Exploring Unbalanced Disk Dynamics: Model Identification and Non-Linear Control Policy Learning

1st Wenyu Song
std. nr
Eindhoven, The Netherlands
w.song@student.tue.nl

2nd Kevin Laemers
0848982
Eindhoven, The Netherlands
k.laemers@student.tue.nl

3rd Ryo Sugimura
std. nr
Eindhoven, The Netherlands
r.sugimura@student.tue.nl

4th Marten Hanegraaf
std. nr
Eindhoven, The Netherlands
m.hanegraaf@student.tue.nl

***Abstract*—This paper presents an in-depth study on the control of an unbalanced disc system, a practical model for the inverted pendulum system. The system dynamics are identified using a combination of Artificial Neural Networks (ANN) and Gaussian Processes (GP), in conjunction with Nonlinear AutoRegressive with eXogenous inputs (NARX) models. The pendulum swing-up problem is addressed using Deep Q-Network (DQN) and Advantage Actor Critic (A2C) approaches, with the architecture, learning process, and environment setup of both agents detailed comprehensively.The study extends to the design of a single policy for multiple targets using the A2C approach, aiming to move the pendulum to three different positions using a single policy. Despite efforts, challenges were encountered in training the agent for multiple targets. An attempt to implement the PILCO algorithm, a model-based policy search method, in Python was also made, but the agent consistently failed to reach the top position. The paper concludes with potential improvements and future work, which could involve further investigation into the use of PILCO for this problem, potentially using the MATLAB implementation or refining the Python implementation.**

***Index Terms*—Unbalanced Disc System, System Identification, Artificial Neural Networks, Gaussian Processes, Nonlinear AutoRegressive with eXogenous inputs, Deep Q-Network, Advantage Actor Critic, Reinforcement Learning, Policy Learning, Multiple Target Policy, PILCO**

I. Introduction

The study of the inverted pendulum system has garnered significant attention due to its wide-ranging applications in various fields such as unicycles and hoverboards. The primary challenge of this system lies in its inherent instability, necessitating precise control to maintain equilibrium. This assignment paper focuses on an analogous system represented by an unbalanced disc, as shown in Fig.1, which serves as a practical model for the inverted pendulum system.

The unbalanced disc system is actuated by voltage, with the disc's angular position, θ, being a function of the applied voltage, u. The relationship between u and θ is governed by the following equations:

$$\dot{\theta}(t) = \omega(t) \tag{1}$$

$$\dot{\omega}(t) = -{\omega_0}^2 sin(\theta(0)) - \gamma\omega(t) - F_c\text{sign}(\omega(t)) + K_u u(t) \tag{2}$$

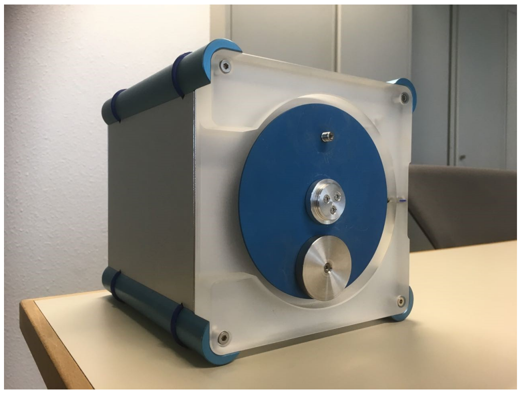

Fig. 1. Unbalanced disc set up

This paper aims to explore the system dynamics of the unbalanced disc system using a combination of Artificial Neural Networks (ANN) and Gaussian Processes (GP) in conjunction with Nonlinear AutoRegressive with eXogenous inputs (NARX) models. The objective is to identify the system dynamics accurately and subsequently develop an effective policy for system control.

II. System Identification

Two methods are introduced in order to identify the unbalanced disk system from the given dataset of voltage input and angle output. These two methods include Gaussian Processes (GP) and artificial neural networks (ANN).

A. Gaussian Process

Besides form ANN, a probability-based non-parametric model called Gaussian Process can also be used to model functions.

1) Gaussian Process: The Gaussian Process (GP) assumes that the relationship between data points obeys a multivariate Gaussian distribution. The core idea of GP is to describe the distribution of the function through the mean function and the covariance function:

$$f(x) \sim GP(m(x), \kappa(x, x')) \tag{3}$$

Where the mean and covariance can be described with the following functions:

$$\begin{aligned} m(x) &= \mathbb{E}\{f(x)\} \\ k(x,\tilde{x}) &= \mathbb{E}\{(f(x)-m(x))(f(\tilde{x})-m(\tilde{x}))\} \end{aligned} \tag{4}$$

The covariance function $k(x,\tilde{x})$, e.g. kernel function, is the most crucial part in GP. Different kernel function can capture different types of relationships. Given a set of observed data, Gaussian Process can be used for inference to obtain the conditional distribution of the target function. By calculating the posterior distribution, we can obtain predictions of function values and corresponding uncertainties given the observed data.

