Learning Robots Exercise 5

TECHNISCHE UNIVERSITÄT DARMSTADT

Dominik Marino - 2468378, Victor Jimenez - 2491031, Daniel Piendl - 2586991
March 2, 2021

Contents

1 Task 1 - Model Learning

1.1 Problem Statement

The given problem is a supervised learning problem, which can solved by a simple linear regression. The model of the inverse dynamics is given as

$$u_i = \phi(q, \dot{q}, \ddot{q})^T \theta_i, \qquad \text{where } \phi \text{ is a nonlinear function and, } \theta \text{ is a weight vector}$$

For the learning process we have to measure the data. In the given file "*spinbotdata.txt*" already collects the measured types of data to learn the model direcly from the system.

- Input data: $q_1, q_2, q_3, \dot{q}_1, \dot{q}_1, \dot{q}_2, \dot{q}_3, \ddot{q}_1, \ddot{q}_2, \ddot{q}_3$
- Output data: u_1, u_2, u_3

1.2 Assumptions

The most important assumption for this model is, that this data does not consists completely of reality. The goal of a supervised learning problem is to approximating the functions. Discovering all forces and torques that affects the joints is not possible. Another problem can be approximating velocities and accelerations. This is important for approximating data and getting the right data. To sample data for the trajectories is to randomly choose point-to-point movements. Otherwise an inaccurate dynamics model can lead in a bad behavior and instability based on physics in the real world.

Also different supervised learning models makes different assumption and how predicting the output form the model function. Jan Peters and Duy Nguyen-Toung makes similar assumption when it comes to learning models. They differentiate global, local and mixured techniques:

> "*Global regression techniques model the underlying function f using all observed data to construct a single global prediction model [50]. In contrast to global methods, the local regression estimates the underlying function f within a local neighborhood around a query input point. Beyond the local and global types of model learning, there are also approaches which combine both ideas. An example of such hybrid approaches is the mixture of experts [55,109].*[1]"

1.3 Features and Parameters

Our model has the shape of:

$$u_i = \phi(q_1, q_2, q_3)^T \theta_i$$

- for i = 1, 2, 3 where ϕ_i are the features and θ_i are the parameters

u_1, u_2, u_3 is given by

$$\begin{aligned} u_1 &= (m_1 + m_2)(\ddot{q}_1 + g) \\ u_2 &= m_2(2\dot{q}_3\dot{q}_2 q_3 + q_3^2\ddot{q}_2) \\ u_3 &= m_2(\ddot{q}_3 - q_3\dot{q}_2^2) \end{aligned}$$

Now we can rewrite the formula

$$u_i = \underbrace{\begin{bmatrix} \ddot{q}_1 & 0 & 1 \\ 0 & 2\dot{q}_3\dot{q}_2 q + q_3^2\ddot{q}_2 & 0 \\ 0 & \ddot{q}_3 - q_3\dot{q}_2^2 & 0 \end{bmatrix}}_{\phi(q_1,q_2,q_3)^T} \underbrace{\begin{bmatrix} m_1 + m_2 \\ m_2 \\ gm_1 + gm_2 \end{bmatrix}}_{\theta_i} \tag{1}$$

[1]Nguyen-Tuong, D., Peters, J. Model learning for robot control: a survey. Cogn Process 12, 319–340 (2011). https://doi.org/10.1007/s10339-011-0404-1

1.4 Learning the Parameters

$$\theta = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{u}$$

where $\theta \in \mathbb{R}^3$, $\mathbf{\Phi} \in \mathbb{R}^{3n \times 3}$ and $\mathbf{u} \in \mathbb{R}^{3n}$. The value obtained from the Least Squares Method is $\theta = [1.58, 1.65, 15.1]^T$.

```
data = list()

with open('spinbotdata.txt', 'r') as f:
    Lines = f.readlines()
    for line in Lines:
        data.append([float(x) for x in line.split()])

q = np.asarray(data[0:3])
n = q.shape[-1]
dq =  np.asarray(data[3:6])
ddq = np.asarray(data[6:9])
u = np.zeros((3*n, 1))
u[0::3,0] =  np.asarray(data[9])
u[1::3,0] =  np.asarray(data[10])
u[2::3,0] =  np.asarray(data[11])


phi = np.zeros((3*n, 3))
phi[0::3, 0] = ddq[0, :]
phi[0::3, -1] = 1
phi[1::3, 1] = 2*dq[2, :]*dq[1, :]*q[2, :] + q[2, :]**2 * ddq[1, :]
phi[2::3, 1] = ddq[2, :] - q[2, :]*dq[1, :]**2

pseudo_inv = np.linalg.inv(phi.T @ phi)
theta = pseudo_inv @ phi.T @ u

u_approx = phi@theta
```

Listing 1: Finite Horizon

1.5 Recovering Model Information

It is possible to recover the values of the masses from the previous learned parameter vector θ. Using the relationship in equation 1 one can compute the values, meaning that $m_2 = 1.65$, following $m_1 = 1.58 - m_2 = -0.07$ and $g = 9.57$. The robot hasn't learned a plausible model because the masses in the physical model cannot have negative values, but in this case m_1 violates this constraint.

Using the true value of g will not solve this problem, because even thought the model would be using the accurate value of the gravity constant, this parameter only influences the least squares method as a "bias" term. Writing out the least squares formula for the first torque yields

$$J = \sum_{i=0}^{n} (\ddot{q}_{1i}(m_1 + m_2) + \underbrace{g(m_1 + m_2)}_{b} - u_{1i})^2$$

which helps to recognize that $b = g(m_1 + m_2)$ is a bias term which won't improve the learning results.

