Learning Pole Balancing from High-Frequency Tactile Feedback

Erlernen des Balancierens eines Stabes durch hochfrequentes taktiles Feedback
Master thesis in the department of Computer Science by Rolf Gattung
Date of submission: May 26, 2025

1. Review: Jan Peters
2. Review: Niklas Funk
3. Review: Tim Schneider
4. Review: Kai Ploeger
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Rolf Gattung, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 26. Mai 2025

R. Gattung

Abstract

Tactile sensing in robotics is becoming increasingly important, especially for tasks involving grasping. Vision-based tactile sensors have been applied to tasks such as in-hand object pose estimation, but their low-frequency data streams limit their use in dynamic scenarios, which require high-frequency control. A novel high-frequency event-based optical tactile sensor, Evetac, was recently introduced and has shown promise in dynamic tasks such as slip detection and force measurement. However, interpreting event data and exploring the sensor's applicability to broader tasks remains an open challenge. To address this, we investigate the sensor's capability for controlling a freely moving inverted pendulum, a classical benchmark task for control. We use a real robotic system controlled by a Linear Quadratic Regulator and leverage Evetac for tactile-based state estimation of the pole. For event-based state estimation, we explore neural architectures including MLPs, CNNs, and LSTMs, focusing specifically on data preprocessing and the impact of different layers. We detail the system setup, data collection process, model training, and evaluation, demonstrating the potential of event-based tactile sensing for dynamic state estimation and control. We find that incorporating global context and regularization improves the tactile pole state estimation, but limitations in accuracy due to offsets and over-smoothing result in instability during control.

Zusammenfassung

Taktile Sensorik wird in der Robotik immer wichtiger, insbesondere bei Greifaufgaben. Optische taktile Sensoren wurden für Aufgaben wie die Schätzung der Objektposition in der Hand eingesetzt, aber ihre niedrig-frequenten Datenströme schränken ihre Verwendung in dynamischen Szenarien ein, die eine hochfrequente Steuerung erfordern. Ein neuartiger ereignisbasierter optischer Tastsensor mit hoher Frequenz, Evetac, wurde kürzlich vorgestellt und hat sich bei dynamischen Aufgaben wie der Erkennung von Rutschen eines Objekts und der Kraftmessung als vielversprechend erwiesen. Die Interpretation von Ereignisdaten und die Erforschung der Anwendbarkeit des Sensors für breitere Aufgaben bleibt jedoch eine offene Herausforderung. Deshalb untersuchen wir die Fähigkeit des Sensors, ein frei bewegliches umgekehrtes Pendel zu steuern, eine klassische Benchmark-Aufgabe für die Steuerung. Wir verwenden ein reales Robotersystem, das von einem linearen quadratischen Regler gesteuert wird, und setzen Evetac für die taktile Zustandsschätzung des Pols ein. Für die ereignisbasierte Zustandsschätzung untersuchen wir neuronale Netze wie MLPs, CNNs und LSTMs, wobei wir uns besonders auf die Vorverarbeitung der Daten und die Auswirkungen der verschiedenen Schichten konzentrieren. Wir beschreiben den Aufbau des Systems, die Datenerfassung, das Modelltraining und die Auswertung und zeigen das Potenzial der ereignisbasierten Tastsensorik für die dynamische Zustandsschätzung und -steuerung. Wir stellen fest, dass die Einbeziehung des globalen Kontexts und der Regularisierung die Schätzung des taktilen Polzustands verbessert, aber Einschränkungen in der Genauigkeit aufgrund von Offsets und Überglättung führen zu Instabilität während der Steuerung.

Contents

1. Introduction

1.1. Motivation

For humans, tactile sensing is essential and plays a crucial role in everyday tasks involving grasping, such as lifting a cup of water to the mouth. As robotic manipulation of small objects becomes increasingly important, particularly in applications such as assistive robots for daily living or industrial assembly, there is a growing opportunity to enhance these systems by incorporating tactile feedback into a robot's control loop. Amending the perception of robots with tactile sensors and improving their performance has already been done in several works, including estimation of 6-D object-poses [1, 2] or rotating objects in a robotic hand [3].

With tactile information, humans and potentially robots can gain insights into several otherwise difficult-to-estimate properties of unknown objects. As highlighted in [4], a tactile sensor enables us to:

- Detect precisely when contact occurs between the end-effector and the object.
- Extract surface properties, such as shape, material type, contact points, and edges.
- Estimate key reaction properties like friction and impedance, enabling more accurate force control for grasping or holding objects.
- Detect slip forces to prevent objects from falling out of the grasp.

These properties are challenging to estimate, relying solely on external sensors like cameras for several reasons. For example, a camera struggles or fails to detect the exact time of contact between an end-effector and a small object due to the end-effector covering the object. Surface and reaction properties are difficult, if not impossible, to estimate visually without knowing the object type. We need contact and interaction data to make accurate assessments. In particular for humanoid robots performing dexterous manipulation tasks,

incorporating tactile feedback inspired by human sensing and motor actions offers a promising avenue to improve robotic capabilities in e.g., industrial assembly, rehabilitation, or assistance.

A well-established method for validating control algorithms that has been adapted to various scenarios and challenges is the balancing of a pole. For humans, this task is easily performable using the eyes, and a list of related works balancing a pole with a robot, such as [5, 6, 7], can be found in Chapter 2. For a robotic system, the perception of the pole state is typically achieved through a camera or given by simulation. However, tactile information has, to the best of our knowledge, never been used for balancing a pole. For humans, balancing a pole blindly without any visual cues and solely relying on the contact information provided by the nerves on the skin might be achievable with a lot of training. However, it is considerably harder than using the eyes.

Nevertheless, for a robot, it might be possible to achieve pole balancing based solely on tactile information, as previous works have shown reliable pose estimation based on tactile sensors [1] and simple control laws based on an estimated state have been implemented before [8]. For these reasons, the pole balancing task serves not only as a challenging benchmark to assess the potential of tactile sensing for dynamic, contact-rich state estimation, but also as a critical testbed to evaluate the ability of robots to perform complex, real-time control in the absence of visual feedback, highlighting the true capabilities and limitations of tactile-based decision-making in dynamic environments.

1.2. Scope of Thesis

In this master's thesis, we explore the capabilities of estimating the dynamic pole state with an event-based optical tactile sensor called Evetac [9]. We want to investigate the potential of tactile state estimation for real-world pole balancing on the Barrett WAM robotic arm with a completely free pole. For this, we design an LQR control pipeline for the WAM based on visual estimation of the pole state and record the tactile sensor observations during real-world roll-outs of the pipeline, thus creating datasets to learn on. We focused on finding LQR parameters that generate versatile tactile observations while being able to balance the pole and compare their performance in simulation and real life. Second, based on the LQR parameter and settings we tested in the first step, we generate the dataset and train neural networks to learn a mapping from the tactile sensors output to the respective visually estimated pole state. We thoroughly investigate the performance

and evaluate different datasets and network configurations and types with the ultimate goal to replace the visual estimate with our tactile sensor perception.

In Chapter 2, we provide a short review of related works, analyzing previous approaches and techniques for pole balancing as well as summarizing recent tactile sensor applications for pose estimation and developments, especially focusing on the novel tactile sensor we used in this thesis. The mathematical foundations essential for understanding our setup and method are defined in Chapter 3. Our method for balancing the pole and learning the tactile state estimation is described in detail in Chapter 4. Chapter 5 presents the experiments in simulation and the real world, providing results and analyzing them. Finally, Chapter 6 summarizes the results and highlights limitations and future research options for improving upon the results.

2. Related Work

In this chapter, we review relevant prior work. We summarize various approaches to the pole balancing problem, including reinforcement learning, optimal control methods like LQR, and different sensor modalities such as cameras and IMUs. Furthermore, we present a few related papers working with the WAM and OptiTrack, which are hardware and software we used. Finally we present a few recent developments and papers using tactile sensor to estimate a position, aid object manipulation and slip detection.

2.1. Pole Balancing

Since our objective is to balance a 2-dimensional pole on an anthropomorphic robotic arm using visual and later tactile information, we first review prior work in the domain of pole balancing. We highlight different control strategies, setups, data acquisition methods, and hardware used across different papers. Pole balancing is a well-studied benchmark problem in nonlinear control research [10]. Given the position, tilt angle, and their velocities, the inverted pendulum dynamics can be precisely modeled and controlled.

One of the earliest works [11] demonstrates how a neural network trained with reinforcement learning [12] can outperform rule-based controllers for a 1-dimensional cart-pole system in simulation. This line of research is extended in [13], where a self-organizing neural network with eligibility traces (SONNET) is used to control a real robotic system, achieving near-optimal pole stabilization despite unknown physical parameters such as friction. Building on learning-based strategies, Schaal et al. [5, 6] present a method that enables a robot to perform swing-up and stabilization of a 1-dimensional inverted pendulum using Q-learning [12] and Linear Quadratic Regulators (LQR) [14] on an anthropomorphic robot arm with a single human demonstration. Their approach combines human demonstrations with iterative optimal control to adapt to imperfect dynamics models, showcasing the importance of compensating for friction and energy loss. Further

exploring model-based control, [15] compares LQR control against basic state-feedback methods for the inverted pendulum, reinforcing the effectiveness of LQR in such systems.

In contrast to 1-dimensional setups, Gomez et al. [16, 17] tackle more complex configurations, including a double pendulum, consisting of 2 separate 1-dimensional inverted pendulums, and a 2-dimensional inverted pendulum. They use Enforced Sub-populations (ESP) to evolve neural networks capable of inferring velocity from position and angle inputs alone, closely mirroring real-world sensor limitations. Their results demonstrate that even under partial observability, balancing can be achieved in simulation. Expanding into alternative control paradigms, Fadholi et al. [7] apply fuzzy logic [18] instead of binary logic to stabilize a pole on a flying quad-copter, which abstractly is analogous to 2-D pole balancing on a movable plate. Similarly, [19] utilizes fuzzy inference to stabilize a pole on a cart, both when stationary or moving, even on an incline. Addressing data efficiency, Nguyen-Tuong et al. [20] employ Bayesian modeling to achieve swing-up and stabilization after less than $30\,\mathrm{s}$ of interaction time, applicable to both simulation and real hardware. Conradt et al. [8] estimate the state of a pencil with 2 event-based cameras placed with 90-degree disparity around the scene. The system is stabilized via a PD controller on the estimated state, moving the plate under the pencil.

