Policy Learning for an Unbalanced Disc

5SC28 Machine Learning for Systems and Control - Group 2

C.W.D. Bielders
0937368
c.w.d.bielders@student.tue.nl

C. Bouwens
1392794
c.bouwens@student.tue.nl

D.E.W. Soons
0936065
d.e.w.soons@student.tue.nl

L. Zuo
1481207
l.zuo@student.tue.nl

***Abstract*—This paper presents machine learning approaches for the modelling and control of a nonlinear unbalanced disc system, similar to that of an inverted pendulum. Using both a Artificial Neural Networks (ANN) and Gaussian Processes (GP) methods, models of the system were developed. The ANN model showed better performance over the GP model. However, if only a relatively small training data set is available, the GP model performs better. Furthermore, an implementation of GP based policy search is presented that is able to swing up and stabilize the pendulum in the upright position. Lastly, an approach for learning a multi-task policy is presented that is able to swing-up and stabilize the pendulum at multiple target angles.**

***Index Terms*—inverted pendulum, policy learning, Gaussian Process, Artificial Neural Network**

I. Introduction

The unbalanced disc system, depicted in figure 1, is a good platform for learning and getting a better understanding of system identification principles and control methods. Swinging the disc from its stable downward position up, and stabilising it in its upwards position is a challenging control problem. A torque can be exerted on the disc via an electric motor. The input voltage of this motor is bound between $\pm 3V$. The dynamics of the unbalanced disc system can be mathematically described by

$$\begin{aligned} \dot{\theta}(t) &= \omega(t) \\ \dot{\omega}(t) &= \frac{Mgl}{J}\sin(\theta(t)) - \frac{1}{\tau}\omega(t) + \frac{K_m}{\tau}u(t) \end{aligned} \tag{1}$$

where $\theta(t)$ is the angle of the disc and $u(t)$ is the input voltage. This equation shows that the system is inherently nonlinear. In this paper, π radians is the stable downward position of the disc.

Fig. 1. Unbalanced disk setup

This paper presents a machine learning approach to obtain models that accurately capture the dynamics of the unbalanced disc system using only observed input/output data, without previous knowledge of the systems dynamics. An Artificial Neural Network (ANN) Model and a Gaussian Process (GP) Model will be developed. The models will be compared to see which model more accurately captures the systems dynamics. Furthermore, a machine learning approach will be used to learn a control policy for the system that can swing up the disc and stabilise it at the top position. The policy is then expanded to be able to position the disc at two more angles other than the top position, i.e. a multi-task policy. These control policies are also only based on input/output data.

The paper is structured as follows: section II discusses the identification of the system dynamics starting from data and the development of the different models, section III discusses the policy learning to swing up and stabilise the disc, section IV discusses the policy learning to use multiple targets and move from target to target and section V provides a conclusion and future recommendations.

II. Modeling the System Dynamics

A. Data Generation for Modeling

Input and output data of the system is required for ANN and GP modeling. This data needs to be generated. The models should accurately model the dynamics of the unbalanced disc for a rotation of $\pm 120°$. Therefore, it is important that the input is designed such that the disc reaches these angles. If the disc overshoots these angles and makes a full rotation, the data becomes invalid for the ANN and GP modeling. Further more, the system input can not exceed $\pm 3V$, as it is saturated at this voltage.

The input signal used for the data generation is a Random Binary Sequence (RBS) with clock period of 2 seconds and amplitude of $\pm 3V$. This input is given to the system for $300s$. Figure 2 shows the output angle given this input signal. The plot shows that the angle stays mostly between the $\pm 120°$ bound. This gives a training data set which is used in the ANN and GP modeling. A second data set is generated via the same method, which is used for testing the models.

B. Artificial Neural Network (ANN) Model

The systems output is not only dependant on the input, but also on the systems states. Therefore, a feedforward network will not be sufficient for the modeling of the systems dynamics. A NARX network is used in the modeling process,

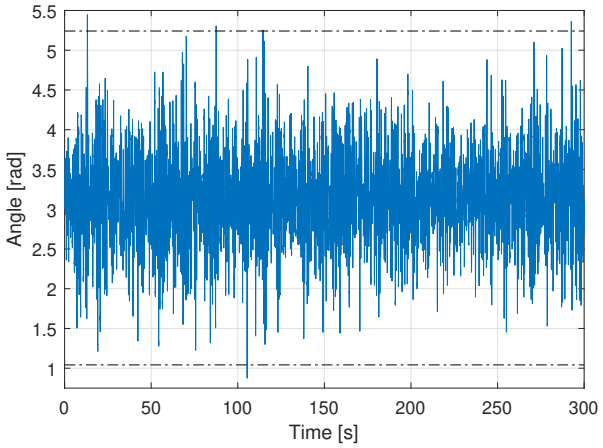


Fig. 2. Plot of the training dataset for the GP and ANN model. The dash-dotted lines reprisent the $\pm 120°$ bound.

since this allows the output to be fed back as input to the network.

To train the network, the training dataset is used. The input and output data is first normalized to fit between $[-1, 1]$. To test the performance of the network, the normalized test dataset is used. The performance of the network can be determined with the squared L_2 norm:

$$V_N(\theta) = \frac{1}{N} \sum_{k=0}^{N-1} (y_k - \hat{y}_k)^2 \tag{2}$$

The lower the value of $V_N(\theta)$, the better the performance of the network. Note that $V_N(\theta)$ can never be 0, since the output of the system contains noise. The noise is white noise with variance of $\sigma^2 = 0.001$. Therefore, $V_N(\theta)$ is bounded at 0.001. The neural network is trained in Matlab using the Deep Learning Toolbox.

The final NARX network can be seen in figure 3. The network contains 2 hidden layers of each 5 nodes. Both hidden layers feature a `tansig` activation function, which is defined by

$$f(n) = \frac{2}{1 + e^{-2n}} - 1 \tag{3}$$

The output layer has a linear activation function.