1.6 Model Evaluation

Only for u_1 are the predictions accurate during all the time steps. For u_2 the approximation doesn't resemble the measured torques during the measurements. The approximation for the force u_3 has a similar behavior as the measured values. Nevertheless, the values differ substantially during the beginning and also at the end. By $t = 50$ the model has the same value for u_3 as the measured value. In conclusion this method does not have an acceptable accuracy, because only one of the torques/forces was estimated with high accuracy while the others have high differences during the simulation time.
The problem of this method is the usage of fixed basis functions ϕ. This parametric functions are not capable of capturing nonlinear effects like friction[2]. In order to capture this nonlinear effects combining our parametric model with nonparametric model, e.g. Gaussian Process, could improve the performance of the model[3].

[2]Lecture 7 Model Learning

[3]D. Nguyen-Tuong and J. Peters, "Using model knowledge for learning inverse dynamics," 2010 IEEE International Conference on Robotics and Automation, Anchorage, AK, USA, 2010, pp. 2677-2682, doi: 10.1109/ROBOT.2010.5509858.

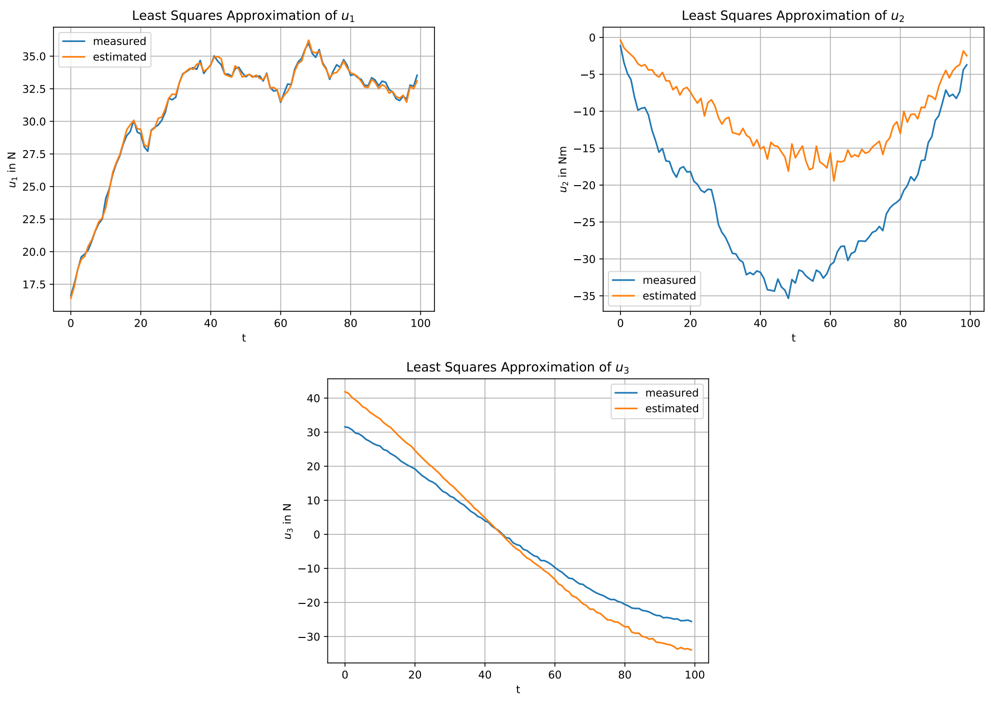


Figure 1: Comparison of the approximated and measured forces/torques

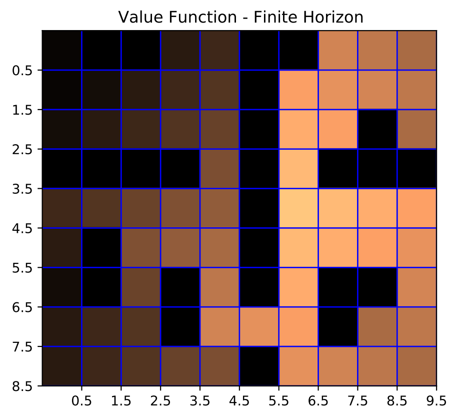
1.7 Models for Robot Learning

- *Forward models* predict the new state given the current state and action. They represent a causal relationship between states and actions $(s_k, a_k) \Rightarrow s_{k+1}$. Examples for applications are prediction, filtering, learning simulations.
- *Inverse Models* are different than the forward models. They represent an anti-causal relationship. They infer the current action, given the current state and the desired new state $(s_k, s_{k+1}) \Rightarrow a_k$. Examples for application are inverse dynamics control and feedback linearization control.
- *Mixed Models* are models which use a combination of forward and inverse models. This model can be used for example in inverse kinematics and operational space control.[4]

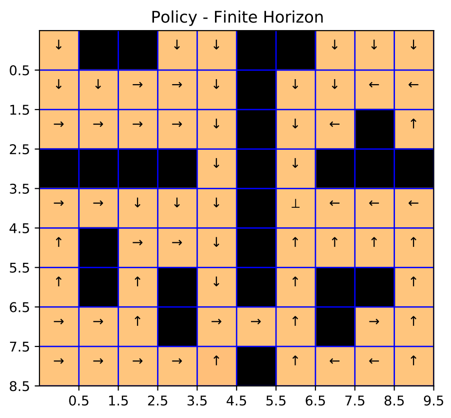
[4] Nguyen-Tuong, D., Peters, J. Model learning for robot control: a survey. Cogn Process 12, 319–340 (2011). https://doi.org/10.1007/s10339-011-0404-1

2 Task 2 - Reinforcement Learning

2.1 a - Finite Horizon Problem



(a) Plot for the value function with horizon $T = 15$



(b) Plot for the policy with horizon $T = 15$

Figure 2: Value function and policy for $T = 15$

The policy plotted in Figure 2b shows how the robot will act when it still has 15 time steps ahead. The plotted policy shows a clear pattern. The robot tries most of the time to reach the toy the fastest way as possible and stay there but always evades going through the black squares. Depending on its current position it will evade going past by the cat, even though it isn't the fastest way. For example if the robot starts on the position (8,4), the robot take the longer way to the toy. The fastest way would be going through the cat and the robot needs only 12 action for it, but the robot takes not the direct way. The robot needs 14 action now.