Recent advancements revisit pole balancing through reinforcement learning. Lee et al. [21] introduce a rotational inverted pendulum (RIP) setup with three coupled joints forming a 'Z'-like shape, achieving stabilization using disturbance-rejecting. Bhourji et al. [22] apply deep deterministic policy gradient with proximal policy optimization (DDPG-PPO) to stabilize the RIP in both simulation and real-world experiments. Similarly, [23] deploys a fractional LQR approach for the same task. Combining reinforcement learning with real robot control, Safeea et al. [24] use Q-learning with the KUKA IIWA arm, effectively applying a discretized policy trained in simulation to the continuous real-world system. Klink et al. [25] implement curriculum learning to stabilize and control a spherical pendulum using only position data on the WAM Barrett arm [26], highlighting the role of gradual task complexity in successful training. In another notable setup, Baek et al. [27] stabilize a triple inverted pendulum using model-free RL and virtual experience replay, validating performance in both simulation and real-world hardware. Collectively, these papers highlight diverse approaches to the pole balancing challenge, from classical controllers to modern RL methods, each offering valuable insights for our robotic arm setup involving 2-dimensional balancing in a 3-dimensional space with limited sensory feedback.

Regarding state estimation, if not given by simulation, several works utilize external motion capture systems like OptiTrack [28]. For instance, [29] employs adaptive control

to balance a wheel pendulum while tracking a human, [30] uses a PID controller for stabilization, and [31] demonstrates pole stabilization on a quad-copter using motion capture data. An alternative to visual perception of the pole state and stabilization is given in [32]. The author uses data from an internal measurement unit (IMU) mounted on the pole to estimate the state, succeeding in simulation but not in real-world trials on an anthropomorphic robotic arm.

As highlighted, for most works, perceiving the state of the pole is either given by simulation or commonly estimated via cameras. A few different methods like IMU-based estimation [32] have been investigated, but to the best of our knowledge, no tactile sensor has ever been used for state estimation for pole balancing. To close this gap, during this thesis, we research the capabilities of the novel tactile sensor Evetac [9] for this dynamic task on a real robotic setup like [5, 6]. The Evetac paper is introduced in the upcoming section 2.2, and its behavior is described and explained in section 3.2.4. For the robotic setup, we use the WAM, which has been used as hardware for several works such as [33, 34, 35] involving optimal control, trajectory planning, and system identification.

2.2. Tactile Sensing

For humans, the loss of tactile sensing, such as in individuals with hypoesthesia or those recovering from a stroke, can significantly impair performance, particularly in terms of accuracy and safety. In this context, Tzorakoleftherakis et al. [36] demonstrated how an LQR controller, trained and employed to assist humans in stabilizing a pendulum, could enhance performance by providing tactile feedback that guided hand movements. Similarly, as described in [37], tactile input plays a crucial role in estimating acceleration, which, when combined with visual perception of the pole's pose and velocity, enables effective fingertip balancing for humans.

Tactile sensing plays a crucial role in robotics, enabling human-like grasping by providing information about contact location, object shape, force, and properties like compliance, friction, and mass, especially when the object is occluded by the end-effector, limiting visual input, as mentioned in [4]. This early research shows that tactile sensing enables precise detection of contact events, material types, and slip forces, which are vital for dynamically adjusting grasp forces. [38, 39] review tactile-based algorithms developed for robotic manipulation, including techniques for grasp stability estimation, tactile object recognition, tactile servoing, and force control.

There are a lot of different tactile sensors, namely: Recent examples for using tactile sensing include Lepora [3], who introduced a four-fingered robotic hand with 3-D-printed papillae-like fingertips, enabling in-hand object rotation and human-like dexterity. Lenz et al. [40] further advanced this by integrating visual and tactile feedback for precise object insertion using a robotic arm. Their experiments with the GelSight Mini sensor [41] showed improved success rates when combining both modalities. Specifically for tactile pose estimation, several approaches have been proposed. For instance, [1] introduced a method for estimating the 6-degree-of-freedom (DoF) pose of objects during in-hand manipulation. For this, the authors fuse visual and tactile data using a convolutional neural network, addressing the occlusion issues of vision-only methods. Similarly, Bimbo et al. [42] proposed a covariance-based pose estimation method. Their technique matches tactile data to known object geometries using compact descriptors that encode local geometric features, iteratively refining pose through covariance-based descriptor difference cost function minimization. In another approach, Bauza et al. [43] focused on pose estimation from the first touch. Their method uses contrastive learning to embed tactile RGB observations and match them to a precomputed set of binary contact shapes for known objects, yielding a probability distribution over possible poses.

Li et al. [44] introduced ViTa-Zero, a framework for zero-shot visuo-tactile 6D object pose estimation. Their method refines visually estimated poses using tactile perception through a combination of feasibility checks and test-time optimization, enabling robust estimation without retraining. Kuppuswamy et al. [2] proposed a method that estimates an object's pose by modeling their tactile sensor surface as a deformable membrane. Assuming quasi-static equilibrium between internal stresses and external contact forces, while neglecting friction, they use finite element methods to describe the surface deformation. Their pipeline first identifies the contact area using a pressure-based contact patch estimator. This initial estimate is then refined using the Iterative Closest Point (ICP) algorithm applied to a point cloud of the sensor's deformed surface, improving alignment between the contact geometry and object model. Caddeo et al. [45] developed a technique that employs multiple vision-based tactile sensors to estimate an object's 6D pose while in-hand. Their method begins with a CNN-based selection of candidates on the object's surface poses based on sensor-image compatibility. These candidates are then optimized using gradient descent and ranked through a physically informed scoring system that penalizes poses resulting in sensor-object collisions. The final estimate corresponds to the pose with the lowest physical inconsistency score.

For our setup, we rely on the event-based optical tactile sensor introduced by Funk et al. [9]. Building upon the GelSight Mini [41], this sensor replaces the internal camera with an event-based one, which increases the frequency from 25Hz up to 1000 Hz.

Since our control requires a high-frequency correction of the pole's state, we also need a high-frequency pole state estimation, which is enabled by this sensor. The experiments performed by Funk et al. showed that the sensor can precisely reconstruct applied forces, based on dots tracked in the gel. Furthermore, the sensor was trained to detect as well as prevent slips when holding an object and gradually losing the grip. We use the same sensor as in [9], because, due to its high frequency of publishing, it is well suited for the high-frequency control needed for the pole balancing task. Different from Funk et al., we work with gels both with and without dots. Regarding the method, we try to regress a state of a dynamic system instead of learning torques or detecting slips in rather static environments, as in their experiments. Furthermore, we use different neural network architectures and datasets.

3. Foundations

After reviewing previous works, this chapter introduces key concepts essential for understanding our approaches and experiments. This includes the mathematical foundations and important hard- and software descriptions like ROS, OptiTrack, Evetac, and finally deep learning.

3.1. Mathematical Foundations

The mathematical foundations we introduce include spaces and forward kinematics linearization, different linear feedback controllers, and filtering techniques.

3.1.1. Forward Kinematics

Defining an objective, like holding a position, can be done in two ways in robotics. First, we could specify the respective goal in (Cartesian) coordinates, e.g., when following a trajectory. However, you can also specify the same goal in the joint space by defining the goal as the parameters of the robot's joints, which in most cases are revolute joint angles in radians or prismatic shift values in meters, leading to the same Cartesian coordinates. Forward kinematics describes the process of determining the position and orientation of a robot's end-effector given its joint angles and link parameters. To our setup this is crucial because we need to know the orientation and position of the end-effector in order to estimate the local pole angle during balancing, when using the setup in real life. Mathematically, forward kinematics is expressed as a function:

$$\mathbf{p} = f(\psi)$$

where $\mathbf{p}$ is the end-effector orientation and position and ψ is the joint configuration vector of the robot. The transformation from joint space to Cartesian space is computed using a series of homogeneous transformation matrices that describe the relative position and orientation of each link with respect to the previous one. The position of the end-effector is obtained by multiplying the transformation matrices, commonly defined through the Denavit-Hartenberg convention, of each joint:

$$\mathbf{T} = \mathbf{T_1} \cdot \mathbf{T_2} \cdot \cdots \cdot \mathbf{T_{n-1}} \cdot \mathbf{T_n}$$

where each $\mathbf{T_i}$ is a homogeneous transformation matrix encoding the rotation and translation between consecutive links. A figure of such a Kinematic chain and the corresponding transformation matrices can be seen in Figure 3.1.

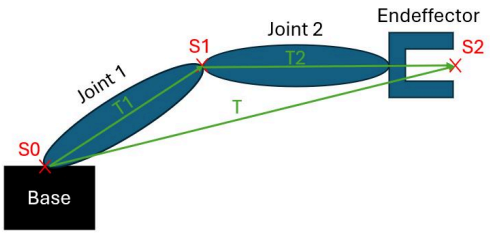


Figure 3.1.: Kinematic chain example for a simple 2-DoF robot

3.1.2. Linearization of Differential Equations

A first-order linear ordinary differential equation has the form

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state vector and $\mathbf{u} \in \mathbb{R}^m$ is an external control signal input. Changing the values for $\mathbf{u}$ allows us to influence and control the behavior of the system, referred to as controlling. $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times m}$ are the matrices mapping the state and control to the individual parts of $\dot{\mathbf{x}}$. We can linearize any differential equation around any $\mathbf{x}$, forming a simple approximation of the true equation that is locally accurate as long as the original differential equation is sufficiently smooth. For example, for the inverted pendulum, we can linearize around the equilibrium point, i.e,. a stable point, where the behavior is approximately linear, to approximate the differential equation.

3.1.3. Linear Feedback Controller

To actively balance a pole, we need a controller to compute the $\mathbf{u}$ we want to apply. There are several types of controllers, e.g. simple P, PD, or PID controllers, or optimal controllers such as LQR. These controllers are called feedback-driven because they control the system based on its state observation, which is also the way humans operate when controlling a system like balancing a pole, as mentioned in [46]. We present the equations of the PD and LQR controllers (taken from [47, 48]) for computing $\mathbf{u}$ as both are used in our setup.

PD-Controller

The PD controller has the following control law:

$$\mathbf{u} = \mathbf{K_P}(\mathbf{x_t} - \mathbf{x}) + \mathbf{K_D}(\dot{\mathbf{x_t}} - \dot{\mathbf{x}})$$

Where $\mathbf{x}, \mathbf{x_t} \in \mathbb{R}^n$ are the state and the target state of the system, and $\mathbf{K_P}, \mathbf{K_D} \in \mathbb{R}^{n \times n}$ are diagonal matrices scaling the respective error terms on position and velocity. This simple control law is often used to hold position or follow trajectories with a system. The sensitivity of the controller to errors can be defined by choosing the values of $\mathbf{K_P}$ and $\mathbf{K_D}$, commonly determined by an analysis of the Eigenvalues of the system matrix or by manual trial and error.