2) NARX model: NARX (Nonlinear AutoRegressive with eXogenous inputs) is a non-linear autoregressive model commonly used for time series modelling and prediction. The NARX model captures the non-linear relationship in time series data while considering the influence of exogenous inputs.

The general form of the NARX model can be expressed as

$$y(t) = f(y(t-1), y(t-2), ..., y(t-n), u(t), u(t-1), ..., u(t-m)) \tag{5}$$

The key to the NARX model is to define the function f, which describes the relationship between the independent variable (output) and past observations and external inputs. The function f can be any nonlinear function, such as a polynomial function, neural network, etc. Choosing an appropriate function form and model structure is the key to building an effective NARX model.

By combining NARX and Gaussian Process regression, NARX GP can capture non-linear dynamics and temporal dependencies using the NARX structure while providing probabilistic predictions and uncertainty estimates using the Gaussian Process regression framework. This hybrid model is particularly useful when dealing with time series data with complex non-linear relationships and uncertain dynamics.

3) Algorithm: In this situation, we used NARX GP and managed to identify the system dynamics according to the given data of input voltage u and output angle θ. The basic logic is shown in Algorithm 1 below. The algorithm was implemented in Python using the GaussianProcessRegressor and kernel functions from scikit-learn.

Algorithm 1 NARX GP

```
1: Load training data from file.
2: Assign the voltage data ($X_N$) and theta (the location of the disk) as ($Y_N$).
3: Create input data ($x(t)$) using the past data $y(t-1), y(t-2), ..., y(t-n), u(t), u(t-1), ..., u(t-m)$ which can be used to predict $y(t)$
4: Create a Gaussian Process regression model (reg) using GaussianProcessRegressor and a kernel function.
5: Fit the training set to the model and estimate the parameters of the Gaussian Process model by calling function reg.fit.
6: Use the fitted model to predict the training set and validation set, and calculate the standard deviation of the predicted results.
```

4) Kernel: Multiple kernels are available for Gaussian process regression namely, Radial Basis Function(RBF) Kernel, Matérn kernel, Rational quadratic kernel, Exp-Sine-Squared kernel, Linear kernel and Dot-Product kernel. Out of these kernels, the RBF kernel sees the most use in practice since the RBF kernel is highly flexible and can capture a wide range of smooth functions.

The Matérn kernel is similar to the RBF kernel with the main difference being that the Matérn kernel is used for estimating functions which can not be differentiated indefinitely. The Rational quadratic kernel also shows similarities to the RBF kernel since it consists of multiple RBF kernels with different length-scales.

The Exp-Sine-Squared kernel, Linear kernel and Dot-Product kernel are all used to approximate a specific type of function. The Exp-Sine-Squared kernel can be used to approximate periodic functions, whereas the linear kernel excels in approximating linear functions and the Dot-Product kernel in exponential functions.

Based on the characteristics of these kernels, the most applicable kernels are the RBF and Exp-Sine-Squared kernel. This can be attributed to the nature of the relation between the angular position and the applied voltage. In testing, both these kernels showed almost identical performance. However, the Gaussian Process regressor with the RBF kernel was slightly faster. Because of this, the RBF kernel was selected.

In addition to the RBF kernel, a white kernel was added. White kernels are often used to account for observation noise. This addition helps since the output from the unbalanced disk setup is likely to be subject to some noise in the measurements.

5) Hyper-parameter tuning: The success of a model is hugely dependent on picking the right hyper-parameters. Hyperparamete

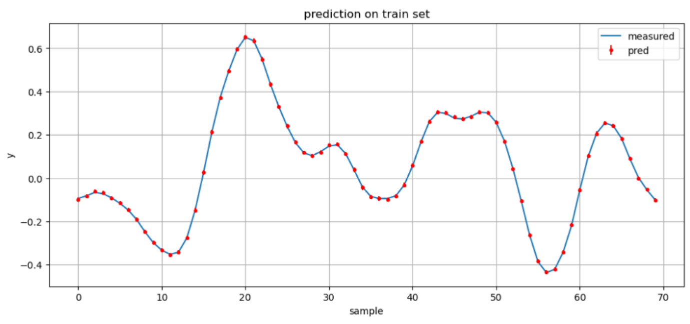

Fig. 2. Prediction on training set

6) Result: As shown in Fig2 and 3, a small uncertainty indicates a successful Gaussian process, implies that the GP model fits the data well and provides reliable predictions.