In those cases the cost of stumbling upon the cat are too high. The robot collects dirt along the way but it seems that it never stays where the dirt is and tries to reach the toy.

If the robot begins from the state (9, 4) and $T = 15$, it needs 7 actions to reach the toy, leaving 8 time steps to stay playing with the toy. The total reward of staying 8 time steps at the toy are 8000 and having to go through the cat costs him -3000 reward. This means that the robot has a gain of 5000. If the time horizon equals 10, the reward obtained by playing with the toy is too low in order for the robot to go through the cat. In this case it will try to go somewhere else.

```python
# Finite Horizon
V, Q = ValIter(grid_world, maxSteps=15, infHor=False)
V = V[:, :, 0]
showWorld(np.maximum(V, 0), 'Value Function - Finite Horizon')
if saveFigures:
    plt.savefig('value_Fin_15.pdf')

policy = findPolicy(Q)
ax = showWorld(grid_world, 'Policy - Finite Horizon')
showPolicy(policy, ax)
if saveFigures:
    plt.savefig('policy_Fin_15.pdf')
```

Listing 2: Finite Horizon

```python
def ValIter(R, discount=None, maxSteps=None, infHor=False, probModel=False):
    T = maxSteps
    rows, columns = R.shape

    robot_actions = {
        "down": 0,
        "right": 1,
        "up": 2,
        "left": 3,
        "stay": 4
    }

    Q = np.zeros((rows, columns, len(robot_actions)))  # Basis Function
    if not infHor:
        V = np.zeros((rows, columns, T))                #Value Function
        V[:, :, -1] = np.copy(R)

        for t in reversed(range(T - 1)):
            for row in range(rows):
                for col in range(columns):
                    for action in robot_actions.items():
                        if not probModel:
                            new_pos = calcNextPosition(action[0], np.asarray([row, col]))
                            Q[row, col, action[1]] = R[row, col] + V[new_pos[0], new_pos[1], t + 1]
                        else:
                            pass
                            #some other code
                    V[row, col, t] = max(Q[row, col, :])
    else:
        pass
        #some other code

    return V, Q
```

Listing 3: ValIter for Finite Horizon

2.2 b - Infinite Horizon Problem - Part 1

The discount factor has to be between 0 and 1 to scale down the rewards. The total sum remains bounded. It is a trade-off between long term vs. immediate reward. In the infinite horizon case $T = \infty$, so that the sum can reach infinity. In order to constraint the sum, the discount rate γ is used. The discount rate determines the present value of future rewards. Thereby future rewards are discounted by γ per time step. Its a mathematical trick to improve convergence, because the sum of rewards keeps growing. It is called unboundedness in mathematical terms.

Rewriting the first equation:

$$J_\pi = \mathbb{E}_\pi\left[\sum_{t=1}^{T-1} r_t(s_t, a_t) + r_T(s_T)\right] \qquad \Longrightarrow \qquad J_\pi = \mathbb{E}_{\mu_0,p,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$$

Because its deterministic and the action is independent. The formula has to be:

$$J_\pi = \sum_{t=0}^{\infty} \gamma^t r(s_t)$$

2.3 c - Infinite Horizon Problem - Part 2

The new policy looks almost exactly the same as the old policy with finite horizon. The difference lies on the fact, that the robot will not visit the cat on its way to the toy. The reason behind this policy is that now the robot has infinite time to reach and stay with the toy. It will not care taking a longer route because of the new horizon.

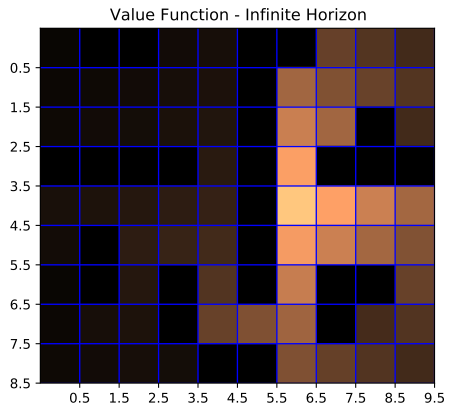
```python
# Infinite Horizon
V, Q = ValIter(grid_world, discount=0.8, infHor=True)
showWorld(np.maximum(V, 0), 'Value Function - Infinite Horizon')
if saveFigures:
   plt.savefig('value_Inf_08.pdf')

policy = findPolicy(Q)
ax = showWorld(grid_world, 'Policy - Infinite Horizon')
showPolicy(policy, ax)
if saveFigures:
    plt.savefig('policy_Inf_08.pdf')
```