LQR

Among the approaches surveyed in [49], LQR stands out for its performance and simplicity, though it is sensitive to noise and model inaccuracies. An LQR gives the optimal solution to a system whose control costs are defined as a quadratic function in $\mathbb{R}$ and the dynamics are a set of linear differential equations, as in our setup. The cost function can be defined as

$$J = \int_{t_s}^{t_f} (\mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{u}^T \mathbf{R} \mathbf{u}) dt$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state, $\mathbf{u} \in \mathbb{R}^m$ and the control signal, $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is the state cost matrix, $\mathbf{R} \in \mathbb{R}^{m \times m}$ the control cost matrix and t_s, t_f mark the start and end of our time

horizon. The (optimal) control $\mathbf{u}$ given these conditions, i.e. the one that minimizes this cost function, is defined by

$$\mathbf{u} = -\mathbf{K}\mathbf{x}$$
$$\mathbf{K} = \mathbf{R}^{-1}(\mathbf{B}^T\mathbf{P})$$

where $\mathbf{K} \in \mathbb{R}^{m \times n}$ is the gain matrix and $\mathbf{P} \in \mathbb{R}^{n \times n}$ is computed by solving the continuous time Ricatti differential equation

$$\dot{\mathbf{P}} = -(\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} - \mathbf{P}\mathbf{B})\mathbf{R}^{-1}(\mathbf{B}^T\mathbf{P}) + \mathbf{Q})$$

given $\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R}$. For large systems, the Ricatti equation is typically solved numerically as explained in [50], but for our smaller system, we solve it analytically. For given system dynamics $\mathbf{A}$ and $\mathbf{B}$, we can tune the LQR by changing the values for $\mathbf{Q}$ and $\mathbf{R}$. As described by the cost function, higher values in $\mathbf{Q}$ discourage the respective state error more drastically. Similarly, $\mathbf{R}$ discourages our external actions on the system.

3.1.4. Luenberger-Observer

State estimation and tracking given an (in)complete, noisy observation of the true state has many practical applications. For example, we can employ a Kalman-Filter [51] to deal with noisy measurements of our state under the assumption of a linear state transition model. Another related approach is the Luenberger-Observer [52]. The Luenberger-Observer relies on a prediction of the current state given the previous state to estimate a complete state at all times. We model the dynamic equations of the robot and pole setup as follows:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} = \boldsymbol{\tau} + f(\dot{\mathbf{q}}) - C(\mathbf{q}, \dot{\mathbf{q}}) - G(\mathbf{q}) + \mathbf{d},$$
$$\ddot{\mathbf{q}} = M^{-1}(\boldsymbol{\tau} + f(\dot{\mathbf{q}}) - C(\mathbf{q}, \dot{\mathbf{q}}) - G(\mathbf{q}) + \mathbf{d}).$$

Given the computed acceleration $\ddot{\mathbf{q}}$, the state update of the Luenberger-Observer follows a simple integration scheme:

$$\dot{\mathbf{q}} \leftarrow \dot{\mathbf{q}} + \ddot{\mathbf{q}} \cdot \Delta t,$$
$$\mathbf{q} \leftarrow \mathbf{q} + \dot{\mathbf{q}} \cdot \Delta t.$$

where Δt is the time-step difference between now and the previous step, $\boldsymbol{\tau}$ is the applied torque, $C(\mathbf{q}, \dot{\mathbf{q}})$ is the Coriolis force, $\mathbf{d}$ is a disturbance model, $f(\dot{\mathbf{q}})$ is modeling the friction, $\mathbf{M}$ is the mass matrix and $\mathbf{q}$ is the state vector.

After the model-based prediction step, we compute the difference $\boldsymbol{\epsilon} \in \mathbb{R}^n$ between the computed result and the observed data. We adapt our state based on $\boldsymbol{\epsilon}$ multiplied with the respective gains for position $\mathbf{L_P}$, velocity $\mathbf{L_D}$ and integration $\mathbf{L_I}$, which are diagonal matrices $\in \mathbb{R}^{n \times n}$ defining the sensitivities to the observations. This helps to smooth the state observation while also allowing it to react to rapid changes.

$$\begin{aligned}\mathbf{q} &= \mathbf{q} + \mathbf{L_P} \cdot \boldsymbol{\epsilon} \\ \dot{\mathbf{q}} &= \dot{\mathbf{q}} + \mathbf{L_D} \cdot \boldsymbol{\epsilon} \\ \mathbf{d} &= \mathbf{d} + \mathbf{L_I} \cdot \boldsymbol{\epsilon}\end{aligned}$$

However, if no observation is present, we solely rely on the computed state. Being capable of working with incomplete state observations is important because for the physical robot implementation, we do not always have the complete state available.

3.1.5. 1 Euro Filter

The LQR provides optimal control within constraints, but the resulting torques can be jittery due to the noise in the observations, which may damage the robot. To address this, we smooth the torques using a 1-Euro Filter [53]. This filter is a type of exponential low-pass filter with an adaptive alpha, allowing it to respond to rapid signal changes. It filters both the signal and its derivative with a dynamically adjusted cutoff frequency. The following equations define the filter:

$$\begin{aligned}\alpha(f_c) &= \frac{1}{1 + \frac{f_s}{2\pi f_c}} \\ \dot{x}_t &= (x_t - x_{t-1}) f_s \\ \dot{\tilde{x}}_t &= \alpha(f_d)\dot{x}_t + (1 - \alpha(f_d))\dot{\tilde{x}}_{t-1} \\ f_c &= f_{min} + \beta|\dot{\tilde{x}}_t| \\ \tilde{x}_t &= \alpha(f_c)x_t + (1 - \alpha(f_c))\tilde{x}_{t-1}\end{aligned}$$

where:

- f_s, f_c, f_d, f_{min} are the sampling frequency, the adaptive cutoff frequency for the main signal, the cutoff frequency used to filter the derivative, and the minimum cutoff frequency.
- α, β are the smoothing factor and the parameter controlling the sensitivity to fast signal changes.

- $x_t, \tilde{x}_t$ are the raw/filtered input signal at time t (the torque).
- $\dot{x}_t, \dot{\tilde{x}}_t$: Raw/filtered numerical derivative of the signal.

3.2. Hard- and Software

3.2.1. ROS, URDF, Pinocchio

To control the real WAM, we rely on the Robot Operating System (ROS) [54], which serves as a middleware between our software and the robot's hardware. ROS provides a flexible framework for developing, managing, and integrating robotic systems by enabling modular communication between different software components. ROS operates through a node-based architecture, where individual nodes, representing separate programs, communicate with each other via message passing. This allows for distributed processing, making it easier to handle tasks such as sensor integration, motion planning, and real-time control.

A Unified Robot Description File (URDF) is an XML-based format used to define a robot's structure, including its links, joints, and physical properties. It provides a standardized way to represent kinematic and dynamic properties for simulation, visualization, and control in robotics frameworks such as ROS. Each URDF consists of hierarchical definitions of links (rigid bodies) and joints (connections between links), forming a tree structure. Joints can be of different types, such as revolute, prismatic, or fixed, defining how the links move relative to each other. The file also contains optional elements like collision, inertia, and visual properties to enhance physical accuracy in simulation environments.

Pinocchio is a robotics library [55, 56], which instantiates the state-of-the-art Rigid Body Algorithms for poly-articulated systems based on revisited Roy Featherstone's algorithms. Pinocchio provides the analytical derivatives of the main Rigid-Body Algorithms, such as the Recursive Newton-Euler Algorithm or the Articulated-Body Algorithm (taken from [56]). Pinocchio interprets the URDFs and XML, declaring the rigid body tree of the WAM, and computes the necessary terms for control.

3.2.2. WAM

According to Barrett: "The WAM Arm is a highly dexterous, naturally back-drivable manipulator. The only arm sold in the world with direct-drive capability supported by

Transparent Dynamics between the motors and joints, so its control of contact forces is robust, independent of mechanical force or torque sensors. It is built to outperform conventional robots with unmatched human-like grace and dexterity." [1]. A picture of the WAM with both 4 and 7 Degrees of Freedom (DoF) and the corresponding joint locations can be seen in Figure 3.2. We use the simpler 4 DoF variant of the WAM as it lowers the state space and provides sufficient control options.

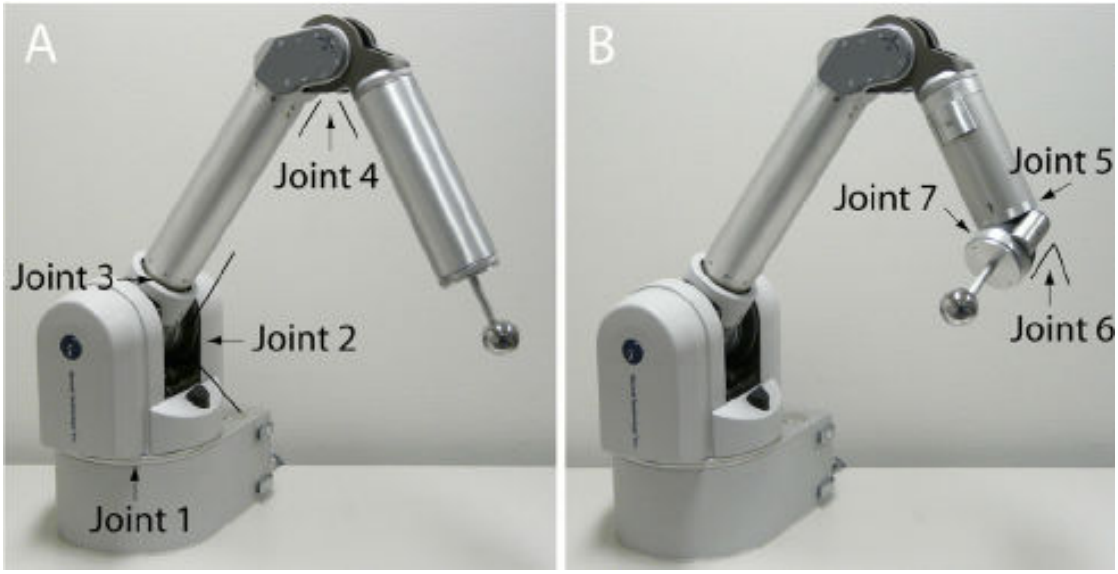


Figure 3.2.: WAM with 4 and 7 degrees of freedom

The WAM uses an advanced tendon-driven system for joint actuation. This unique design enables high flexibility and back drivability, meaning the arm can be moved manually by an external force, making it ideal for interaction with the environment and human collaboration. The tendons, which are integrated within the arm's structure, transmit forces from the motors to the joints, enabling precise and smooth movements.