B. Artificial Neural Network

1) Neural Network: An artificial neural network (ANN) is a black box system that can be used for approximation of both linear and non-linear dynamics of systems. An artificial neural

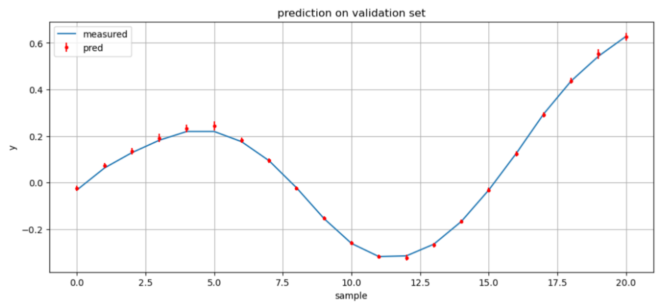

Fig. 3. Prediction on validation set

network is connected by nodes which calculate based on the input, weight and biases. The output from this is fed into the non-linearity function. The equation for a node is shown in the equation below.

$$y = \sigma(w^T x + b) \tag{6}$$

The inputs are multiplied by the weights and bias is added. The output of that is fed into a function. For the structure, a multi-layer perceptron with the non-linearity function was used. The base structure of the model was decided on further research of system identification using neural networks. However, since the system dynamics are different to the one in the paper and hidden layers may over-complicate the model resulting in lower identification accuracy, further hyperparameter research was done. The relation between the number of hidden layers and identification loss was further researched to finalize the structure of the model. For the unbalanced disk, the best number of hidden layers was found to be 2. Using the ANN structure that was defined, the NARX and recurrent neural networks (RNN) along with the state space model were considered.

As mentioned before (II-A2), the NARX model can be used in conjunction with ANN where the functions can be approximated using the nodes and hidden layers. For the NARX model, the n and m mentioned previously are hyperparameters that can be tuned. These hyperparameters are looked further into in the form of a grid search. The combination of the parameters in a certain range is taken to see how it affects the loss.

Along with the NARX model, we incorporated a non-linear state-space model which is another way of describing the dynamics of a system. The state-space models are given by the equations below.

$$\dot{x} = f(x, u) \tag{7}$$
$$y = g(x, u) \tag{8}$$

The y, x and u correspond to the output state, input state and action respectively. The f and g are the non-linear functions that express the relation between the given input state/action to each output. The number of states and actions that are the input for state-space models is another kind of hyperparameter and has to be optimized. This can also be optimized similarly to the NARX with a grid search. However, there is another parameter that comes into play when creating the state-space model which is the order of the model. Similar to the ANN used in NARX, the state space model can be over-complicated when having higher orders. Therefore, the order of the state space model is decided using the grid search which has been done for other parameters.

A recurrent neural network is similar to the NARX model in the sense that it takes in past outputs. However, the biggest difference is that the RNN only takes that exact timestep and no past actions as input.

2) Results: With regards to the hyperparameter optimization of the ANN structure, the loss with respect to the number of hidden layers of the model is shown in Figure 4.

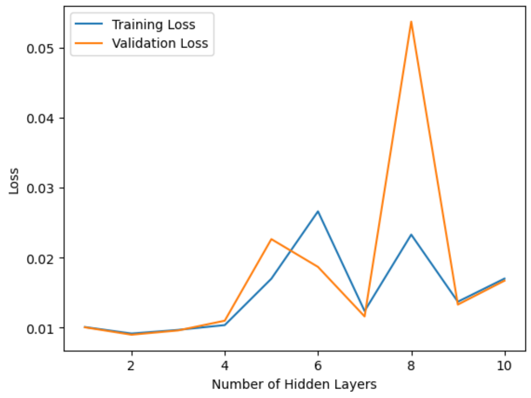

Fig. 4. Effect of numbers of hidden layers on loss

Looking at the results, the optimal number of hidden layers is two. The number of hidden layers is one component that defines how complex the ANN models are, having an optimal number is needed for the best identification. A high number of hidden layers makes the model complex making the model tend to overfit to the data, while lower hidden layers are simple and underfit.

In the hyperparameter optimization for the number of past outputs and inputs to be taken into account for the next timestep, the results are shown in Figure 5.

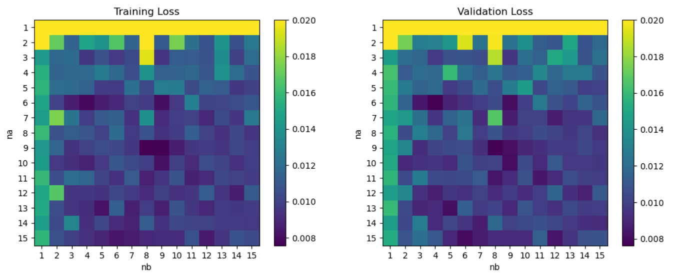

Fig. 5. (Right) Training loss and (Left) validation loss of the NARX model hyperparameter grid search

The parameters with low losses are ones such as the $(n_a, n_b) = (6, 4)$ and $(9, 8)$. However, looking at the training and validation, the areas closer to the latter combination have differences in shades meaning the loss goes up for validation. However, the former combination seems to be a lower loss. This fact can also be supported by the actual loss values where the former loss is 7.51×10^{-3} and 7.62×10^{-3}, but the latter loss is 7.57×10^{-3} and 8.15×10^{-3}.