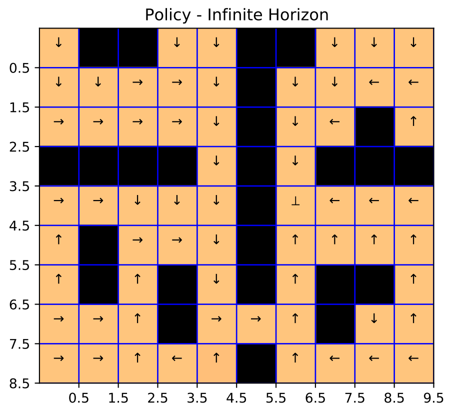
Listing 4: Infinte Horizon

```python
def ValIter(R, discount=None, maxSteps=None, infHor=False, probModel=False):
    T = maxSteps
    rows, columns = R.shape

    robot_actions = {
        "down": 0,
        "right": 1,
        "up": 2,
```



(a) Plot for the value function with infinite horizon



(b) Plot for the policy with infinite horizon

Figure 3: Value function and policy for $T = \infty$

```
        "left": 3,
        "stay": 4
    }

    Q = np.zeros((rows, columns, len(robot_actions)))  # Basis Function

    if not infHor:
        pass
        #some other code
    else:
        improvement = np.inf
        V = np.zeros(R.shape)
        V_last = np.zeros(R.shape)

        while improvement > 1e-5:
            for row in range(rows):
                for col in range(columns):
                    for action in robot_actions.items():
                        new_pos = calcNextPosition(action[0], np.asarray([row, col]))
                        Q[row, col, action[1]] = R[row, col] + discount * V[new_pos[0], new_pos[1]]
            V = np.max(Q, axis=2)
            improvement = abs(np.sum(V.flatten() - V_last.flatten()))
            V_last = V

    return V, Q
```

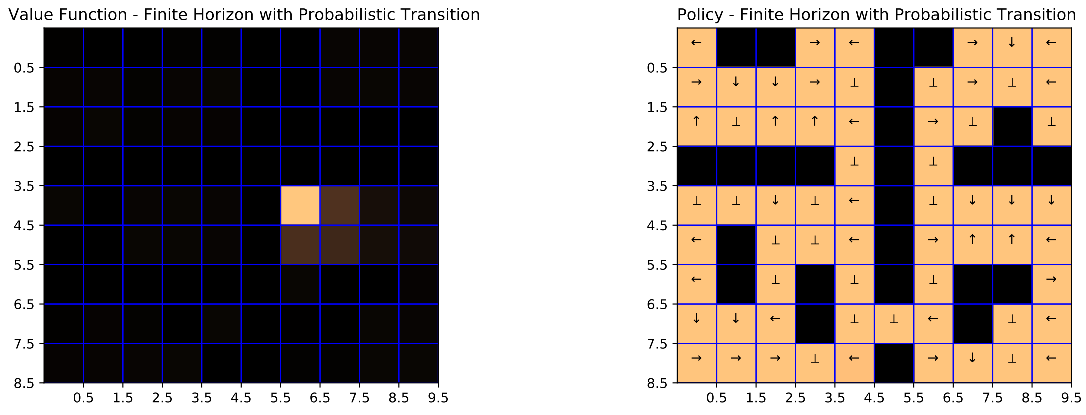
Listing 5: ValIter for Infinte Horizon

2.4 d - Finite Horizon Problem with probabilistic Transition Function

The most common actions are staying in the current state and moving away from the black squares. For example, when the robot is in state (9, 4) and $T = 15$, it will decide to stay, because the probability of taking a wrong step and landing in a black square is too high $(0.1 + 0.1)$, so it decides to take advantage of the reward it already acquires by cleaning the dirt. Another example is the state (6, 7), in the water square below the toy. Even thought the toy is one action upwards away, the robot decides to move away from the black square, because moving upwards means for it, that it has a 0.1 probability of landing on the left black square. This would result in high costs, so it decides just moving away from the water.

```
# Finite Horizon with Probabilistic Transition
V, Q = ValIter(grid_world, maxSteps=15, probModel=True)
V = V[:, :, 0]
showWorld(np.maximum(V, 0), 'Value Function - Finite Horizon with Probabilistic Transition')
if saveFigures:
    plt.savefig('value_Fin_15_prob.pdf')

policy = findPolicy(Q)
ax = showWorld(grid_world, 'Policy - Finite Horizon with Probabilistic Transition')
showPolicy(policy, ax)
if saveFigures:
```



(a) Plot for the value function with horizon $T = 15$ (b) Plot for the policy with horizon $T = 15$

Figure 4: Value functions and policy with probabilistic transition model and $T = 15$

```
    plt.savefig('policy_Fin_15_prob.pdf')
```

Listing 6: Finite Horizon with Probabilistic Transition

```
def ValIter(R, discount=None, maxSteps=None, infHor=False, probModel=False):
    T = maxSteps
    rows, columns = R.shape

    robot_actions = {
        "down": 0,
        "right": 1,
        "up": 2,
        "left": 3,
        "stay": 4
    }

    Q = np.zeros((rows, columns, len(robot_actions)))  # Basis Function

    if not infHor:
        V = np.zeros((rows, columns, T))                #Value Function
        V[:, :, -1] = np.copy(R)

        for t in reversed(range(T - 1)):
            for row in range(rows):
                for col in range(columns):
                    for action in robot_actions.items():
                        if not probModel:
                            pass
                            #some other code
                        else:
                            new_pos, prob = calcNextPosition(action[0], np.asarray([row, col]), True)
                            Q[row, col, action[1]] = R[row, col]
                            for pos, p in zip(new_pos, prob):
                                Q[row, col, action[1]] = Q[row, col, action[1]] + p * V[pos[0], pos[1], t + 1]
                    V[row, col, t] = max(Q[row, col, :])
    else:
        pass
        #some other code

    return V, Q
```