The input and feedback signal are delayed by 1 to 20 delays. this gives that the input of the network is $[u(k) \ \ldots \ u(k-20) \ y(k-1) \ \ldots \ y(k-20)]$. This network structure results in a $V_N(\theta) = 0.0013$ for the training dataset. For the test dataset, $V_N(\theta) = 0.0016$. Increasing the number of layers, nodes per layer or input delays does not improve the values of $V_N(\theta)$.

To further analyse the performance of the ANN, a chirp input is given to the unbalanced disc setup and the ANN model. The chirp starts at $1Hz$ and reaches $2Hz$ in $10s$, with an amplitude of $2V$. Figure 4 shows the output of the system and the ANN model. The plot shows a relation between the error and the angular velocity. The error is the biggest when the disc reaches the highest angular velocity, at πrad in a

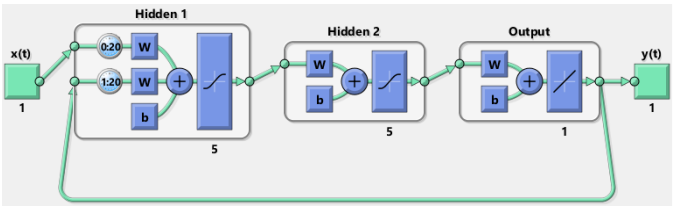


Fig. 3. NARX Network for modeling of disc dynamics

swing from the lower limit to the upper limit. This shows that the ANN network is less accurate for states with high angular velocity. This can be explained by the training dataset. This dataset contains little information of states with high angular velocities. Therefore, the ANN has not been trained as well for these states compared to low angular velocity states. The performance of the ANN model can potentially be increased if the training dataset contains more information of high angular velocity states.

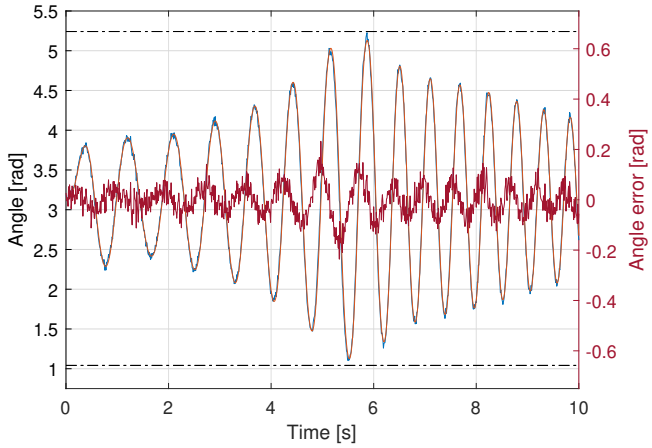


Fig. 4. Output angle of the true system (blue) and the ANN model (orange) given a chirp input signal. The dash-dotted line represents the $\pm 120°$ limit. The error is represented by the red line. Note that the error a different y-axis.

C. Gaussian Process (GP) Model

1) Training Data Generation: To train the GP model, two data sets were generated for training. The first data set is the one obtained with a RBS signal as described in section II.A. Of this data set only the first 30 seconds were used due to memory overloading issues occurring whenever the training set became too large. The second is a combination of a chirp signal and a Random Binary Sequence (RBS) as shown in. Firstly, a chirp signal with an amplitude of 2.4V was generated which performs a frequency sweep from 0.1 to 2 Hz in 20 seconds, thereafter an input of $0V$ was given for 5 seconds and lastly an RBS signal was generated for 25 seconds with clock period of 0.25 seconds and amplitude of $\pm 3V$. This input signal was then applied to the simulated setup in Simulink while logging the input-output data. Figure 5 shows the input and output signals obtained using the chirp+RBS signal. As

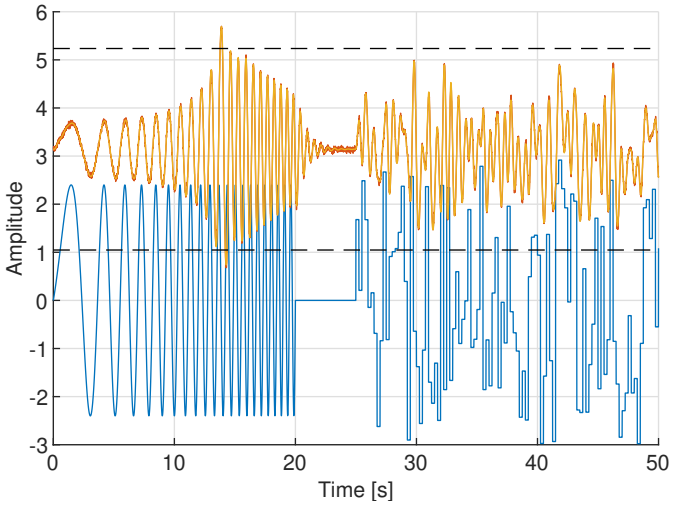


Fig. 5. Chirp+RBS training data for training the GP model showing the input signal in Volts (blue), the output signal in radians (orange) and the noiseless output obtained using the ODE45 solver in radians (yellow).

shown, the input signal stays within bounds as well as the output signal which mostly remains in the $\pm 120°$ region.

The output of the plant is dependent on previous inputs as well as previous outputs, which means the model includes states. Therefore, the gathered training data needs to be converted to an input for a second order system to prepare it for training the GP model. To create this multi-dimensional input data, an input data set $\mathbf{x}$ was created as

$$\mathbf{x} = [\mathbf{y}(k-1), \mathbf{y}(k-2), \mathbf{u}(k), \mathbf{u}(k-1)], \tag{4}$$

where $\mathbf{y}$ is the time-series data containing the output data, $\mathbf{u}$ the time-series data containing input data and k the k-th time-sample at time $t = k \cdot T_s$ [s], with T_s the sampling time. Due to this time-shift, the data starts at the third time-sample.