Each joint of the WAM arm is controlled using direct-drive motors that are connected to the joints via these tendons. The direct-drive capability eliminates the need for complex gear systems, reducing backlash and friction while improving overall control of the joint motions. This results in highly responsive movements and more precise control, particularly important for delicate tasks such as pole balancing. Motor control in the WAM is achieved via advanced servo controllers, which regulate the speed, torque, and position of the

[1] cited from Barrett-Website: https://barrett.com/wam-arm

motors based on commands from the control system. These controllers provide real-time feedback on the motor's position and speed, allowing for accurate tracking and adjustments of joint angles. Combined with ROS, this feedback loop ensures that the WAM arm can execute complex motions with fine precision, whether in a simulation or while performing physical tasks.

3.2.3. OptiTrack

We need a continuous, fast, and accurate pole state estimate during balancing to reliably and timely detect falls so that we can balance them out. For this task, we used OptiTrack [28]. OptiTrack Motion capture is a commercially available motion capture system consisting of software and cameras. Multiple cameras are placed around a scene, continuously capturing markers present. A marker is any retroreflective surface, in our setting, UV-reflecting duct-tape strips. The cameras are synchronized and estimate the position of these markers together. A marker is tracked if observed by more than two cameras and the pose is estimated and visualized in Motive, the respective software responsible for fusing the different camera observations. The newest OptiTrack cameras are capable of sub-millimeter accuracy with an error of $0.0001\,\mathrm{m}$ at a rate of up to 1000 frames per second for marker tracking. An example of such a camera can be seen in Figure 3.3

Figure 3.3.: OptiTrack camera, from OptiTrack Website

3.2.4. Evetac

The vision-based tactile sensor we used, called Evetac [9], consists of a DVXplorer Mini event-based camera placed under the silicon gel of a GelSight Mini. The components are

visualized in an exploded view in Figure 3.4 taken from [9]. The camera is housed in a

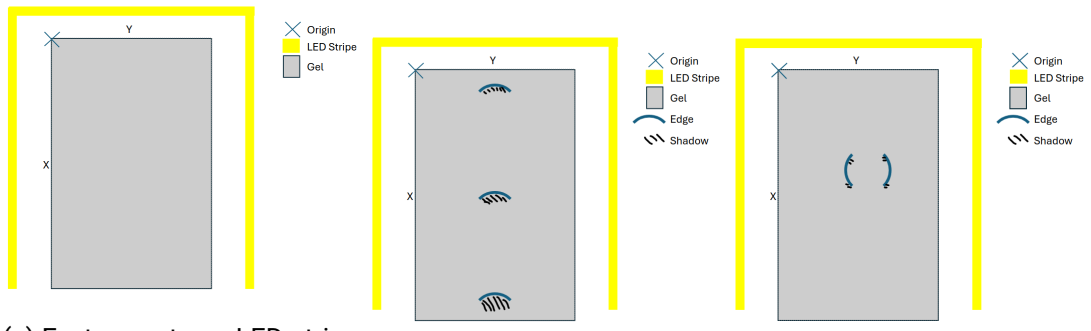
Figure 3.4.: Exploded view of the proposed Evetac sensor. From left to right: A) DVXplorer Mini, event-based camera, B) 3-D printed camera housing, C) LED-stripe for illumination from the inside, and D) GelSight Mini gel.

3-D-printed enclosure, with the gel sealing the top and the camera filming from below. Inside the housing, an LED-strip illuminates the otherwise dark interior. This LED-strip runs along three of the four edges of the gel: the two long edges parallel to the x-axis and the short edge parallel to the y-axis, starting at the origin.

The gel deforms when objects exert force on it, such as the pole in our case. The camera captures an event whenever it detects a change in brightness at any pixel, operating at a resolution of 640×480 pixels. Each event contains the corresponding pixel's location, the polarity of the brightness change (positive or negative), and a timestamp. When the gel remains flat, no significant shadows form, and the camera primarily detects noise. However, when the gel deforms, two key phenomena occur:

- The pixels at the edges of the deformation facing the light receive increased illumination from the nearby LED-strip. This happens because, in the gel's flat state, light rays strike at a shallow angle. Deformation alters this reflection angle, increasing the light intensity perceived by the affected pixels and generating positive events. These positive events strongly correlate with the edges of the deforming object.
- Conversely, areas behind the newly formed edges receive less illumination because the deformation blocks light that previously reached them. This results in negative events in regions dependent on the edge's location, orientation, and size.

These effects of an edge's location and orientation on the perceived events by the sensor are qualitatively visualized in Figure 3.5. The general setup of the gel and LED-strip is depicted in Figure 3.5a. In Figure 3.5b, we show how edge position affects shadow length. The lower gel region is less illuminated due to the lack of a bottom light source. As a result, edges deforming the lower gel cast longer shadows, since the x-axis-aligned LED-stripe provides less effective lighting in that area. Edge orientation also impacts



(a) Evetac setup, LED-stripe around gel (b) Impact of position of edges (c) Effect of rotation of edges

Figure 3.5.: Evetac setup analysis

shadow formation. Edges parallel to the y-axis obscure a larger area than edges parallel to the x-axis. This effect is shown in Figure 3.5b and 3.5c. Note that shadow sizes are not to scale. Additionally, the size of the edge affects the shadow cast. Larger edges result in larger shadowed areas, following a simple proportionality principle. It is important to note that events are only triggered when intensity changes occur. Once an edge deforms the gel and generates an initial set of events, it becomes practically invisible to the camera until it changes again. Therefore, the size and position of the edge mainly determine the potential number of events it can trigger.

Furthermore, two additional factors influence event generation in our setup. These factors are:

- The rate of change in the deformed area: Even a large edge causing a big deformation may generate very few events if it moves slowly enough over the gel. If the motion is too gradual, intensity changes may be too subtle for the camera to detect. This situation can occur when the pole tilts very slowly or remains at an almost constant tilt angle, resulting in minimal edge motion and deformation.
- The depth of the gel deformation: If the object pressing on the gel is too lightweight, it may not deform the gel sufficiently to create a noticeable edge or shadow. In such cases, the sensor's sensitivity is limited by the physical properties of the silicon gel. To ensure reliable detection, the pole must have sufficient weight to produce a meaningful deformation.

Finally, by physics and as you can see in the previous figure 3.5, shadows are always cast

to the side of the rectangle, where no light exists. Furthermore, to point out, events also occur when a previous deformation is redone, but in converse polarity, due to an increase in light in the previous shadow regions and a decrease in intensity due to a shallower angle of the light to the gel where the edge was. By considering these factors, we understand the data produced by the sensor and can optimize event detection and interpretation.

3.3. Deep learning

If we want to use the tactile sensor for control, we need to be able to interpret the events observed by the tactile camera as the state of the pole. In order to learn a mapping from the tactile sensors observations to the actual state of the pole we use neural networks, trained in a supervised fashion. A supervised learning problem consists of a set of values $\boldsymbol{\chi}$ and the respective desired set of outputs $\boldsymbol{\Omega}$. We want to learn a function $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{y}$ where f is the neural network conditioned on its weights $\mathbf{w}$, $\mathbf{x}$ is an input taken from $\boldsymbol{\chi}$, and $\mathbf{y}$ is the respective output taken from $\boldsymbol{\Omega}$. Given that, in theory, a single-layer perceptron with a non-linear activation function is a universal function approximator [57], we hope to be able to learn a function from our data, so that we can accurately map the tactile sensors' observations to the necessary pole state for control. We trained different architectures that are briefly described in the following, together with the data that is fed into these networks.

3.3.1. MLP

A Multilayer perceptron (MLP) consists of multiple sets of nodes, and each set of nodes forms a layer (perceptron). Nodes of adjacent layers are connected in a one-to-one fashion. This means each node in layer i is connected to all nodes of layer i+1 via an outgoing edge and all nodes of layer i-1 via an incoming edge. A visualization of this architecture can be seen in Figure 3.6. A node takes in all the incoming edges, computes a weighted sum based on the transmitted values and the internal weight states, adds a bias and outputs this value to all its outgoing edges. In between the layers there a nonlinear activation functions, e.g. $ReLU$, which enable the network to learn non linear functions. Examples for such non-linear activation functions can be seen in 3.7. The MLP takes in a vector of values, corresponding to the size of the input layer, which can be e.g., dot locations as in [9] or a flattened image. The values get passed through the network, and the goal is to predict the desired values at the final layer's output.

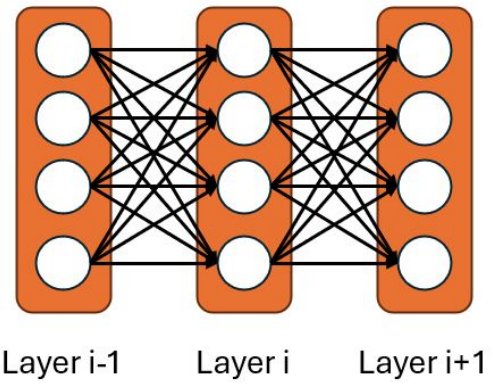


Figure 3.6.: An Example figure explaining the architecture of an MLP and the connections to the previous and following layer.

3.3.2. CNN

A more natural way to process images than flattening them and feeding them to an MLP is to use a convolutional neural network (CNN). A CNN uses convolutions, i.e., matrix operations on image chunks, to compute feature maps. The net stacks multiple convolutions intertwined with Max pools and non-linear activation functions in multiple layers. The Max pooling is used to reduce the size of the images, allowing the CNN to focus on different scales of the image to detect local and global significant things in the image. The CNN outputs a multidimensional tensor in the shape of the final down-sampled image and the extracted feature maps.