Listing 7: ValIter for Finite Horizon with Probabilistic Transition

```
def calcNextPosition(action, pos, prob_model=False):
    new_pos = np.zeros((1, 2))
    prob = 0
    if not prob_model:
        if action == "down":
            new_pos = pos + np.array([1, 0])
            if new_pos[0] > 8:
                new_pos = pos

        elif action == "right":
            new_pos = pos + np.array([0, 1])
            if new_pos[1] > 9:
                new_pos = pos

        elif action == "up":
            new_pos = pos + np.array([-1, 0])
            if new_pos[0] < 0:
                new_pos = pos

        elif action == "left":
            new_pos = pos + np.array([0, -1])

            if new_pos[1] < 0:
                new_pos = pos

        elif action == "stay":
            new_pos = pos
    else:
        if action == "down" or action == "up":
            #also possible action left OR right OR failing
            new_pos = calcNextPosition(action, pos)
            new_pos = np.vstack((new_pos, calcNextPosition("left", pos)))
            new_pos = np.vstack((new_pos, calcNextPosition("right", pos)))
            new_pos = np.vstack((new_pos, calcNextPosition("stay", pos)))
            prob = np.array([0.7, 0.1, 0.1, 0.1]).reshape((4, 1))
        elif action == "right" or action == "left":
            #possible actions up OR down OR failing
            new_pos = calcNextPosition(action, pos)
            new_pos = np.vstack((new_pos, calcNextPosition("down", pos)))
            new_pos = np.vstack((new_pos, calcNextPosition("up", pos)))
            new_pos = np.vstack((new_pos, calcNextPosition("stay", pos)))
            prob = np.array([0.7, 0.1, 0.1, 0.1]).reshape((4, 1))
        else:
            new_pos = pos.reshape((1, 2))
            prob = np.array([1])

    if prob_model == False:
        return np.asarray(new_pos)
    else:
        return np.asarray(new_pos), np.asarray(prob)
```
Listing 8: calculate new position / new state

2.5 e - Reinforcement Learning - Other Approaches

The two assumptions that allow the usage of the value iteration algorithm are that the analytical transition model is known, which is not always the case for enough complex dynamical systems, and that the dynamics of the system fulfill the *Markov* property, which, roughly speaking, assumes that the new state is only dependant of the current state and action. The Markov property is most of the times a simplification of the real dynamics.

When we neither have a transition model or this model doesn't fulfill our Markov requirement, we can use approximation methods. One famous method is the Temporal Differences Learning method, or TD-Learning. It assumes that the value function can be represented using a linear model:

$$V^{\pi}(\mathbf{s}) \approx V_{\omega}(\mathbf{s}) = \phi^{T}(\mathbf{s})\omega$$

where ω represents the parameters to be learned. In order to learn the parameters, one can minimize the (bootstrapped) mean-squared error (MSE) in equation 2 using iterative gradient based optimization methods.

$$MSE_{BS,t}(\omega) = \frac{1}{N}\sum_{i=1}^{N}(\hat{V}^{\pi}(\mathbf{s}_i) - V_{\omega}(\mathbf{s}_i))^2 \tag{2}$$

3 Task 3 - Episode Policy Search with Policy Gradients

3.1 a) Analytical Derivation

The multivariate Gaussian π for $\boldsymbol{\theta}$ and the parameter $\boldsymbol{\omega}$ ($\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$) is:

$$\pi(\boldsymbol{\theta}|\boldsymbol{\omega}) = \frac{1}{(2\pi)^{\frac{K}{2}}} \cdot \frac{1}{|\boldsymbol{\Sigma}|^{\frac{1}{2}}} \cdot e^{-\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu})} \tag{3}$$

Notice the difference between the policy function symbol π on the left side of the equation and the actual value π (3.14...) on the right.

in formula 3, K is the dimension of $\boldsymbol{\theta}$ (in this case 10), sample mean $\boldsymbol{\mu}$ is a K-dimensional vector, covariance matrix $\boldsymbol{\Sigma}$ is a $K \times K$ diagonal positive definite matrix.

The natural logarithm of the policy is:

$$\ln(\pi(\boldsymbol{\theta}|\boldsymbol{\omega})) = -\frac{K}{2} \cdot \ln(2\pi) - \frac{1}{2} \cdot \det|\boldsymbol{\Sigma}| - \frac{1}{2} \cdot (\boldsymbol{\theta}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu}) \tag{4}$$

The derivatives with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are in equation 5 and 6. The first one only holds if $\boldsymbol{\Sigma}$ is symmetric, which it is in our case. The second one is further simplified in the second row, where the symmetric property of $\boldsymbol{\Sigma}$ is utilized. See also equation 57, 61 and 86 in the Matrix Cookbook[5].

$$\frac{\partial \ln(\pi(\boldsymbol{\theta}|\boldsymbol{\omega}))}{\partial \boldsymbol{\mu}} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu}) \tag{5}$$

$$\begin{aligned}\frac{\partial \ln(\pi(\boldsymbol{\theta}|\boldsymbol{\omega}))}{\partial \boldsymbol{\Sigma}} &= -\frac{1}{2} \cdot (\boldsymbol{\Sigma}^T)^{-1} + \frac{1}{2} \cdot (\boldsymbol{\Sigma}^T)^{-1} \cdot (\boldsymbol{\theta}-\boldsymbol{\mu}) \cdot (\boldsymbol{\theta}-\boldsymbol{\mu})^T \cdot (\boldsymbol{\Sigma}^T)^{-1} && (6)\\ &= -\frac{1}{2} \cdot \boldsymbol{\Sigma}^{-1} + \frac{1}{2} \cdot \boldsymbol{\Sigma}^{-1} \cdot (\boldsymbol{\theta}-\boldsymbol{\mu}) \cdot (\boldsymbol{\theta}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1} && (7)\end{aligned}$$