2) Training the GP model: The GP model was trained using the the GPML toolbox version 4.2 by Rasmussen and Nickisch, which was also provided through Canvas and used during the exercise sessions. The gp-function learns the model and requires a mean, covariance and likelihood function, as well as an inference method, each with its own hyper-parameters. The hyper-parameters are contained in a struct and control the properties of the model, i.e. the GP mean and covariance function and the likelihood function. The mean function computes the mean and its derivatives w.r.t. the part of the hyper-parameters pertaining to the mean. It computes the expected value $m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})]$ of f for the input $\mathbf{x}$. The covariance function computes the covariance and its derivatives w.r.t. the part of the hyper-parameters pertaining to the covariance function. The covariance is computed by $k(\mathbf{x}, \mathbf{z}) = \mathbb{V}[f(\mathbf{x}), f(\mathbf{z})] = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{z}) - m(\mathbf{z}))]$ of f between the inputs $\mathbf{x}$ and $\mathbf{z}$. The likelihood function specifies the form of the likelihood of the GP model and computes terms needed for prediction and inference. The inference method is a function which computes the (approximate) posterior, the (approximate) negative log marginal likelihood and its partial derivatives w.r.t. the hyper-parameters, given a model specification (i.e., GP mean and covariance functions and a likelihood function) and a data set. [2]

The commonly used functions were used first: the **zero prior mean** function $m(\mathbf{x})$, the **squared exponential covariance** function $\kappa(\mathbf{x}, \mathbf{z})$ (`covSEiso`), the **Gaussian likelihood** function $\mathcal{N}(y_i|f_i, \sigma^2)$ (`likGauss`) and the **exact inference method** with Gaussian likelihood (`infGaussLik`). The mean, covariance and likelihood functions are given in equations 5, 6 and 7 respectively. These functions were also used during the exercise sessions and provided a good basis to start training the GP. The corresponding hyper-parameters were initialised to the values shown in Table I. The zero prior mean function did not need a hyper-parameter. The initial values of these hyper-parameters are based on the variance of the Gaussian noise added to the states of the Simulink model: $\sigma^2 = 0.001$, therefore the standard deviation σ is 0.01. No prior optimal initial value was found for σ_f and ℓ, because the optimization function produced the same values whenever the σ hyper-parameter was minimal.

$$m(\mathbf{x}) = 0 \tag{5}$$

$$\kappa(\mathbf{x}, \mathbf{z}) = \sigma_f^2 \exp - \frac{||\mathbf{x} - \mathbf{z}||^2}{2\ell^2} \tag{6}$$

$$\mathcal{N}(y_i|f_i, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(- \frac{(y_i - f_i)^2}{2|sigma^2}\right) \tag{7}$$

The hyper-parameters for the `covSEiso` function are the isotropic length-scale $ln(\ell)$ and the scalar $ln(\sigma_f)$ which scales the covariance function $\kappa(\mathbf{x}, \mathbf{z})$ according to $\sigma_f^2 \kappa(\mathbf{x}, \mathbf{z})$. The hyper-parameter for the Gaussian likelihood function is $ln(\sigma)$. The hyper-parameters were optimized using the GPML toolbox function `minimize()` function, which minimizes a differentiable multivariate function using conjugate gradients. In this case it optimizes the marginal likelihood of the GP prediction with the hyper-parameters given as variables. The resulting optimal hyper-parameters after optimization are shown in Table I. Using theses optimal parameters the GP models achieved a Mean Square Error (MSE) of 0.0046 for the data set using only RBS as input, and an MSE of 0.042 for the chirp+RBS data set. The MSE was calculated using equation 2. From these results it seems that the chirp+RBS-trained model is perfoming better.

TABLE I
GP HYPER-PARAMETER INITIALIZATION AND OPTIMIZED VALUES

Function	Parameter	Value	Optimized Value
meanfunc	[]	[]	[]
covfunc	$[ln(\ell)\ ln(\sigma_f)]$	$ln([0.01\ 0.01])$	$ln([46.52\ 21.75])$
likfunc	$ln(\sigma)$	$ln(0.01)$	$ln(0.068)$

3) Validation: As described in section II.A (*Data Generation for Modeling*), the validation data consists of 300 second long input-output data generated by applying a uniform RBS signal to the plant. The output validation data is shown

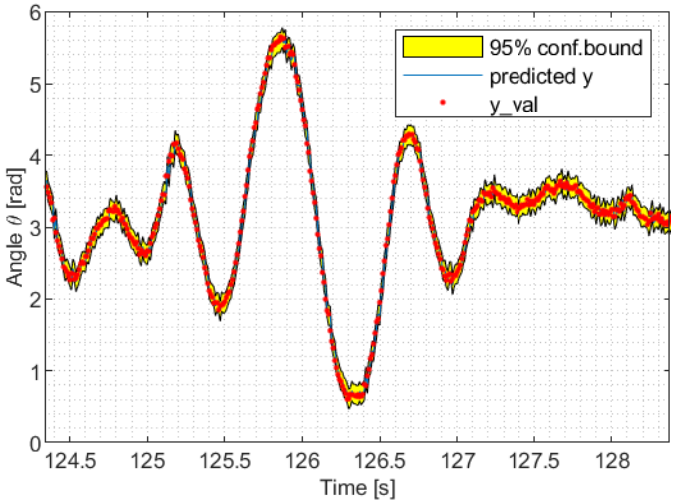


Fig. 6. GP validation result showing a few seconds of the predicted system response (blue) and its 95% confidence bound (yellow) and the actual systems' response (red). Generated with GP model trained with RBS-only data set.