In Figure 3.8, we visualize examples of the different components of a CNN: In Figure 3.8a to each image pixel we apply a convolution, which in this case is a 3×3 matrix, consisting of 9 values, multiplied with the pixel values of the pixel and its neighborhood. The resulting value is stored, so that in the end, a new image is created when all pixels have been convolved. Figure 3.8b visualizes the different convolution filters the CNN uses for a single input, thus creating multiple convolved images, called feature maps, from a single input image. Multiple of these feature maps are stacked in so-called channels. To reduce the dimensionality of the images during processing, after convolution, a Max pooling operation is commonly performed. As depicted in Figure 3.8b, the operation is applied to a neighborhood of pixels, extracting the maximum pixel value, discarding the rest. After the convolutions, the final feature maps are aggregated by e.g. flattening or

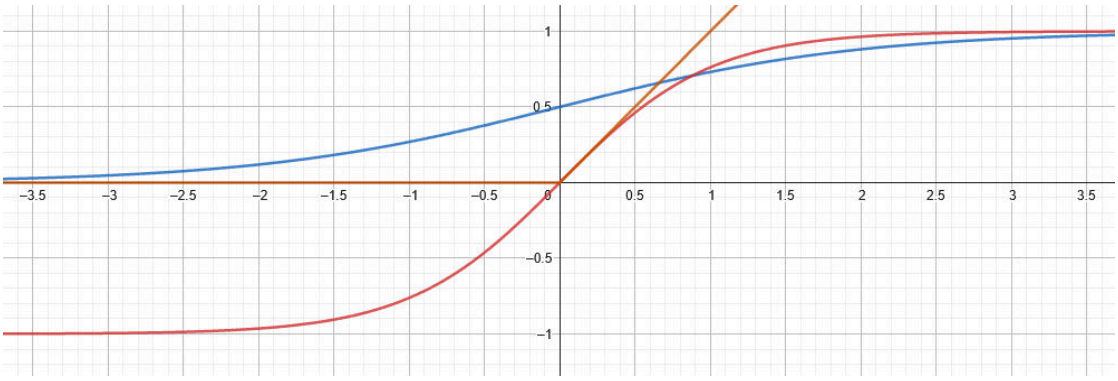


Figure 3.7.: Plots of example activation functions: $ReLU$, $Sigmoid$, $Tanh$

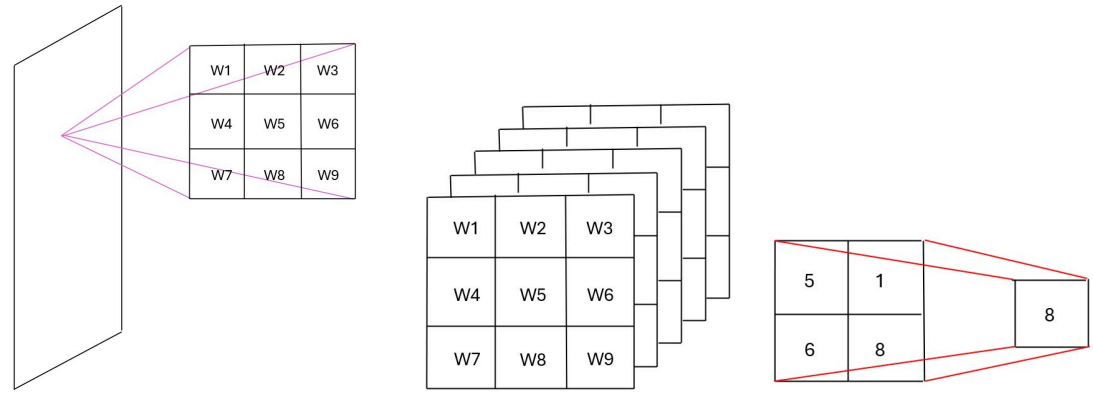
global pooling to form a single feature vector which then is typically processed by a MLP to output a desired value.

3.3.3. LSTM

A long short term memory [58] (LSTM) is a recurrent neural network and is a more complex network designed for sequence analysis like text completion or stock market prediction. It takes as input a tensor, resembling a sequence, i.e., multiple sequential points, and processes them internally one element of the sequence at a time. The structure of an LSTM is visualized in Figure 3.9. It consists of 3 gates, called the forget, input, and output gates. The forget gate is a linear layer with a sigmoid activation function that determines what part of the previous cell-state is unimportant. The input gate consists of 2 layers one with a tanh and one with a sigmoid activation, which determines what new information is to store in the cell-state. The output gate computes what part of the cell-state the LSTM outputs and also what the new hidden-state is, based on the new cell-state and the input. If the end of the sequence is not yet reached, the new hidden state and cell state are used to process the next entry.

3.3.4. PointNet

PointNet [59] is a popular method to convert a set of points, a pointcloud, into a single global feature describing the pointcloud. The net architecture consists of an MLP combined



(a) Convolution (b) Channel stacking (c) Max pooling

Figure 3.8.: Overview of CNN operations: (a) convolution on a single pixel, (b) stacked feature maps, and (c) max pooling operation

with global pooling at the end to aggregate the predicted tensor into a simple vector. The structure can be seen in Figure 3.10. Instead of creating an image from the events and using a CNN, we can take the events as a pointcloud, based on their pixel location and use PointNet to generate a vector. The high-dimensional global vector is processed by other linear layers to a size of 256. In [59], they use this vector to classify the shapes, from which they sample the pointcloud or segment the object into individual parts.

3.3.5. Supervised Learning

There are several ways to train a neural network. In supervised learning, the model continuously makes predictions, computes the loss against a target output, and updates its parameters accordingly. The weight update follows through back-propagation [60], where we compute the derivative of the loss with respect to each weight using the chain rule:

$$\Delta \mathbf{w_i} = \frac{\partial}{\partial \mathbf{w_i}} \text{Loss}(\mathbf{y}', \mathbf{y})$$
$$\mathbf{w_{i+1}} = \mathbf{w_i} - \alpha \cdot \Delta \mathbf{w_i}$$

where $\mathbf{y}'$ is the model's prediction, $\mathbf{y}$ is the true value, and α is the learning rate. This optimization process, known as gradient descent, aims to minimize the loss function by

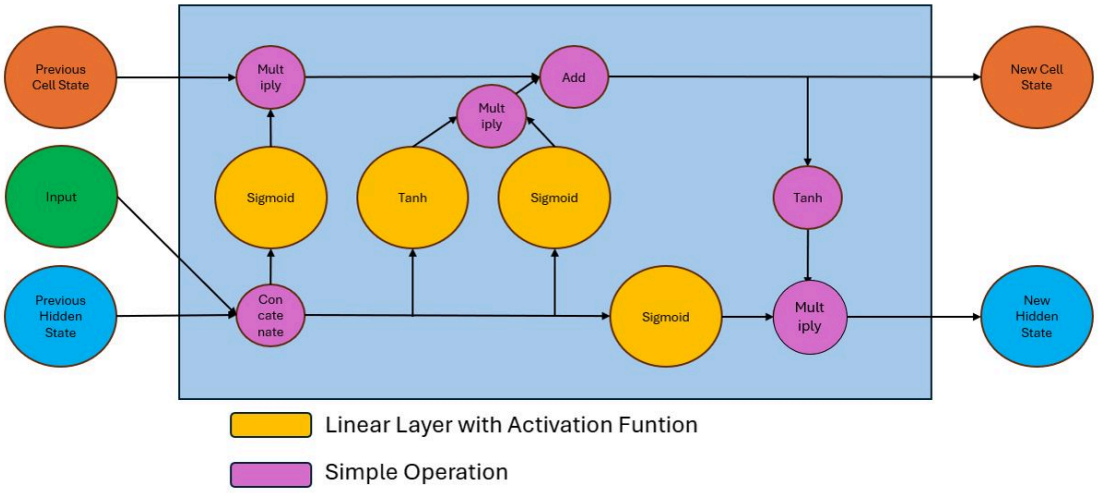


Figure 3.9.: Inner mechanisms of a long short term memory (LSTM)

adjusting the model's weights. While convergence to a global optimum is not guaranteed, the algorithm ensures convergence to a local optimum if the learning rate α is gradually reduced to zero during training.

Common loss functions include the Mean Squared Error (MSE) for regression tasks and the Cross Entropy (CE) Loss for classification tasks:

$$\text{MSE}(\mathbf{y}', \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}'_i - \mathbf{y}_i)^2$$

$$\text{CE}(\mathbf{y}', y) = -\log\left(\frac{e^{\mathbf{y}'_{i,y}}}{\sum_{c=0}^{C} e^{\mathbf{y}'_{i,c}}}\right)$$

For CE $\mathbf{y} \in \{0, 1, 2, 3, ..., C\}$ is the true class and $\mathbf{y}' \in \mathbb{R}^{C+1}$ are the predicted logits of each class. For MSE $\mathbf{y}, \mathbf{y}' \in \mathbb{R}^n$ are the true and predicted continuous vectors of the regression task. These functions quantify the error between predictions and targets, guiding the weight updates during training.

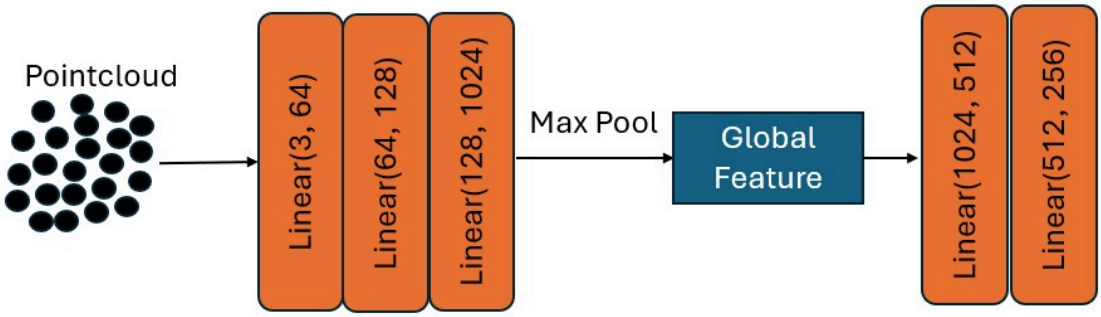


Figure 3.10.: Structure of PointNet as described in [59], orange layers are paired with a ReLU

3.3.6. Regularization

During training, over-fitting is happening when the model we train learns to reduce the training loss but fails to reduce the evaluation loss similarly, i.e., the model does not generalize but specializes on the training data. Over-fitting on the training data is undesired as we want a general model. A neural network can over-fit due to a multitude of reasons, e.g. due to a too complex model structure or when there is a lack of training data. There are several methods to prevent or counteract over-fitting, such as weight decay, dropout, and layer norms. Weight decay [61] amends the loss of the training procedure by adding the sum of the squared weights of the model times a regularization factor to the loss, penalizing large weights, thus limiting over-fitting. Dropout [62] is introducing a layer to the network that simply drops some of its inputs by setting them to 0, based on a probability going from 0 to 1, given as a parameter. This leads to missing features during learning, forcing the network to focus on robust features during learning. Layer Norms [63] are a way to stabilize the training procedure of a neural network by normalizing the input, which can help prevent over-fitting.