The model order of the state-space model is researched with grid search and the results of the optimization are shown in Figure 6.

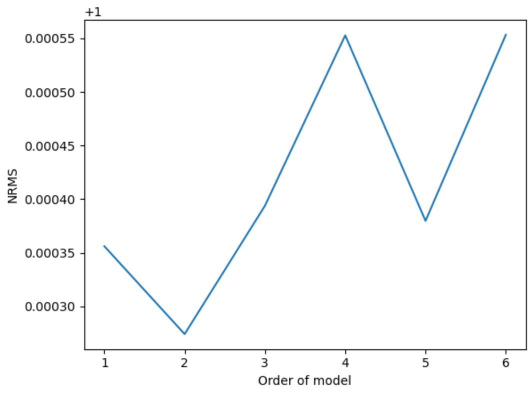

Fig. 6. Effect of model order on the loss

From the results, we can observe that the optimal model order is two. Similar to the hidden layers, the order of the model defines the complexity. Therefore, the number of hidden layers and model order has similar values,

Based on all these tuned hyperparameters the system identification was performed using ANN and three methods as explained above. The results of each of the system identification methods using the test dataset are shown in Figure 7, Figure 8, **??**.

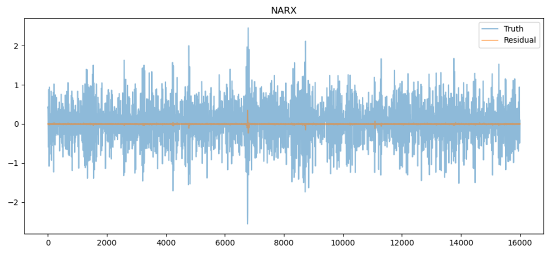

Fig. 7. Residual of the NARX ANN model

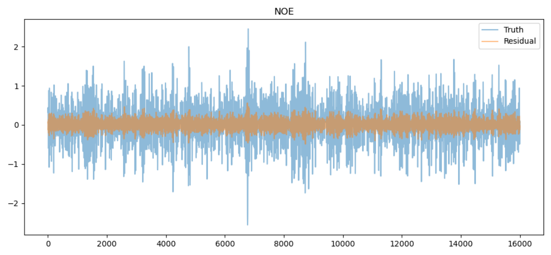

Fig. 8. Residual of the RNN model

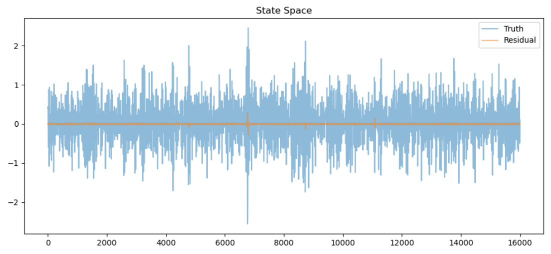

Fig. 9. Residual of state-space model

Given the tuned hyperparameters, the NARX ANN model was able to identify the system with very high accuracy with small errors. However, the RNN was not able to identify the system accurately. However, this was to be expected since in the hyperparameter tuning of n and m, the loss of the model at low values was higher compared to the optimal values. Moreover, since the state-space model has the optimized number of past inputs/outputs and model order, the state-space model has better identification.

III. Swing-up policy

This section outlines the methodology used to solve the pendulum swing-up problem using a Deep Q-Network (DQN) approach and an A2C approach. Both DQN and A2C agent's architecture, learning process, and the environment setup are described in detail.

A. Deep Q-Network (DQN)

1) DQN Architecture: The DQN agent comprises two primary components: the Q-Network and the target network, both of which are implemented as neural networks. The Q-Network is responsible for approximating the Q-value function, which represents the expected future rewards for each action in each state. The target network, on the other hand, is used to generate the target Q-values for training the Q-Network. The architecture of these networks includes two fully connected layers with 64 neurons each, followed by a dropout layer to prevent overfitting. The output layer of the network has a dimension equal to the number of possible actions. The Rectified Linear Unit (ReLU) function is used as the activation function for the neurons.

2) DQN Learning Process: The DQN agent learns by interacting with the environment and storing the experiences in a replay buffer. Each experience is a tuple that includes the current state, the action taken, the reward received, and the next state. The agent samples a batch of experiences from the replay buffer and learns by adjusting its Q-Network parameters to minimize the difference between the predicted Q-values and the target Q-values. This difference, known as the temporal difference error, is calculated using the mean squared error loss function. The learning process also involves a technique known as soft updates to update the parameters of the target network, which helps to stabilize the learning process. This whole process is laid out in Algorithm 2.