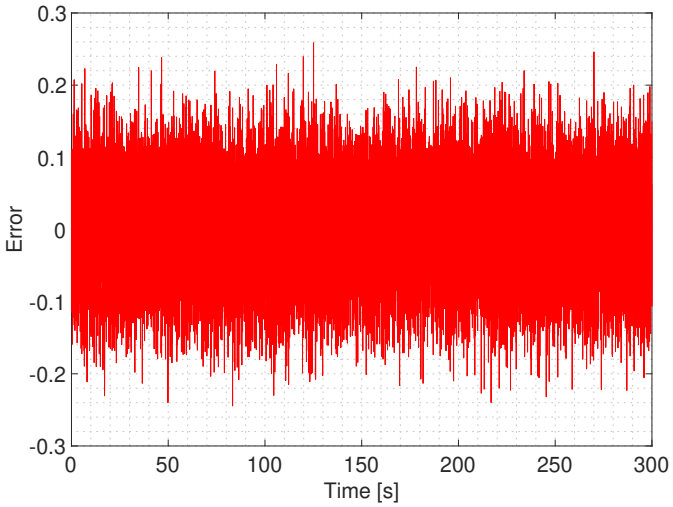


Fig. 7. GP validation error over the full validation time-frame.

in Figure 2. The two GP models were validated using this validation data set and achieved an MSE of 0.0043 and 0.0094 respectively. This shows that the GP model trained using only RBS input performs better than the chirp+RBS trained GP model. This difference in performance might be caused by a deficiency of data-points in the 120° region in the case of the chirp+RBS data set. Furthermore, considering that the noise has a variance of 0.001 the results approaches the optimal model of the pendulum dynamics, because the minimum achievable MSE would be 0.001 as mentioned before in section II.B. The validation prediction of the RBS trained model with its 95% uncertainty bound is shown in Figure 6 in which it is compared to the validation data. The validation error is shown in Figure 7.

D. Comparison between ANN and GP model

The performance of the ANN and GP models can be compared by their MSE of the error, given the test dataset. The MSE is 0.0016 for the ANN model and 0.0043 for the GP model. This shows that the ANN model more accurately models the dynamic of the unbalanced disc system by approximately 269%. The lower accuracy of the GP model can be explained by the shorter training dataset used in training the model. Using a larger dataset for the GP modeling was not possible due to limited memory. This shows a drawback of the GP method, that it takes a lot of memory to train the model with large datasets. However, if for example a controller design needs to be tested, either model will be sufficiently accurate to test the controller. This is of course only true if the model stays within the $\pm 120°$ bound.

In order for a fairer comparison between the ANN and GP method, the ANN model is trained using the same shorter dataset as the GP model. This time the MSE for the ANN model is 0.0058. In this test the GP model performs better. This shows that the GP method is better used for smaller datasets, while the ANN method is better for larger datasets. This statement also holds if the the training time is taken into account. The GP model needs about 7 minutes to train with the sorter dataset. The ANN model takes only about 10 seconds to train with the full 300 seconds dataset.

III. Learning a Swing-Up Policy

A. training the GP model and policy

To create an artificial intelligence (AI) that can swing a pendulum upwards and keep it there, the Matlab PILCO toolbox V0.9 was used. PILCO is a policy search framework for data-efficient reinforcement learning. The toolbox is used to learn a GP model of the system dynamics, perform deterministic approximate inference for policy evaluation, update the policy parameters using exact gradient information and apply the learned controller to the system. The software package provides an interface that allows for setting up novel tasks without the need to be familiar with the intricate details of model learning, policy evaluation and improvement. [3] The PILCO toolbox already included an example environment which featured a pendulum and learned a swing up policy. This pendulum example served as the basis environment for the unbalanced disk policy learning. Adaptations had to be made to match the example environment to the unbalanced disk setup. Firstly, the dynamics of the pendulum were adapted to match the dynamics of the unbalanced disk set up as described in the introduction. The starting angle of the pendulum was also made to be $180°$ (hanging downwards) and the target angle $0°$(the top position). This is $180°$ rotated in comparison to the pendulum example provided by PILCO. However, because the system dynamics were changed to the one used in the introduction. The system is corrected and the pendulum behaves in the expected ways. To speed up the learning process of the GP the amount of line searches of the policy optimization was decreased from 75 to 20, because

the line searches take up most of the time spent per iteration. Furthermore, the GP seemed to make minimal changes to these line searches after 20 line searches.