4. Method

To stabilize the pole, we need to convert the tactile observations into the pole state. Since there is no explicit model for this, we aim to implement a data-based approach. This means we want to learn a mapping from the tactile sensors' output to the actual pole angles of our system using neural networks as presented in the previous section 3.3. If we can interpret the tactile sensor data accurately, it enables us to control the system on the inferred state of the pole combined with the robot state to balance the system as outlined in section 3.1.3. For the controller, we use an LQR due to its optimality given an accurate model and simplicity in terms of computational effort.

Gathering a dataset for learning requires us to be able to reliably balance the pole on top of the tactile sensor at the end-effector of the WAM based on the perceived pole angle and the robot state. Balancing the pole in this free-moving setup relies on precise control by the LQR, as the pole can easily slip off the sensor if the controller is too soft or too hard, resulting in too little or too much motion of the arm. To find these suitable parameter values for the LQR and to get to know the software and hardware of the WAM, we start by designing a simulation in which we can create and test arbitrary control parameters and study their effect on the balancing task. After our simulation trials, excluding the tactile sensor, we adapt our setup to the real WAM and the tactile sensor, amending the control parameter and pipeline accordingly. Stabilizing the pole on the tactile sensor enables us to capture datasets consisting of trajectories and the occurring tactile events during these trajectories.

With these captured datasets, we learn models that predict the pole state based on the tactile events occurring at each time-step. We go in detail over the dataset creation procedure and the preprocessing performed on each data-point for learning. Furthermore, we highlight the different types of networks and how to use the learned neural networks for tactile-based state estimation of the pole in the real control pipeline.

4.1. Pole Balancing Joint Space Description

First, we state the mathematical problem definition for the pole balancing that we used. The pole balancing task can be described in many ways. For example, we can model the pole and the robot end-effector separately, controlling their combined setup in Cartesian coordinates. This would effectively emulate the cart-pole setup mentioned in e.g. [10]. However, we can also alternatively declare the problem in joint space, treating the pole as two new joints of the robot, which cannot be controlled (under-actuated system). We focus on the joint space description of the task, as the control for the robot is more naturally defined there. The inverted pendulum's state can be defined as the pole's tilt angle $\boldsymbol{\theta}$ for the 2 axes in our setup, as well as the velocity $\dot{\boldsymbol{\theta}}$. To describe the state of the robot, we need to know the position of the joint angles $\boldsymbol{\psi}$, as well as their velocities $\dot{\boldsymbol{\psi}}$. Together $\boldsymbol{\theta}$ and $\boldsymbol{\psi}$ and their derivatives give us a complete description of the system state and velocity.

The combined state of the system $\mathbf{q}$ and the target configuration $\mathbf{q_t}$ is then defined as:

$$\mathbf{q} = (\boldsymbol{\psi}, \boldsymbol{\theta}, \dot{\boldsymbol{\psi}}, \dot{\boldsymbol{\theta}})$$
$$\mathbf{q_t} = (\boldsymbol{\psi_t}, \boldsymbol{\theta_t}, \dot{\boldsymbol{\psi}}_{\boldsymbol{t}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{t}})$$

Furthermore, the difference $\mathbf{q}'$ between state $\mathbf{q}$ and target configuration $\mathbf{q_t}$ is defined as:

$$\begin{aligned}\mathbf{q}' &= (\boldsymbol{\psi} - \boldsymbol{\psi_t}, \boldsymbol{\theta} - \boldsymbol{\theta_t}, \dot{\boldsymbol{\psi}} - \dot{\boldsymbol{\psi}}_{\boldsymbol{t}}, \dot{\boldsymbol{\theta}} - \dot{\boldsymbol{\theta}}_{\boldsymbol{t}}) \\ &= (\boldsymbol{\psi} - \boldsymbol{\psi_t}, \boldsymbol{\theta} - \boldsymbol{\theta_t}, \dot{\boldsymbol{\psi}}, \dot{\boldsymbol{\theta}})\end{aligned}$$

Since we want the robot and the pendulum to have zero velocity, i.e. $\dot{\boldsymbol{\theta}}_{\boldsymbol{t}} = \dot{\boldsymbol{\psi}}_{\boldsymbol{t}} = 0$, we can shorten the velocity terms. We approximate the differential equation of the inverted pendulum task with a linearization around $\mathbf{q_t}$ with the known system properties (mass, inertia, and volume) of the WAM and the pole as explained in section 3.1.2. The linearization gives us a simple and complete equation with which we can control the pole by computing the control signal $\mathbf{u}$ as outlined in section 3.1.3. For this actual control, we replace $\mathbf{x}$ with $\mathbf{q}'$ to linearize around the equilibrium point, where the task is approximately linear. However, since it is only locally accurate, we need to ensure that our robot and pole remain close to the equilibrium point during control.

4.2. LQR Parameter

For the control algorithm we choose an LQR, due to its optimal control properties given accurate model and state perception. In our setup, the 4-DoF WAM and the 2-DoF pole, resulting in a 6-DoF system and a $\mathbf{q_t} \in \mathbb{R}^{12}$. Therefore, $\mathbf{Q}$ is a 12×12 diagonal matrix with different weights for each non-zero entry. The entries get multiplied with the state error $\mathbf{q}'$, meaning the higher the individual entry, the more we punish deviations from $\mathbf{q_t}$.

We can split the 12 diagonal entries of $\mathbf{Q}$ in four parts relevant to different kinds of $\mathbf{q}$: The entries 1 to 4 target the robot state ψ, the entries 5 to 6 work on the poles state $\boldsymbol{\theta}$, 7 to 10 corresponds to the robot joint velocities $\dot{\psi}$ and the entries 11 to 12 weight the poles velocity $\dot{\boldsymbol{\theta}}$. We assume a uniform value for each of the four groups' entries, as e.g., all pole tilts should be treated as equally undesired. Therefore, for $\mathbf{Q}$ we can tune 4 values to weight the different parts of $\mathbf{q}$ differently. We call these values d_1, d_2, d_3, d_4.

$\mathbf{R}$ is in our setup a 4×4 diagonal matrix with different weights for each non-zero entry. For $\mathbf{R}$, we can weight the actions applied to each of the 4 joints or motors, meaning we would get another 4 values in total to tune. However, most commonly, for an LQR, the $\mathbf{R}$ values are equal, to signal all actions should be treated as equally undesired, which is why we end up with one parameter to tune. Since $\mathbf{Q}$ and $\mathbf{R}$ both influence the cost function, we choose a single value for the entries of $\mathbf{R}$ and evaluate the $\mathbf{Q}$ values to find suitable parameters for them in section 5.1.2.

4.3. MuJoCo Simulation Control

For the simulation environment, we rely on MuJoCo (Multi-Joint dynamics with Contact), a popular step-based physics engine used for simulating robots and physical systems. An image of the simulation can be seen in Figure 4.1.

A simulation is especially useful to us because we can query all data perfectly from it and safely test different controller values and pole setups without damaging any hardware. This enables us to evaluate different LQR parameters, robot positions, and pole configurations in an exhaustive, fast, and safe manner, finding well-performing configurations. For rolling out the LQR controller, the robot starts in the target configuration $\mathbf{q_t}$ by setting the respective joint state and motor state if we model the tendons.

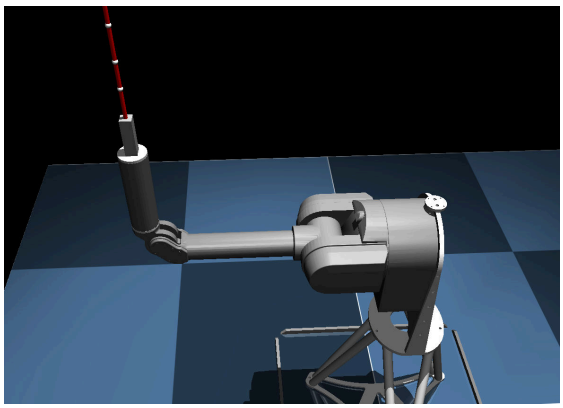


Figure 4.1.: MuJoCo simulation example image, with the Evetac emulating end-effector

With Pinocchio, we compute the gravity compensation term $\mathbf{gc}$ as well as the linearization around $\mathbf{q_t}$ by deriving the acceleration given by the articulated body algorithm with respect to the system state $\mathbf{q}$ and torques τ. The resulting vectors fill the rows of $\mathbf{A}$ and $\mathbf{B}$. Furthermore, we initialize the $\mathbf{Q}$ and $\mathbf{R}$ matrices defining state error punishment and action punishment. Given $\mathbf{A}, \mathbf{B}, \mathbf{Q}$ and $\mathbf{R}$, we compute the LQR optimal controller gain matrix $\mathbf{K}$ by solving the Ricatti equations approximately as explained in 3.1.3. With our controller gain matrix available, we start the simulation control loop, which can be seen in 4.2.

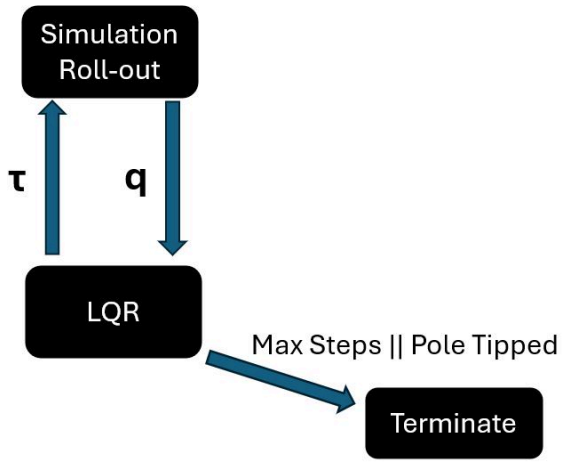


Figure 4.2.: MuJoCo control pipeline

The control loop consists of 2 major steps: First, we read the complete state $\mathbf{q}$ directly from the MuJoCo data structure in each iteration and compute the difference $\mathbf{q}'$ to the desired state $\mathbf{q_t}$. We log the state and compute the LQR defined control torque $\boldsymbol{\tau}_{joint} = \mathbf{u} = -\mathbf{K}\mathbf{q}' + \mathbf{gc}$. Noticeably, the control term is not the same as listed in section 3.1.3 as we add the gravity compensation term $\mathbf{gc}$ to $\mathbf{u}$. Adding $\mathbf{gc}$ is crucial because for a perfect configuration, meaning $\mathbf{q} = \mathbf{q_t}$ we would have $\boldsymbol{\tau}_{joint} = \mathbf{0}$, which would lead to a fall of the robotic arm, since it generally cannot hold any position $\mathbf{q}$ due to gravity accelerating the system. To prevent this acceleration, we need the gravity compensation to hold the position. The resulting torque $\boldsymbol{\tau}_{joint}$ is either converted and applied to the motors or directly applied to the joints (which is only possible in the simulated WAM), depending on whether we modeled the tendons and motors of the WAM.