Algorithm 2 Enhanced Deep Q-Network with Experience Replay and Dropout

```
 1: Initialize replay memory $D$ with capacity $N$
 2: Initialize Q-network $Q$ with weights $\theta$, adding dropout layers
 3:     for prevention of overfitting
 4: Initialize target network $\hat{Q}$ with weights $\theta^- = \theta$
 5: for episode = 1 to $M$ do
 6:     Initialize state $s$
 7:     Set $\epsilon$ according to decay strategy
 8:     for time-step = 1 to $T$ do
 9:         With prob. $\epsilon$ select random action $a$, else $a = \arg\max_a Q(s, a; \theta)$
10:         Execute action $a$ in emulator and observe reward $r$, next state $s'$
11:         Store experience tuple $(s, a, r, s')$ in $D$
12:         Sample random mini-batch of tuples $(s, a, r, s')$ from $D$
13:         Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s', a'; \theta^-) & \text{otherwise} \end{cases}$
14:         Perform a gradient descent step on $(y_j - Q(s, a; \theta))^2$
15:             with respect to $\theta$, using mixed precision training
16:         Every $C$ steps perform a soft update on $\hat{Q}$ with
17:             $\tau * Q + (1 - \tau) * \hat{Q}$
18:         $s = s'$
19:     end for
20: end for
```

3) DQN Training: The DQN algorithm, as implemented, employs a Q-Network and a target network, both of which are neural networks with two hidden layers of 64 neurons each. The networks are trained using a mean squared error loss function, and the Adam optimizer is used for weight updates. An epsilon-greedy policy is used for action selection, with epsilon decaying over time to balance exploration and exploitation. The DQN agent's memory, implemented as a deque, stores past experiences for replay, enhancing the learning process. The agent is trained over a number of episodes, with each episode consisting of several steps. In each step, the agent selects an action, observes the next state and reward, and stores this experience in its memory. The agent then samples a batch of experiences from its memory and learns from them.

B. Advantage Actor Critic (A2C)

1) A2C Architecture: The actor and critic are implemented as separate neural networks, each with two hidden layers of 64 neurons and a dropout layer for regularization. The actor network outputs a probability distribution over actions, while the critic network outputs a value estimate.

2) A2C Learning Process: The agent uses the actor network to select actions and the critic network to evaluate them. The agent learns from its experiences by storing them in a replay buffer and sampling mini-batches to update the networks. The actor network is updated to maximize the expected return, and the critic network is updated to minimize the difference between its value estimates and the actual returns. The full behaviour is worked out in Algorithm 3.

3) A2C Training: The A2C algorithm, on the other hand, employs two separate networks: an actor network for policy estimation and a critic network for value estimation. Both networks are implemented as neural networks with two hidden layers of 64 neurons each. The actor network outputs a probability distribution over actions, from which an action is sampled, while the critic network estimates the value of the current state. Both networks are trained using the Adam optimizer. Unlike the A2C agent, it does not use experience replay, as it is an on-policy method. The agent is trained over a number of episodes, with each episode consisting of several steps. In each step, the agent selects an action, observes the next state and reward, and updates based on this information.

Algorithm 3 Advantage Actor-Critic (A2C)

```
 1: Initialize actor network with weights $\theta$ and critic network with weights $\phi$
 2: Initialize memory $D$ with capacity $N$
 3: Initialize batch size $B$, discount factor $\gamma$, and learning rate $\alpha$
 4: Initialize optimizer for actor and critic with learning rate $\alpha$
 5: for each episode do
 6:     Initialize state $s$
 7:     for each time step do
 8:         Select action $a$ from actor network given state $s$
 9:         Execute action $a$ in the environment and observe reward $r$ and next state $s'$
10:         Store transition $(s, a, r, s')$ in $D$
11:         if $|D| > B$ then
12:             Sample random mini-batch of transitions $(s, a, r, s')$ from $D$
13:             Compute critic loss: $L_c = (r + \gamma V(s'; \phi) - V(s; \phi))^2$
14:             Compute actor loss: $L_a = -\log \pi_\theta(s, a)(r + \gamma V(s'; \phi) - V(s; \phi))$
15:             Update $\phi$ by minimizing $L_c$ using optimizer and GradScaler
16:             Update $\theta$ by minimizing $L_a$ using optimizer and GradScaler
17:         end if
18:         $s = s'$
19:     end for
20: end for
```

C. Environment Setup

The agent is trained in a simulated environment that models the dynamics of the pendulum. The state space of this environment includes the angle and angular velocity of the pendulum. Using these parameters, a reward function is constructed as follows:

$$
\begin{aligned}
\theta &= \arctan\left(\frac{\sin(\theta)}{\cos(\theta)}\right) \\
r_{\text{angle}} &= \left(3\exp\left(-\frac{(cos(\theta)-\mu)^2}{2\sigma_{\text{angle}}^2}\right)\right)^2 \\
r_{\text{peak}} &= 3\exp\left(-\frac{(cos(\theta)-\mu)^2}{2\sigma_{\text{peak}}^2}\right) \\
p_{\text{action}} &= 0.001 \cdot |u|^2 \\
p_{\text{velocity}} &= 0.003 \cdot |\omega|^2 \exp\left(-\frac{(cos(\theta)-\mu)^2}{2\sigma_{\text{velocity}}^2}\right) \\
r &= r_{\text{angle}} + r_{\text{peak}} - p_{\text{action}} - p_{\text{velocity}}
\end{aligned}
\tag{9}
$$

where:

- θ is the angle of the pendulum, calculated from the sine and cosine of the angle.
- μ is the mean of the Gaussian functions used in the reward function, set to -1 as the goal is to swing the pendulum up to the upright position where $\cos(\theta) = -1$.
- σ_{angle}, σ_{velocity}, and σ_{peak} are the standard deviations of the Gaussian functions used in the reward function, determining the width of the Gaussian functions. Their values are chosen to reflect the spread or tolerance around the desired mean, with a smaller value leading to a narrower peak and thus a stricter reward or penalty. The values were chosen as follows:

$$
\begin{aligned}
\sigma_{\text{angle}} &= 0.6 \\
\sigma_{\text{velocity}} &= 0.1 \\
\sigma_{\text{peak}} &= 0.001
\end{aligned}
\tag{10}
$$

- r_{angle} is the angle reward, which encourages the agent to swing the pendulum up to the upright position. It's a Gaussian function of the cosine of the pendulum's angle, peaking at $\cos(\theta) = -1$.
- r_{peak} is the peak reward, a Gaussian function providing a high reward when the pendulum is very close to the upright position. It serves as a fine-tuning reward to give the agent an extra incentive to hit the exact upright position.
- p_{action} is the action penalty, proportional to the square of the action, discouraging the agent from taking large actions.
- p_{velocity} is the velocity penalty, a Gaussian function of the cosine of the pendulum's angle, multiplied by the square of the angular velocity $omega$, discouraging high angular velocity near the upright position. This is important to prevent wild swings and to achieve more smooth and controlled movements.
- r is the total reward, the sum of the angle reward and the peak reward, minus the action penalty and the velocity penalty. The goal of the agent is to maximize this total reward, which would mean swinging the pendulum up to the upright position in a controlled manner and stopping there.
- u represents the input voltage that powers the internal motor. Given that u lies within the range $[-3, 3]$, and considering the squaring of the action in the penalty term, smaller input values are incentivized, thus promoting energy efficiency and smoother control actions.

To sum it up, the applied approach consists of training a DQN agent to derive an optimal control policy for the swing-up problem of a pendulum, based on interactions with a simulated environment. The agent's performance is measured in terms of average reward accumulated over numerous episodes, and the learned policy is subsequently implemented in a real-world scenario. Designing the reward function is an intricate task, often involving a process of iterative refinement and adjustments. One potential enhancement to the reward function could be the incorporation of a sparsity reward that accounts for the number of input actions instead of their magnitude. However, this enhancement was not implemented anymore due to the late discovery of this idea.

D. Learning Curves

To visualize the performance of our models during the training process, we plotted learning curves for both the DQN and A2C algorithms. The x-axis represents the number of training episodes, and the y-axis represents the reward obtained in each episode. Due to the nature of the unbalanced disk problem, hyperparameter optimization has been applied from the start instead of regular training, since hyperparameters have great impact on the convergence of reinforcement learning systems. With this in mind, the total amounts of epochs is reduced, but a Tree Parzen-Estimator (TPE) optimizer is used to pick the hyperparameters for each reduced run.

Hyperparameters of the agent, including the learning rate, dropout rate, discount factor, soft update factor, batch size, maximum number of time steps per episode, and size of the replay buffer, are optimized using the Optuna library. The objective function for the optimization is the average score over the last 100 episodes. The optimization is run for 20 trials, and the best set of hyperparameters is used to train the final agent.

1) DQN Learning Curve: The DQN learning curve illustrates the training progress of the DQN algorithm. The learning curve as seen in Figure 10 shows the agent's performance over multiple episodes, depicting the improvement and convergence of the DQN algorithm using the best hyperparameters as seen in Table I.

