix this, we first consider all combinations of parameters that result in a new network a constant KL divergence away from the old network. This constant value can be viewed as the step size or learning rate. Out of all these possible combinations, we choose the one that minimizes our loss function.

Basically, we're adding in a constraint to our update, that the new network will behave relatively similar to our old one. Our step size corresponds directly to the actual distribution of the network, not it's parameter's values.

This comes with the added benefit of a more stable learning process. Especially when we're updating using a randomly sampled batch, some outliers may make drastic changes to a network's parameters. With natural gradient descent, a single update can't make too much of an impact.

Note that: Replacing a sigmoid activation with a tanh function would change the standard gradient, but not the natural gradient.

Is it always easy to use: No actually NG is much more computationally expensive compared to the simple gradient. Moreover there are two types of KL:

- Moment projection
- Information projection (is not unique selects the mode!)

It is not so easy to select which KL should be used.

Note that KL can be approximated by the FIM which captures information how the single parameters influence the distribution.

Problem 4.3 Reinforcement Learning [10 Points]

a) RL Exploration Strategies [10 Points]

In which spaces can you perform exploration in RL? Discuss the two exploration strategies applicable to RL.

Exploration can be performed in Action Space and Parameter space.

Parameter Space:
In Episode Based Policy Search a distribution is learned over the parameters of the low level control policy. Exploration in parameter space allows more sophisticated strategies and is often very efficient for a small amount of parameters.

A problem is the high variance of the returns, because one add different random variables. If one has a too high variance, the learned policy can become instable. Too avoid this, you can use the baseline trick for example.

An advantage is, that you do not have too know the structure of the optimization problem. So it is an "Black-Box Optimizer".

Action Space:
In the Episode based Policy Search you learn an full movement as episode and then get the reward for it. In Step-based Policy Search one decomposite the episode in single timesteps. So we get the reward for single state-actions pairs. This reward has a lower variance.

In Step Based Policy Search there is less variance in quality assessment. Moreover as the exploration happens in action space it is less likely to create unstable policies.