For converting the motor torques to joint torques, we use the tendon Jacobian. The tendon Jacoby matrix relates the joint positions to the motor positions and is defined for the WAM as:

$$\mathbf{J} = \begin{bmatrix} -42.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 28.25 & -16.8155 & 0.0 \\ 0.0 & -28.25 & -16.8155 & 0.0 \\ 0.0 & 0.0 & 0.0 & -18.0 \end{bmatrix}$$

To get the resulting matrix from motor torques to joint torques, we can use the transpose of $\mathbf{J}$, $\mathbf{J}^T$. However, since we are interested in the matrix from joint torques to motor torques, we also need to invert the matrix, so we get:

$$\boldsymbol{\tau}_{motor} = \mathbf{J}^{-T}\boldsymbol{\tau}_{joint}$$

where $\mathbf{J}^{-T}$ is the inverse and transposed jacoby matrix. The final torque $\boldsymbol{\tau}$ is applied to the WAM, and the simulation is progressed by one step, rolling out the behavior of the system in simulation, which restarts the loop. The loop is repeated until either the pole angle is greater than a threshold, indicating that the pole is no longer approximately balanced, or a maximum number of steps has been fulfilled, which marks the graceful end of the stabilization process, meaning the pole was successfully balanced for the desired period. Satisfying either of these two conditions leads to an immediate termination of the LQR control. As a side note, for the MuJoCo simulation, we do not include the tactile sensor as it is (currently) not modeled in MuJoCo.

4.4. ROS-based Simulation Control

The advantages of the pure MuJoCo simulation are twofold: First, we can directly and perfectly access the full system state for our control problem. Second, since the MuJoCo roll-outs can be controlled, we always have a sequential pattern consisting of getting the complete system state observation, computing the action, and performing the roll-out. However, for the real WAM setup, both these circumstances are no longer given. The accessibility of the robot and pole state is neither perfect nor does it come from a single source, and there is no guarantee that, for example, our new pole observation arrives in time for computing the new action. This is caused by the fact that the real WAM software stack spawns multiple ROS-nodes through a launch file (details see appendix A), communicating with each other by publishing their respective information to ROS topics other nodes can subscribe to. This form of communication, called message passing, introduces problems like missing or delayed messages, leading to empty observations.

To bridge this gap to real-world deployment on the WAM, we adapt the simulation to mimic the real data observation process and modify the control pipeline to a ROS-based control loop implemented in C++ [64], inspired and amended from Klink [25], integrating RViz for visualization. For this simulation, the roll-outs of the reaction of the robot to the torques and the pole tilts are still performed in the MuJoCo environment, but can easily be replaced by the actual hardware and real roll-outs. Contrary to the MuJoCo standalone, in the ROS simulation, the real robot is set to be in a starting position that is generally not equal to the desired position (although configurable). This is reflective of reality because the real WAM (after switching on the electricity) is hanging in a random position and first needs to be brought in $\mathbf{q_t}$ for balancing. To get to $\mathbf{q_t}$, we move the robot with a PD controller gradually along a trajectory starting from the current position and holding position at $\mathbf{q_t}$ after reaching it. We compute the LQR controller gain matrix $\mathbf{K}$ and gravity compensation $\mathbf{gc}$ as prior. The pole is artificially reset to the target value, to ensure a start in $\mathbf{q_t}$. Finally, we switch the controller from PD to the LQR and enter the control loop.

The control loop largely functions the same as for the pure MuJoCo simulation, but since the data is now transmitted via ROS messages or interface handles like for the real setup, the state is no longer perfectly and completely given but has to be combined from pole measurement and robot state measurement, with the possibility of a missing pole measurement. A picture of the ROS-Sim pipeline can be seen in Figure 4.3.

The control loop consists of 3 major steps: First, the robot state is accessed through the encoder of each joint, which can be inaccurate in the real world, since the encoders need

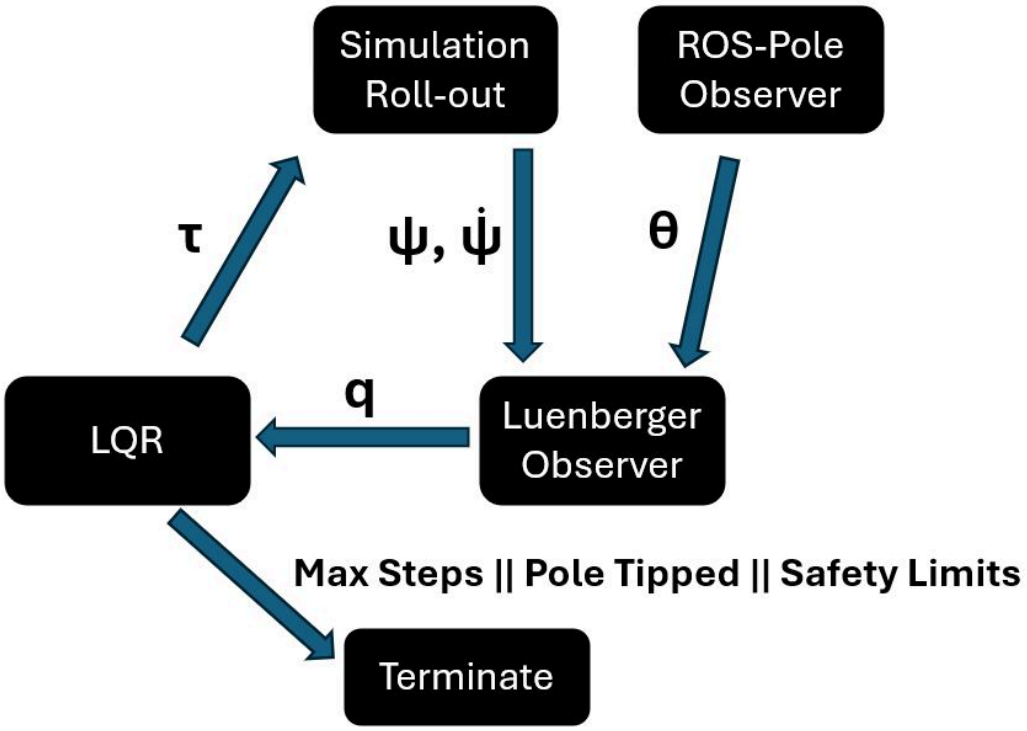


Figure 4.3.: ROS-Simulation control pipeline

to be initialized and can have slight skews. However, for the simulation, this still accesses the accurate MuJoCo data. The robot state and velocity $\psi, \dot{\psi}$ are combined with a pole state measurement $\boldsymbol{\theta}$ coming from a ROS-node that emulates an external camera motion capture system (OptiTrack).

Just like in Optitrack, $\boldsymbol{\theta}$ is created from a direction vector with respect to the global frame $\mathbf{dir}_g \in \mathbb{R}^3$ that represents the direction in which the pole is pointing. To infer $\boldsymbol{\theta}$ from $\mathbf{dir}_g$ we need to perform the following steps: First, we transform $\mathbf{dir}_g$ to the end-effector frame $\mathbf{dir}_l \in \mathbb{R}^3$ by computing the position and orientation of the end-effector frame with a forward kinematic call based on the robot joint angles. Given the transformation $\mathbf{T}$ from global to local end-effector frame, we compute

$$\mathbf{dir}_l = \mathbf{T}\mathbf{dir}_g$$

Then we split $\mathbf{dir}_l$ into the tilt in x and y direction by computing:

$$x = \mathbf{dir}_l(0),\ y = \mathbf{dir}_l(1), z = \mathbf{dir}_l(2),\ r = \sqrt{y^2 + z^2}$$
$$\varphi = -\tan^{-1}\left(\frac{y}{z}\right), \phi = \tan^{-1}\left(\frac{x}{r}\right)$$

ϕ is the tilt around the y axis and φ is the tilt around the x axis, together forming $\boldsymbol{\theta}$ as (φ, ϕ).

However, as mentioned, in this ROS-based scenario, there is no guarantee that $\boldsymbol{\theta}$ has been published when we query for it. Furthermore, we no longer get any information about the pole velocity $\dot{\boldsymbol{\theta}}$. Both these changes require handling. Since we always have an incomplete estimate of $\mathbf{q}$, either because of a missing $\dot{\boldsymbol{\theta}}$ or because we also lack a pole state estimate $\boldsymbol{\theta}$, we rely on a Luenberger-Observer for a complete state estimation of $\boldsymbol{q}$. As outlined in section 3.1.4, a Luenberger-Observer is capable of working with incomplete measurements of the state, estimating the missing state based on the dynamic equations of the system. Therefore, for the ROS-Simulation, we use a Luenberger-Observer to get a complete state estimate at all times. To this end, we feed the observation ($\boldsymbol{\psi}, \dot{\boldsymbol{\psi}}$, and possibly $\boldsymbol{\theta}$) to the Luenberger-Observer, which executes its internal state estimation and updates the state according to the dynamics equations outlined in section 3.1.4.

Given the resulting complete state $\mathbf{q}$, we again compute the necessary joint torques and convert and apply them to the motors. We apply the torques to the motors because in the ROS simulation, we always model the robot realistically, with tendons and motor currents. The simulation is computing the resulting roll-out, and the loop repeats itself. The loop only stops after generating the complete state when either of the aforementioned stop conditions is met or if any of the safety limits, limiting the robot joints' positions and velocities, are violated. This is necessary to prevent damage to the robot on the real system.