TABLE I
DQN HYPERPARAMETER OPTIMIZATION

Parameters	Values
Learning Rate	$4.628173547089761 \times 10^{-5}$
Dropout	0.07756570481404663
Gamma	0.9598977569388523
Tau	0.004450741037889503
Batch Size	256
Max T	4983
Mem	4027

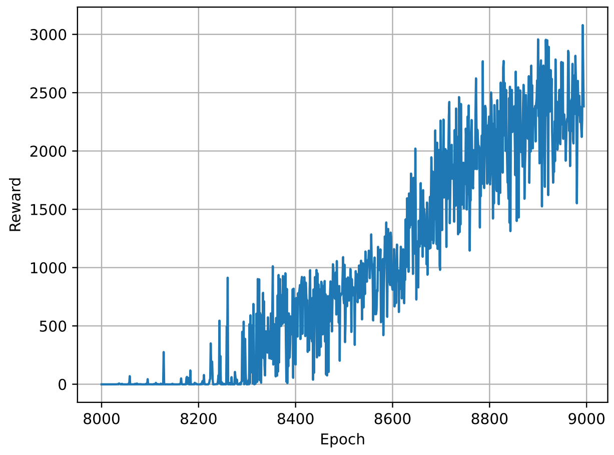

Fig. 10. DQN Learning Curve under best found hyperparameter configuration as seen in Table I. Please note that the epochs start at 8000, since it is part of a larger hyperparameter optimization run.

2) A2C Learning Curve: The A2C learning curve as seen in Fig. 11 showcases the training progress of the A2C algorithm. The best hyperparameter configuration obtained from the optimization process was utilized to train the final A2C agent. The learning curve plot presents the performance of the A2C algorithm over numerous episodes, capturing its learning progress and convergence characteristics as seen in Table II.

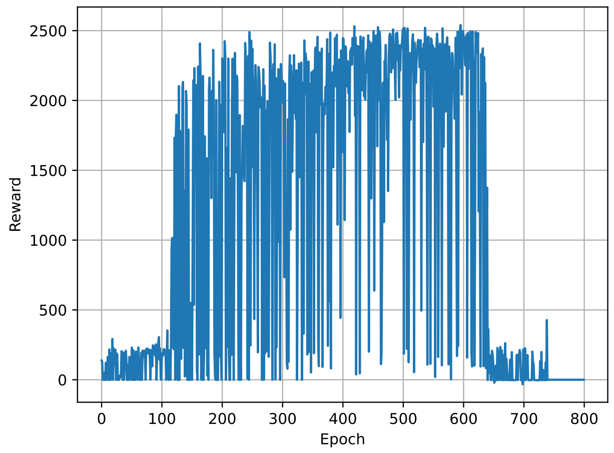

Fig. 11. A2C Learning Curve under best found hyperparameter configuration as seen in Table II.

E. Mitigating the Simulation-to-Reality Gap

The discrepancy between simulation and reality often renders DQN and A2C models, trained in simulation, ineffective when transferred directly to physical systems. However, the introduction of parameter perturbations serves as a viable

TABLE II
A2C HYPERPARAMETER OPTIMIZATION

Parameters	**Values**
Learning Rate	$7.399100501524246 \times 10^{-4}$
Dropout	0.021030343699059095
Gamma	0.9295846438644592
Max T	3058

strategy to bridge this gap. By introducing variations into the simulation parameters, an environmental augmentation process analogous to domain randomization is enacted, which effectively diversifies the learning scenarios. Such a methodology improves the development of a model that is conditioned to function efficiently under a wide range of conditions, thus also on the set of available physical systems, which all slightly vary.

At the start of each episode, a random set of parameters is selected. Parameters subjected to this randomization process includes the base frequency ω_0 of the pendulum in rad/s, dynamics friction coefficient γ, the input proportionality constant K_u, Coulomb friction coefficient F_c, and speed dependent friction ω_c. A random perturbation of 10% is added around the base value of each parameter, creating a uniform distribution within this specified range.

Furthermore, training on the physical system is performed to improve the behaviour. However, when judging both visually and from the reward score, the system decreased in performance after training. The hyperparameters used on the simulation that were deemed effective are now ineffective as seen in Figure 12. In the future, it would be advisable to also perform hyperparameter optimization with limited amounts of episodes per trial on the physical system.

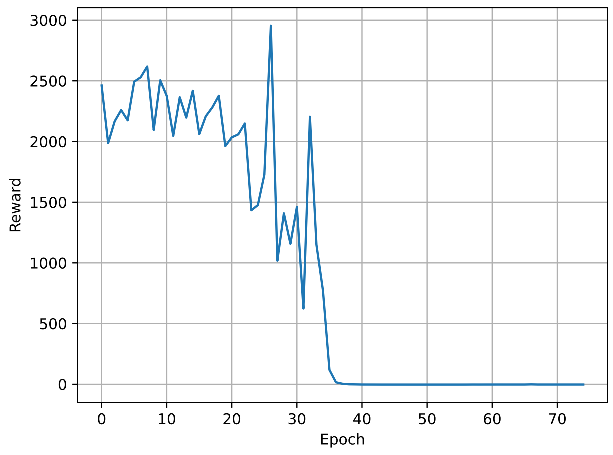

Fig. 12. The reward after retraining the already trained model on the physical setup.

IV. Learning a Single Policy for Multiple Targets

The final part of this study involved extending the swing-up policy to a multiple target policy using the A2C approach. The goal was to design a policy that could move the pendulum from the bottom position to the upright position, and two other positions $\pm 10^\circ$ to the left and the right of the upright position using a single policy. Furthermore, the aim was to improve the obtained policy such that it could also move from one target position to another using one policy, both for swing-up and target-to-target movement.