For a diagonal covariance matrix, the inverse can be calculated very easily as shown in equation 8 for our case where the dimension of the covariance matrix is (10 x 10).

$$\boldsymbol{\Sigma}^{-1} = \operatorname{diag}\left(\frac{1}{\sigma_1^2}, \frac{1}{\sigma_2^2}, \dots, \frac{1}{\sigma_{10}^2}\right) \tag{8}$$

3.2 b) Programming Exercise

Figure 5 shows the average learning progress over 10 runs with 100 iterations with 25 episodes per iteration and its 95% confidence interval. At around iteration 25 the learning drastically worsens, resulting in an average return of approx. $-0.15 \cdot 10^{-69}$. Furthermore the confidence interval indicates a high variance from that iteration on-wards between the runs. Thus, not utilizing a baseline for the gradient is not a stable approach for this task.

3.3 c) A Little Trick

To improve the results from the previous exercise, we now subtract a baseline from the gradient, as shown on slide 24 in lecture 10. This is done to reuduce the variance in the estimate. Furthermore it is unbiased. In this task, the average reward per iteration is used as a baseline. Figure 6 shows the learning progress with the same parameters as in the previous task but this time a baseline is subtracted. This results in a much better learning progress. The average return increases with every iteration, converging towards 0. Also the variance is relatively small, especially in comparison to the previous task. This proves to be a much better approach to learning then the previous task.

[5] `http://www2.imm.dtu.dk/pubdb/edoc/imm3274.pdf`

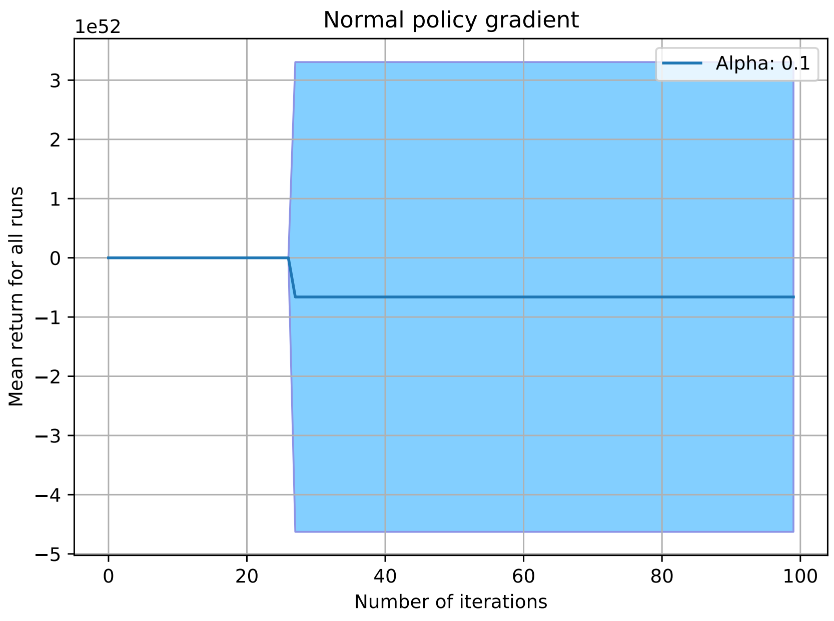


Figure 5: Mean return with Alpha = 0.1

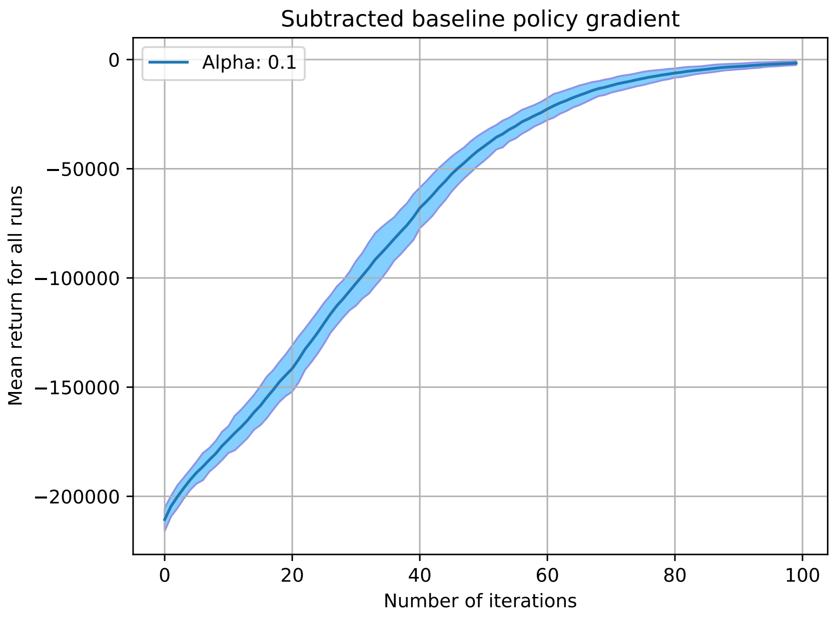


Figure 6: Mean return with Alpha = 0.1 and subtracting the baseline gradient

3.4 d) Learning Rate

Figure 7 shows the average return for different learning rates, the baseline is still utilized. In this example, an increasing learning rate achieves better results within the same number of iterations and all learning rates converge towards the maximum return of 0. Utilizing a learning rate of 0.4, the learning process is four to five times as fast in comparison to a learning rate of 0.1. However, increasing the learning rate beyond 0.4 is no guarantee that the results will further improve.