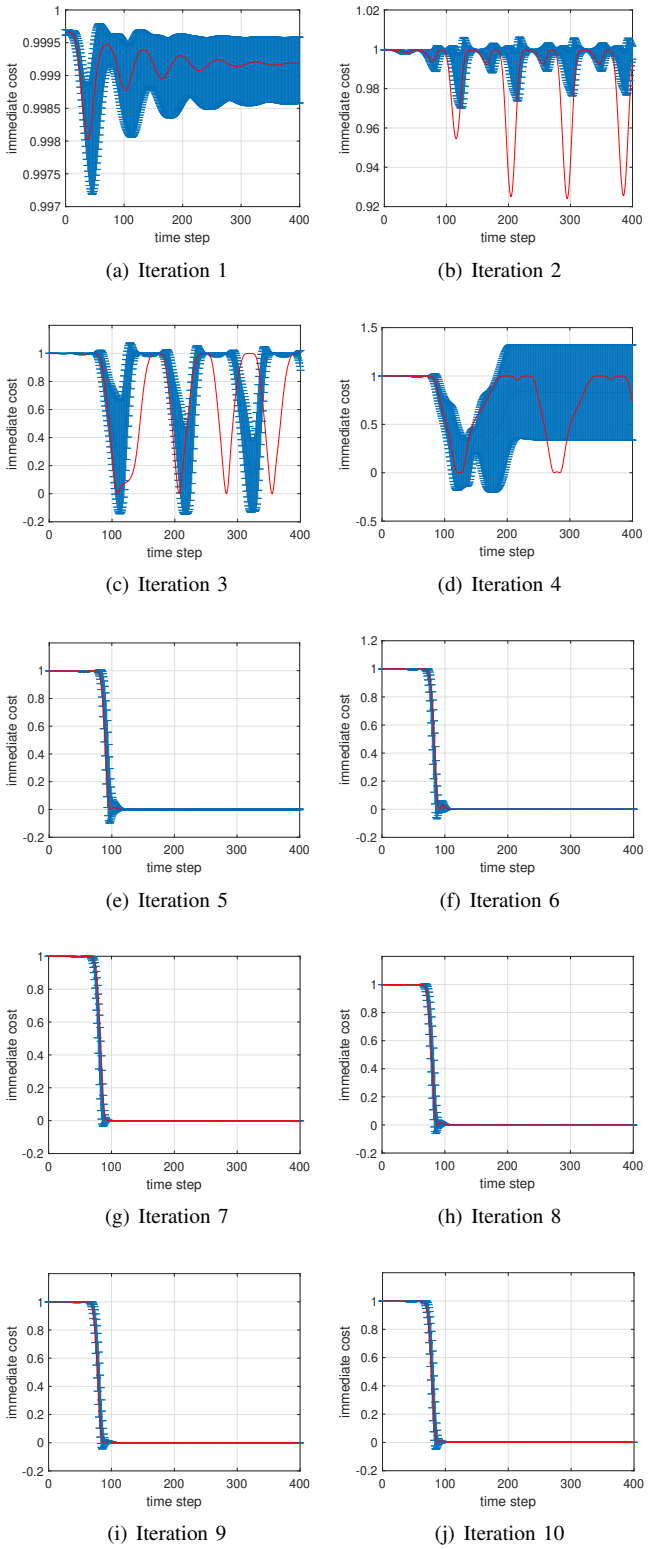
However, these changes seemed to create a problem where the learned policy stabilized the pendulum at the target angle for some time, but destabilizes again shortly after. The learned policy was therefore not optimal, which could have been caused by three suspected problems. The first problem could have been that the policy learning did not get enough iterations to achieve the optimal outcome. The second problem could have been the decreased amount of line searches per iteration. The third problem could have been that the policy learning ended up in a local minimum of the loss function, which prevented the GP to learn the global minimum of the loss function.
The first problem was discarded, due to the GP already showing the less than optimal behaviour multiple iterations before the final iteration was reached. To validate the second problem, the amount of line searches per iteration was put back to 75. However, the same behaviour occurred which invalidated the problem. Therefore, the amount of line searches per iteration was set back to 20. To validate the third problem, the target of the neural network was changed so the neural network would try to aim for a non zero angular velocity. This would remove the local minimum of the loss function by changing the first few tries the policy learning makes. The non-zero value chosen for the target speed was $\omega = 1$. This value was chosen because it is relatively low but high enough to help the policy learning move towards the right direction. This seemed to solve the problem: the pendulum was swung up to the top position and stabilized there from the 6th iteration and onward. Presumably, the target speed gives an almost negligible reward compared to the target angle, therefore prohibiting the policy to aim for a higher speed when the pendulum is at the target position. Consequently, the target speed promotes the swinging action whenever the pendulum is hanging still in the bottom position, which gives the least reward.

With these changes the GP was able to learn a policy that stabilizes the pendulum in the top position. The PILCO toolbox produced figures that show the immediate cost over time for every iteration. These figures are shown in Figure 8 and show the value of the immediate cost over time per iteration. The subfigures show the learning process of the policy. The red line is the immediate cost of the PILCO toolbox applying the policy on the simulated model, and the blue area is the expected cost predicted by the PILCO toolbox before running the simulation. The closer the pendulum is to the target state the lower the cost. The cost becomes higher the further the pendulum moves away from the target state. This means that the GP will try to minimize the immediate cost and move the pendulum toward the target state. During the first four iterations the GP had not yet found a working policy to swing the pendulum up. Figure 8(a) shows what happens when the policy decides to input either 3V or -3V. In Figure 8(b) the policy tries to swing the pendulum up. This swinging does bring the pendulum close to $\pm 90°$, which does give a better immediate cost so the policy will try to improve on this. This can be seen in Figure 8(c) where the policy manages to swing the pendulum up, but is not able to stop the pendulum before it falls down. During the fourth iteration, as shown by Figure 8(d), the policy over corrects and the pendulum falls back the way it came. This is why the peaks at zero are bigger than during the third iteration. From the fifth iteration and onward the GP learns a policy that can stabilize the pendulum in the top position, as shown by Figure 8(e). Furthermore, the only difference between Figures 8(e) to 8(j) is a decrease in settling time, i.e. the time to reach the target state. It seems that after the tenth iteration the policy has learned to swing the pendulum up in the smallest time possible. The final policy manages to reach the target state by swinging the pendulum from left to right twice. These swings are hardly visible in the plots of the immediate cost, however Figure 8(e) shows the clearest drop in cost before moving to an upright position, which is the second and largest swing the pendulum makes. These swings are not clearly visible without magnifying the curves because the immediate cost decreases exponentially the closer the pendulum gets to the target state. Swings below $90°$ are barely visible in the scope shown in, for example, Figure 8(b) where the immediate cost did not go below 0.92 even though it did reach $\pm 90°$.

B. implementing the policy into Simulink

After the PILCO environment was adapted sufficiently to learn a policy that was able to stabilize the system at the target state, a method had to be found to train the policy using the actual system. Methods using the Matlab `sim()` function have been explored and tried, but to no avail. Thereafter, the dynamics from the Simulink model were implemented in the PILCO environment by means of implementing the same mathematical equations and using the `ode45` solver to simulate the pendulum's behaviour. Also the same additive white Gaussian noise was added to the two states. This method proved successful in learning a policy that could be exported and implemented on the Simulink model to compute a control signal. The following paragraphs describe the process of this implementation.

The first thing that was done to test if the policy obtained could work in the Simulink model was to give the Simulink model the same inputs as the last iteration of the PILCO model, i.e. applying a recorded time-series of inputs. The difference in angle, can be found in Figure 9. So, these system responses are generated by applying the same input that the PILCO model used to produce the responses in Figures 8(a) to 8(e). To achieve this, the control signal was imported into Simulink using the "From Workspace" module. To be able to do this, a time variable had to be added to the input variables. Figure 9(a) shows that the error between PILCO's response and the Simulink model's response is around $0.05°$if the pendulum only gets a constant value for its input. The difference between the angle obtained from the PILCO and Simulink simulation is small enough to assume that the