A. Advantage Actor Critic

To achieve this, changes were made to the reward function of the A2C algorithm. The reward function was designed to encourage the agent to reach and maintain three different positions depending on the number of steps that have passed. For the first 100 steps, the agent is rewarded for reaching the upright position. Between 100 and 200 steps, the agent is rewarded for reaching a position 10 degrees to the left of the upright position. After 200 steps, the agent is rewarded for reaching a position 10 degrees to the right of the upright position.

The reward function is based on a Gaussian function, with the mean shifting according to the desired position and the standard deviation controlling the range of positions for which the agent receives a high reward. The function also includes a penalty for high velocity near the top and a penalty for each action, encouraging the agent to use smaller actions and maintain a low velocity. This was done by shifting every μ value by the same amount of degrees for the aforementioned cases, thus extending on Equation 9.

B. Difficulties of Advantage Actor Critic

Despite efforts, difficulties were encountered in training the agent to learn a single policy for multiple targets. The primary challenge was maintaining the disc's stability at the top position, with the disc exhibiting a tendency to oscillate within a range of approximately 5 degrees. Moreover, while the implementation was successful in achieving a swing-up, it was less effective at encouraging exploration near the top to discover new peak rewards. This issue could be attributed to several factors, including the design of the reward functions and the choice of hyperparameters. Also the action space could have an impact, since small inputs could prevent the overshooting of the disk.

C. PILCO in Python

In the exploration of different approaches to the problem, an attempt was made to implement the PILCO algorithm, a model-based policy search method, in Python. The PILCO implementation involved defining an environment, setting an initial state, and specifying a target state. A Radial Basis Function (RBF) controller and an exponential reward function were used, with the reward function designed to encourage the agent to reach and maintain the three different positions. The agent was trained over multiple rollouts, with each rollout involving

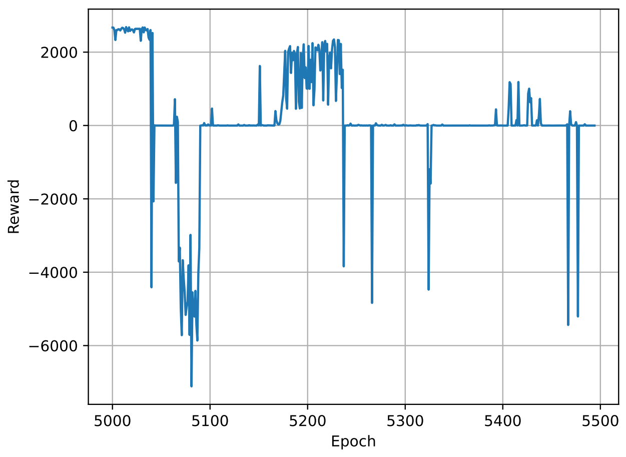

Fig. 13. The reward after transferring the model from a swing-up simulation to the multiple swing policy. Please note that the epochs start at 5000, since it is part of a larger hyperparameter optimization run.

an optimization of the models and the policy, followed by a rollout to gather new data.

However, the Python implementation of PILCO did not perform as expected. The agent consistently failed to reach the top position, instead continuing to swing without achieving the desired stability. This issue persisted even after numerous attempts and adjustments. The exact cause of this issue remains unclear, but potential factors could include the design of the reward function, the choice of hyperparameters, or the specifics of the PILCO implementation in Python.

It was also noted that the MATLAB implementation of PILCO generally performed better than the Python implementation. However, due to time constraints this was not possible.

V. Conclusion

Appendix

Hyperparameter optimization has been applied to many training runs. Including the DQN as seen in Fig. 14.

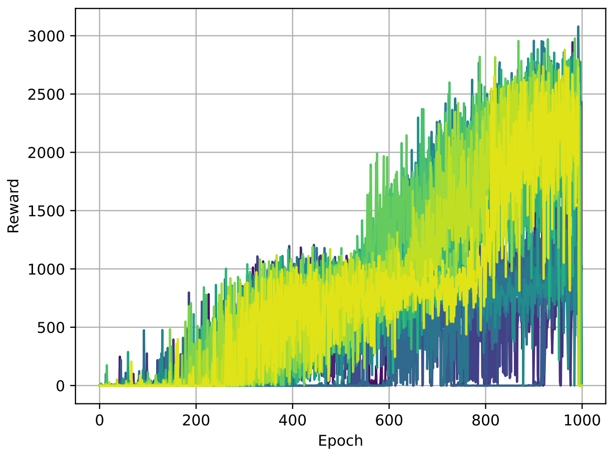

Fig. 14. DQN hyperparameter optimization. Twenty trials are visualized, and it can be seen that some runs converge faster and some runs end up with higher reward values.