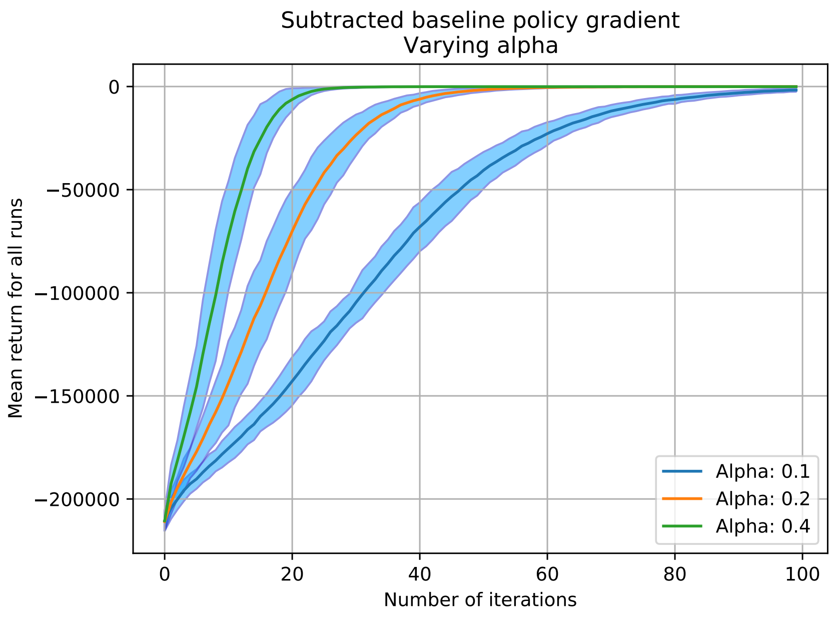


Figure 7: Varying Alpha while subtracting the baseline gradient

3.5 e) Variable Variance

Figure 8 show the learning progress when using $\alpha = 0.4$ but also updating Σ. There are almost no differences in comparison to not learning the variance. The confidence interval is very slightly smaller but not on a significant scale. The lower bound for the variance was implemented and set to 1 but not needed since it did collapse to small values during the training. Also learning the variance is slightly harder than only learning the mean, since the gradient for the covariance matrix also has to be evaluated. However the inverse of the covariance matrix also has to be evaluated for the gradient of the mean, so the extra computational effort is not too high.