(a) Iteration 1 (b) Iteration 2

(c) Iteration 3 (d) Iteration 4

(e) Iteration 5 (f) Iteration 6

(g) Iteration 7 (h) Iteration 8

(i) Iteration 9 (j) Iteration 10

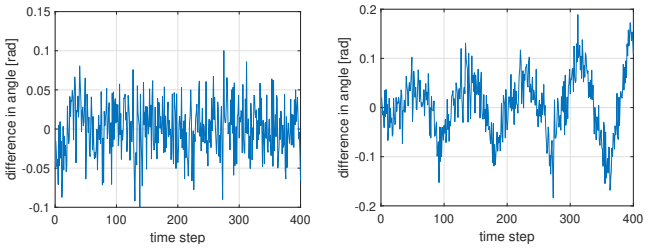
Fig. 8. The immediate cost over time using the PILCO toolbox for GP-based policy learning.

dynamics of the system are the same. Naturally, the error between the two responses can never constantly be 0°due to the two independent noise realizations between the two simulations. However, Figure 9(b) shows that the error will get bigger over time when a time-varying input value is applied to the system, so the slight differences in the angle at the start of the simulation continue to accumulate over time. Yet, the error is still below 0.2° so this shows that the dynamics of the Simulink model somewhat follows the PILCO model.
These lower error values do not hold for Figures 9(c) and 9(d), which shows that the error of the angle becomes a lot higher. A commonality is seen when looking at the points in Figures 8(c) and 8(d) where the same error starts to occur when the pendulum is around the target of 0°. When the PILCO simulation reaches the target state, the error in angle between the PILCO and the Simulink simulations goes just over a critical point. Either the Simulink simulation does not make a full circle and the PILCO simulation does, or the Simulink simulation makes a full circle while the PILCO simulation does not. When this happens, the difference in angle will become unpredictable. Subsequently the angle that is given by a certain input voltage is completely different between the two simulations. This effect is also slightly visible in Figure 9(e) where the pendulum stays upright in the PILCO simulation and not in the Simulink simulation because of the difference in noise. This causes the pendulum to fall down from 0° to 180° which is at the bottom of the pendulum.

Given the input of the PILCO model to the Simulink model and getting a somewhat satisfactory result, gives a reason to assume that the learned PILCO policy could potentially work on the actual Simulink model when properly implemented in the feedback loop. To validate this hypothesis, the policy obtained in the PILCO simulation has to be implemented and run on the Simulink simulation.

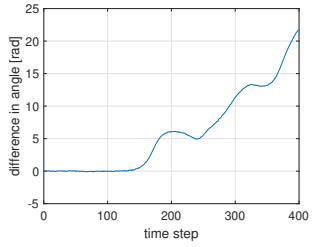
Multiple methods were tried to extract the learned policy and implement it into Simulink. The first method that was looked into was if the PILCO policy could be imported into Simulink. This did not work because a Matlab `struct` cannot be implemented into the Simulink model itself. The second method, was a Matlab script. This could be implemented, but all the values given by the script have to be given before the simulation runs, so it is not able to receive feedback even if the script is a function. The third method was to use a *Matlab System* block in Simulink, which enables the implementation of algorithms using the Matlab language. It was not necessary to give the output values to the Matlab System before running the simulation. The output values were calculated and given while running the simulation. To test if the Matlab system worked as expected, the output was set to the angle divided by the angular velocity. The third method produced a visible output conforming that the Matlab System could be used.

The second problem to prove that the PILCO policy would work on the Simulink model was to successfully implement the policy into the Matlab System. The first focus was being able to import the policy `struct` into the Matlab System. This can be done by making a tunable property in the system

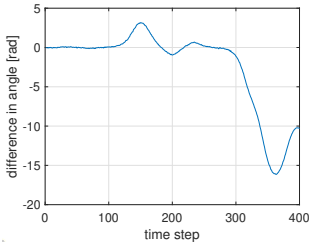


(a) Iteration 1

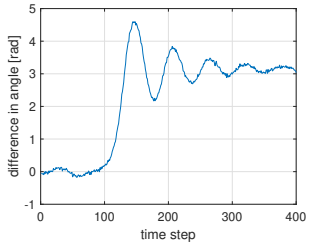
(b) Iteration 2



(c) Iteration 3



(d) Iteration 4



(e) Iteration 5

Fig. 9. The difference between the PILCO and Simulink model

that contains the policy `struct`. When the property is not predefined as a `struct`, the system will not allow a `struct` to be used for that variable.

A third problem arrived when the PILCO policy was imported into the Simulink model. The system did not allow for imported `structs` to have functions. To work around this problem, the function within the policy was put to zero. For convenience, a separate Matlab script was written that loads the parameters needed and then removes the function of the policy. The next step was making the system output the correct voltage according to the policy. The first thing to find out was how the PILCO toolbox calculates the required voltage. This is done in the `rollout` script with the code in Algorithm 1.

In this algorithm the first line `s = x(i,dyno)'` sets the angle and angular velocity of the pendulum at time instance i. For this reason when the code was implemented into the Matlab script `x(i,dyno)'` was replaced with `[velocity; angle]`. `gTrig(s, zeros(length(s)), angi)` computes of the angle and angular velocity: the mean, covariance matrix and the covariance. The variable `angi` is from the plant `struct` and indicates which variables are needed for this calculation. The line

Algorithm 1: PILCO Matlab code to calculate input voltage from policy

```
s = x(i,dyno)';
sa = gTrig(s, zeros(length(s)), angi);
s = [s; sa];
x(i, end - 2 * nA + 1 : end) = s(end - 2 * nA + 1 :
  end);
u(i,:) = policy.fcn(policy, s(poli), zeros(length(poli)));
```

`x(i, end - 2 * nA + 1 : end) = s(end - 2 * nA + 1 : end)` adds the current angle and angular velocity to the variable x. This line is not necessary to implement the code in Algorithm 1 into Simulink because the Simulink model gives the angle and velocity at every time step. The last line of code `u(i,:) = policy.fcn(policy, s(poli), zeros(length(poli)))` computes the input that needs to be given to the pendulum at each time step. Here `policy` is the policy `struct` and `poli` is a variable from the plant `struct` that serve as inputs to the policy. The code mentioned above is the most important code to implement into the Matlab System. But because a Matlab system does not allow for a function to be part of the `struct`, the function was looked up in the `struct` and directly placed into the Matlab System. The code of the Matlab System can be found in Algorithm 2

Algorithm 2: Matlab sytem code to calculate input voltage from policy

```
s = [velocity; angle];
sa = gTrig(s, zeros(length(s)), angi);
s = [s; sa];
u = conCat(@congp, @gSat, obj.policy, s(poli),
  zeros(length(poli)));
```

In this algorithm the line `u = ConCat(@congp, @gSat, obj.policy, s(poli))` is the replacement of `u(i,:) = policy.fcn(policy, s(poli), zeros(length(poli)))`. This is because the function that `policy.fcn(policy, s(poli), zeros(length(poli)))` calls upon is equivalent to the newly replaced code. The only difference is that the new code can be run in the Matlab system.

Another encountered problem was that the PILCO code initialised new variables to make reading the code easier without first initializing the variables to zero. The errors caused by this problem were fixed with a combination of replacing the easier to read variables and initializing the other variables with the correct dimensions. After all errors were fixed, the PILCO policy was able to be implemented but not all the needed variables were yet imported into the system. The only variable that was still needed was the PILCO plant `struct` that contains the variables `angi` and `poli`. All of the functions of the plant `struct` were removed. The Matlab script that removes the function for the policy `struct` was expanded

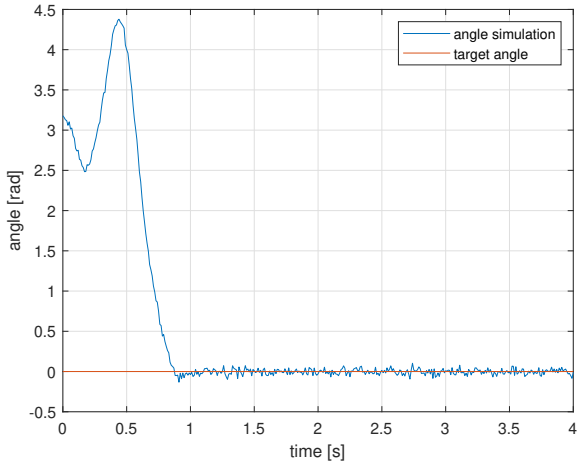


Fig. 10. The angle of the pendulum in Simulink

to include the plant `struct`. The plant `struct` was further implemented in the same way as the policy `struct`.

After all errors were removed and all the variables were added, the Simulink simulation was able to use the policy to get the pendulum to stay still at the target angle. This can be seen in Figure 10.

The downside of all the changes to the PILCO simulation is that this simulation can not be used to learn a different policy.

IV. Learning a Multi-task Policy

Now the policy is trained to solve multiple tasks. In this assignment, the approach for multi-task learning is by mean of a richer policy parametrization. [1] With this Multi-task Policy Search (MTPS) approach, a single policy can be learnt jointly given a set of different tasks. $\pi_i(x, \eta_i, \theta_i)$ to optimize the trajectory and loss function $J_\pi(\theta)$.

A. Swing up to different targets

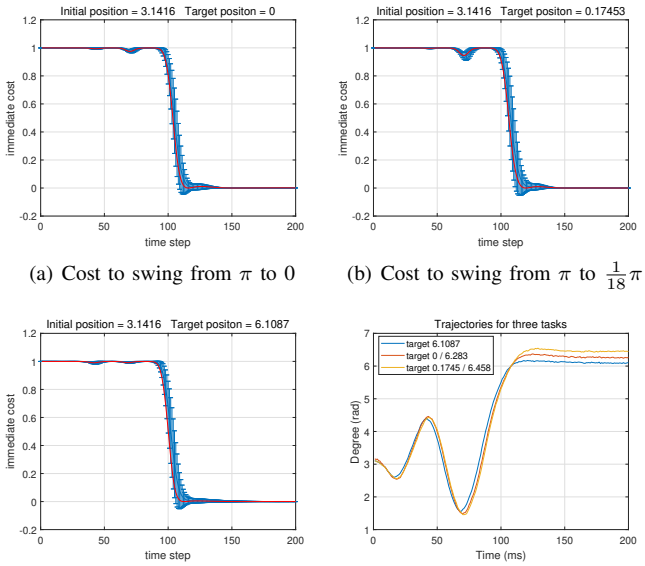
The first attempt of MTPS is to extend the task trained in previous section into three different tasks: swing up from π to 0; from π to $\frac{1}{18}\pi$; and from π to $\frac{35}{18}\pi$.

After the policy training with the initial rollout, a task generation function is implemented in to each learning iteration. To define a new task, there are two elements that requires a update: initial position and target position, '`mu0`' and '`cost.target`' respectively in PILCO. For these three swing-up-tasks, the initial position are identical. Thus, a target position array is defined with three different targets: `[ 0` $\frac{1}{18}\pi$ $\frac{35}{18}\pi$ `]`. For each iteration, a random target position `mu1` is assigned to `cost.target` for training. In Algorithm 3, the concept of the stochastic target swing-up policy is presented.

The pictures presented in Figure 11 demonstrate the policy training result. Figure 11 (a), (b) and (c) are the policy validation results by means of the corresponding immediate costs. All cost functions are converged to 0, which means all three target position can be achieved in 2 second. Figure 11 illustrates the trajectories of approaching targets. As depicted

Algorithm 3: Stochastic target swing-up policy

init: Define rollout task: initial position `mu0`, target position `mu1`. Learn the initial policy π_0 with the generated trajectory rollout x_0.
repeat
  `getnewtask`: Update '`cost.target`'
  `trainDynModel`: Train GP model
  `learnPolicy`: Update policy π_i
  `applyController`: Generate rollout x_i with π_i
until Reaches N iterations
Test trained policy π_*



(a) Cost to swing from π to 0

(b) Cost to swing from π to $\frac{1}{18}\pi$

(c) Cost to swing from π to $\frac{35}{18}\pi$

(d) Trajectories for three tasks

Fig. 11. Policy validation from multi-target swing up.