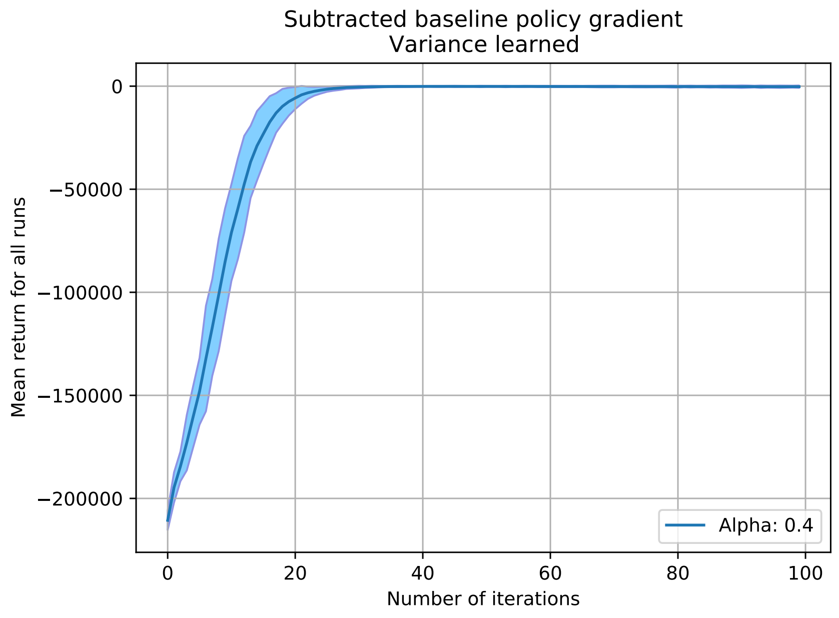


Figure 8: Learning the variance

```
from matplotlib import pyplot as plt
import numpy as np
from Pend2dBallThrowDMP import *
import time

# %matplotlib inline
np.set_printoptions(precision=3, linewidth=100000)

env = Pend2dBallThrowDMP()

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

# YOUR CODE HERE
def log_gauss_derivative(x, mu, sigma):
    sigma_inv = np.diag(1 / np.diag(sigma))
    diff = x - mu
    der_mu = sigma_inv.dot(diff)
    der_sigma = 0.5 * (-sigma_inv + sigma_inv.dot(np.outer(diff, diff)).dot(sigma_inv))

    return der_mu, der_sigma


def exploration(alpha, use_baseline=True, change_sigma=False, sigma_lb=None):
    if change_sigma:
        assert sigma_lb is not None

    assert isinstance(alpha, tuple)

    fig0, ax0 = plt.subplots()


    for alpha_i in alpha:
        reward = np.zeros((numTrials, maxIter, numSamples))
        for trial_i in range(numTrials):
```

```python
            mu = np.zeros(numDim)
            sigma = np.diag((np.ones(10))) * 10 ** 2
            for i in range(maxIter):
                gradient_mu = np.zeros(mu.shape)
                gradient_sigma = np.zeros(sigma.shape)
                theta = [np.zeros(10)] * numSamples
                for j in range(numSamples):
                    theta[j] = np.random.multivariate_normal(mu, sigma)
                    reward[trial_i, i, j] = env.getReward(theta[j])

                assert id(theta[0]) != id(theta[1])
                if use_baseline:
                    b = np.mean(reward[trial_i, i, :])
                else:
                    b = 0

                for j in range(numSamples):
                    derivative_gaussian = log_gauss_derivative(theta[j], mu, sigma)
                    gradient_mu += derivative_gaussian[0] * (reward[trial_i, i, j] - b)
                    if change_sigma:
                        gradient_sigma += derivative_gaussian[1] * (reward[trial_i, i, j] - b)

                mu = mu + alpha_i * gradient_mu / numSamples

                # Learn covariance
                if change_sigma:
                    gradient_sigma = np.diag(np.diag(sigma))
                    sigma = sigma + alpha_i * gradient_sigma / numSamples  # Normalize by number of samples

                    # Set all diagonal elements below lower bound to the lower bound
                    sigma[range(numDim), range(numDim)] = np.maximum(sigma[range(numDim), range(numDim)], sigma_lb)

        reward_mean = np.mean(reward, axis=(0, 2))
        reward_std = np.empty(reward_mean.shape)
        for i in range(maxIter):
            reward_std[i] = np.sqrt(np.sum((reward[:, i, :].ravel() - reward_mean[i]) ** 2) / (numTrials * numSamples))

        ax0.plot(np.arange(maxIter), reward_mean, label=f"Alpha: {alpha_i}")
        plt.fill_between(np.arange(maxIter), reward_mean - 2 * reward_std, reward_mean + 2 * reward_std, alpha=0.5,
     edgecolor='#1B2ACC', facecolor='#089FFF')

    ax0.set_xlabel("Number of iterations")
    ax0.set_ylabel("Mean return for all runs")
    ax0.legend()
    ax0.grid()
    plt.tight_layout()

    return fig0

fig = exploration(alpha=(0.1,), use_baseline=False, change_sigma=False)
plt.gca().set_title("Normal policy gradient")
plt.tight_layout()
fig.savefig("Normal policy gradient.pdf")

fig = exploration(alpha=(0.1,), use_baseline=True, change_sigma=False)
plt.gca().set_title("Subtracted baseline policy gradient")
plt.tight_layout()
fig.savefig("Subtracted baseline policy gradient.pdf")

fig = exploration(alpha=(0.1, 0.2, 0.4), use_baseline=True, change_sigma=False)
plt.gca().set_title("Subtracted baseline policy gradient\nVarying alpha")
plt.tight_layout()
fig.savefig("Subtracted baseline policy gradient_Varying alpha.pdf")

fig = exploration(alpha=(0.4,), use_baseline=True, change_sigma=True, sigma_lb=1)
plt.gca().set_title("Subtracted baseline policy gradient\nVariance learned")
plt.tight_layout()
fig.savefig("Subtracted baseline policy gradient_Variance learned.pdf")
```

Listing 9: Code for 5.3

3.6 f) Natural Gradient

For calculating the natural gradient in case of episode-based exploration and Gaussian distributions the formulas from slide 40 of lecture 10 are utilized. In our case the mean $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ are the changing parameters and thus $\frac{\partial \boldsymbol{\mu}}{\partial \alpha_i}$ and $\frac{\partial \boldsymbol{\Sigma}}{\partial \beta_i}$ are the derivatives in respect to the mean and the covariance itself. The natural gradient is then computed by calculating the FIM and the normal gradient using formula 9.

$$\nabla_{\boldsymbol{\omega}}^{\text{NEG}} J = \boldsymbol{G}(\boldsymbol{\omega})^{-1} \cdot \nabla_{\boldsymbol{\omega}} J \tag{9}$$

The natural gradient is derived from the Fisher information matrix and thus from the Kullback Leibler divergence. The KL-divergence measures the similarity/distance between two distributions. This can be done with either information or moment projection. An approximation of the KL-divergence with a 2nd order Taylor approximation results in the FIM. The FIM shows how the single parameters influence the distribution. The inverse of the FIM can then be multiplied with the normal gradient, resulting in the natural gradient. It is invariant to parameter transformations, thus ignoring the scaling of the parameters. Instead of corresponding to the steepest descent in parameter space they correspond to the steepest descent in policy space. Its advantage are that it does not reduce exploration as quickly as the normal gradient

Using the natural gradient is more difficult than using the normal gradient. As shown in formula 9 the natural gradient needs the normal gradient as well as the inverse of the FIM. For episode-based exploration with Gaussian distributions some simplifications as mentioned before can be used. For step-based exploration the compatible function approximation can be utilized with which follows:

$$\nabla_{\boldsymbol{\Theta}}^{\text{NEG}} J = \boldsymbol{\omega} \tag{10}$$

which makes the natural gradient easy in this case. For all other cases it is not straightforward to use.

4 Reinforcement Learning

4.1 RL Exploration Strategies

The two spaces in which exploration in RL is possible are the action space and the parameter space.

- Action space (Step-based)
  - This approach learns the full movement of the model. Therefore, a full model is learned it results to a less variance in quality assessment. It computes for every given action the rewards and at each time step it will explore the action space with a low-level stochatic policy.
- Parameter space (Episode-Based)
  - Its a low-level policy. The exploration in parameter space allows more sophisticated strategies and is often very efficient for a small amount of parameters. The disadvantage of the parameter space approach is the high variance in return used in the dataset for the policy update. It makes the learned model unstable. To reduce the variance in return it is often modeld as a deterministic policy. The structure of the model remains unknown. This is called Black-Box optimizer