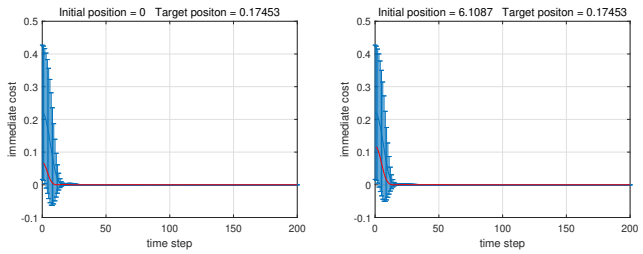
in Figure 11, the policy is learnt after 20 iterations. In Figure 8(e), the single target policy obtains a decent learning performance at the fifth iteration. Thus, a training process with 20 iterations is implemented for MTPS. Since the -10 degree target from up-right position is defined with $\frac{35}{18}\pi$, the policy chooses to approximate the target 0 and $\frac{1}{18}\pi$ with 2π and $\frac{37}{18}\pi$ respectively.

Unlike the MTPS approach in [1], this stochastic target policy is updated based on every single trajectory loss $J^{\pi_i}(\theta|\eta_i)$ instead of the average performance of over tasks $E_\eta(J^\pi(\theta|\eta))$. The reason for the approach is that it is easy to implement as a first step since there is no further modification needed for loss function and gradient calculation. In addition, the given swing-up tasks shares the identical initial starting position and the target positions are adjacent, which bring similarity between task and less difficulty when training the policy.

B. Target to target movement

Approach 1:

From training target to target movement policy, the first step is to define the tasks η^{train}:



(a) Immediate cost track from 0 to $\frac{1}{18}\pi$ (b) Immediate cost track from $\frac{35}{18}\pi$ to $\frac{1}{18}\pi$

Fig. 12. Policy training from multi-target movement.

- Initial position: $[\,0\ \frac{1}{18}\pi\ \frac{35}{18}\pi\,\pi]$
- Target position: $[\,0\ \frac{1}{18}\pi\ \frac{35}{18}\pi\,]$

With the new task definition, the multi-task policy trained for swing-up is introduced to conduct a trial learning for 30 iterations. After 20 iteration, there is some progress made by this approach as presented in Figure 12.

Since the training takes approximately 20 minutes per iteration, the optimal policy is still under training. However, given the result in Figure 12, the confidence in the outcome is acquired.

Approach 2:

In [1], another approach of MTPS is offered. With the single task learning only one optimal policy π_* and optimized by minimizing the loss function

$$J_\pi(\theta) = \sum_{t=1}^{T} E[c(x_t)|\theta] \tag{8}$$

However, in MTPS process, each individual task η_i has its own corresponding information $u_i = \pi_i(x_i, \eta_i, \theta_i)$. Thus, the overall expected long-term cost $J_\pi(\theta)$ is approximated by averaging over all tasks η_i [1]:

$$J_\pi(\theta) = E[J_{\pi_i}(\theta_i)] = \frac{1}{M}\sum_{i=1}^{M} E[c(x_t)|\theta_i] \tag{9}$$

where the number of M represents the number of each individual task η_i applied in the training. In Equation (9), the loss function is no longer optimized for one task but optimal across all possible tasks. Thus, the goal is to obtain a task generalization.

Due to the limitation of time and a better understanding of the theory, this approach does not present a convincing results yet.

V. Conclusion

The dynamics of the unbalanced disc system, within the $\pm 120°$ bound, can be accurately captured by both the ANN and GP model. The ANN model uses a NARX structure with two hidden layers of five nodes each. The ANN model used a $300s$ training dataset, while the GP model used only a $30s$ dataset due to memory limitations. The ANN was able to achieve a validation accuracy of 0.0016 for the MSE. The GP on the other hand, achieved a validation accuracy of 0.0043 for the MSE and achieved the best performance when trained with input-output data generated with an RBS input signal compared to a combination of a chirp and an RBS signal. Comparison between the two models showed that the ANN model is more accurate by approximately 269%. However, if only a relatively small training data set is available, for example 30 seconds of data versus 300 seconds, then the GP model performs better by approximately 135%. The accuracy of both models can be further improved if the training data contains more information on high angular velocity states of the system.

The PILCO toolbox was able to learn a swing-up policy that stabilizes the pendulum in the top position. After successful implementation in the Simulink model, the model was also able to give the same result as the PILCO simulation. This validated that the same dynamics implemented in the PILCO toolbox match the dynamics of the Simulink model, including noise. This means that Simulink can use this policy without a difference between the PILCO and Simulink simulation when the PILCO toolbox is used to learn a new policy. The policy obtained with the PILCO toolbox can reach the target state within a second, which is an acceptable settling time. After the policy was implemented in Simulink it has a low error. The error is below 0.1 radian which is the same as the noise added to the system.

For Multi-task policy training, a different approach is applied according to [1]. In the training, stochastic tasks are generated and introduced to the training loop. The result is yet convincing. The policy can bring high success rate with more training iterations. Another interesting implementation can be compare the approach with the one in [1] to check how to improve the training time and success rate.

Future research could investigate the use of different kernel functions, i.e. mean, covariance and likelihood function, to improve the modelling accuracy of the GP model. In this paper, only the most common functions were used to identify the system dynamics. However, the use of a combined periodic and impulse response kernel function might prove to be beneficial for the modeling accuracy. Furthermore, the learned single task policy was able to swing the simulated pendulum upwards and stabilize it, a validation is necessary on the physical system to validate the feasibility of application on a physical set-up. Therefore, future research could investigate the application of the learned policy on the physical unbalanced disk set-up.

References

[1] M.P. Deisenroth, P. Englert, J. Peters, and D. Fox. Multi-task policy search for robotics. In IEEE International Conference on Robotics and Automation (ICRA), pages 3876–3881, Hong Kong, 2014.

[2] Rasmusen, C. E., & Nickisch, H. (2018). The GPML Toolbox version 4.2.

[3] Deisenroth, M. P., McHutchon, A., Hall, J., & Rasmussen, C. E. (2013). PILCO Code Documentation v0.9. Cambridge.