4.5. Real World Control

We now present the real WAM control pipeline for a visually inferred pole state via Optitrack. Compared to the simulations, several changes and adaptations need to be made to enable pole balancing. A picture of the real WAM pipeline can be seen in Figure 4.4, and it consists of 4 steps. It is worth noting that the roll-outs of the robot's response to the torques and the pole tilts are now performed in the real world on the real hardware. This comes with the downside of not having perfect data for both the pole and the robot,

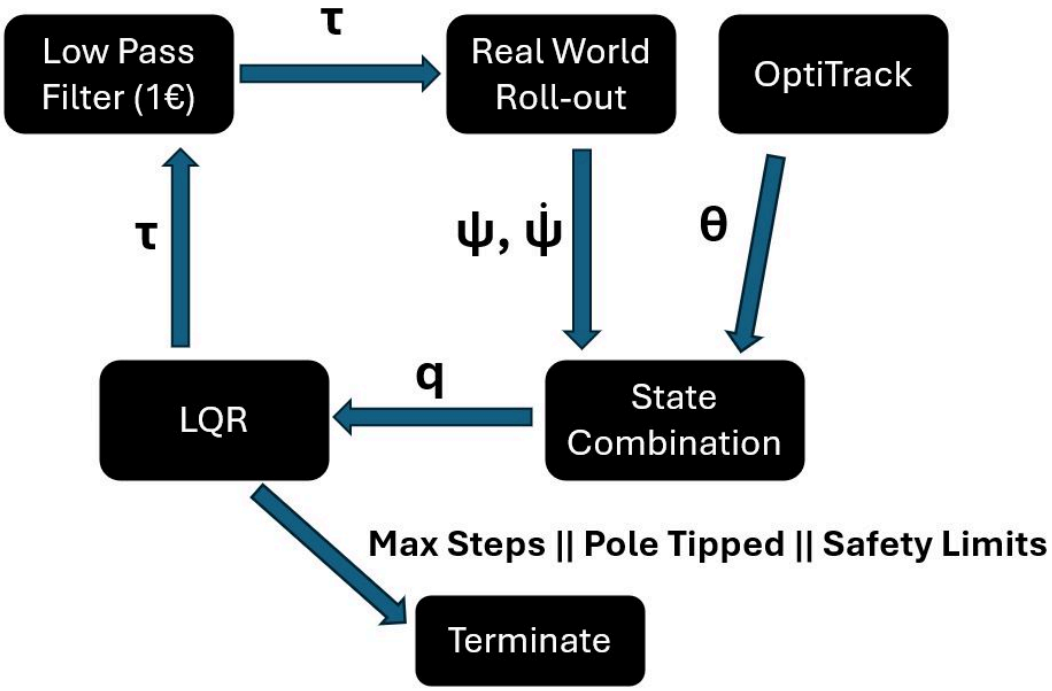


Figure 4.4.: Pipeline for the real WAM, estimating θ with OptiTrack

due to their estimation by Optitrack and the joint encoders. Furthermore, we need to amend the control loop to be able to balance the pole reliably and safely, i.e., with no excessive movement or drops of the pole for the free-moving pole setup to avoid damaging the robot, pole, and gel of the tactile sensor.

For the first step, just like in the ROS-based simulation, we again read the robot state and query the pole state, which is newly estimated by Optitrack. Again, we have no guarantee of having a pole observation, but instead of using a Luenberger-Observer to receive a full state $\mathbf{q}$, we rely on the old value if no new data is present and approximate the velocity with a simple finite difference computation. We got rid of the Luenberger-Observer as it caused unnecessary overhead and problems during stabilization based on tuning the gains for velocity and position $\mathbf{L_P}$ and $\mathbf{L_P}$. During roll-outs, we saw better performance with the simple finite difference approximation.

As a second step, we again compute the torque but we add 4Nm on the second joint to $\mathbf{gc}$ since the computed torque is not capable of holding the robot in the desired position,

otherwise, likely due to a model inaccuracy. Without this manual adding the robot arm falls during each roll-out, once switching to the LQR, since the WAM experiences an abrupt acceleration of the second joint, which has a detrimental effect on balancing.

After computing the joint torque, we chose to additionally filter them with a 1-Euro-Filter [53], which is a low-pass filter (see equations in section 3.1.5). The filtered torques proved to be an improvement on the real robot deployment, as the naive LQR control on the real robot results in jittery torques, i.e., torques that oscillate around a certain value. The oscillation is likely based on the noise in the measurements of both the pole and the robot state, which gets amplified by the high gains we need to hold position and react to falls of the pole quickly. Omitting the filter leads to loud noises and jittery movement of the overall robot, which we deemed dangerous and unhealthy for the robot, and thus relied on the filter to smooth the behavior. With the filtered torques, we obtain a much smoother trajectory per roll-out involving less motion and more stability. Lastly we apply the filtered torques to the motors and the loop continues. The termination conditions are the same as for the ROS-Simulation.

Regarding the pole state estimation through OptiTrack, the pole's direction vector $\mathbf{dir_g}$ is estimated through a least squares line fit on the marker 3-D positions to estimate the direction vector $\mathbf{dir}_g$ going through all markers in 3-D Space. The marker positions published to the Multicast address specified in Motive get processed by the ROS-node. For a robust line fit, we remove other reflective objects from the scene and mask parts of the walls to ensure no markers but the pole markers are used for a line fit (see A for a more detailed description of the pole tracking).

Furthermore we need to set the frame rate to 500Hz as otherwise the frequencies of robot and OptiTrack do not match. The higher frame rate comes with the cost of a reduced field of view (FOV) for the respective camera, as handling a larger FOV is not feasible at high rates. Therefore we need to ensure that the cameras capture the pole.

4.6. Tactile Based Pole State Estimation

Ultimately, our goal is to replace the OptiTrack-based estimate of the pole state with an estimate based on the tactile sensor observation and still be able to balance the pole. To realize this replacement, we need to learn a mapping from the tactile sensors' observations to the angles of the pole, so we can use the tactile information to control the system.

4.6.1. Datasets

As outlined in section 3.3, there are several deep learning approaches capable of learning an arbitrary mapping from data. To learn these mappings, we need data to train on. Therefore, we gather multiple datasets through roll-outs of our real LQR controller on the WAM for the free pole setup on the tactile sensor. The pole is placed on the gel, approximately in the middle, and held upright by a rope, released upon starting the LQR control. We try to minimize the difference in the start conditions on the sensor and the pole tilt to have a dataset with similar trajectories, making learning easier. For each control step, as outlined in the previous control pipelines, we log the pole state and velocity, the robot state and velocity, the OptiTrack pole estimate, and the timestamps of our LQR control loop in each control step, written to a .bin file.

To log the data from the tactile sensor, we spawn another ROS-node and synchronize the tactile sensor recording loop to the start of the LQR controller via a ROS-message. This synchronization is only approximate, which is why we need to preprocess the data later for training. Once the ROS-message arrives, the tactile sensor is enabled to record the events. The camera of the tactile sensor always publishes a continuous stream of events. To turn the stream of events into discrete events corresponding to individual time-steps, we slice the stream into parts of discrete time intervals via an Event-Stream-Slicer provided by the respective software of the camera. We set the frequency of the slicer to 500 Hz, like for the WAM, meaning every 0.002 seconds, we group all events that occurred during these 0.002 seconds into this batch.

After these initial steps, we enter the tactile loop. A visualization of the tactile sensors loop can be seen in Figure 4.5. In this loop the Event-Stream-Slicer is continuously queried

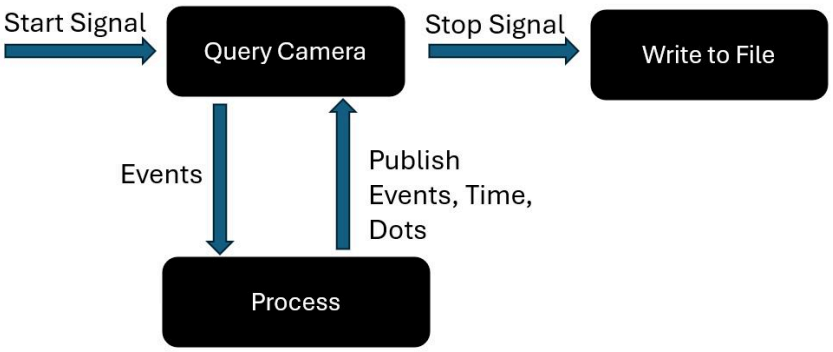


Figure 4.5.: Visualization of the tactile sensors loop

for the current batch of events. If the current batch has any events, we process them by first filtering background noise based on an activity filter, again provided by the camera software stack. The remaining events, which may be none, are put in a set and appended to the list of recorded event batches. In [9], the authors presented an algorithm that can track the locations of dots in the gel. We use this algorithm for the dotted gels, meaning for the current events, we update our perceived dot locations and save them in a set. Finally, the current time is appended to the recorded timestamps. The events, time, and dots are published to a ROS-topic, so that we can use them for tactile state estimation.

The loop is repeated until the LQR control is terminated, signaled by another ROS message, stopping the recording. The recorded events and timestamps, together with the dot locations if tracked, are written to a .npz file.

With this data generation process, we created 3 datasets on different setups for gel, pole, and control algorithms. We captured:

1. 166 trajectories on the dotted gel controlled by the LQR (referred to as dot-dataset)
2. 10 trajectories on the normal gel (without dots) controlled by the LQR (referred to as image-dataset)
3. 33 trajectories on the normal gel, where we let the pole freely fall in different directions without the LQR control and simple gravity compensation as a control law. (referred to as classification-dataset)

All datasets share the recorded pole angles and velocities by OptiTrack, the robot configuration and the events from the tactile sensor for each respective ROS-nodes control step interval of 500Hz (the nodes do not run synchronously, but have the same frequency).

Dot-Dataset

For the dot-dataset, we used a $1\,\mathrm{m}$ aluminum pole and executed the LQR balancing on the dotted gel. To track the dots, we rely on the tracking algorithm proposed in [9]. This algorithm tracks the 63 x and y locations of each dot in the dotted gel starting from an initial position. In each time-step there is an abundant amount of events triggered, because the movement of the dots in the gel is causing frequent shadows cast and thus events.

However, most events are triggered by the dots and not the pole itself, which is why it is hard to tell where the pole is standing when viewing, for example, a GIF of the occurred

events. Still, we captured this dataset because in [9] they showed that it is possible to accurately track the forces applied to the gel when processing the dot locations with a simple MLP. Since different tilt angles result in different kinds of pressure and force distributions on the gel, we hoped to mimic this success. We capture a large dataset of 166 trajectory consisting of 20000 control steps each, with manual placement of the pole approximately upright and in the center of the gel in order to capture a variety of possible trajectories and dot locations.

Image-Dataset

For the second dataset, we used the $1\,\mathrm{m}$ steel pole and executed the balancing on the normal gel, without the dots. This has two reasons: First, in the dotted gel, most events do not correspond to the actual pole but to the dots in the gel, which makes it hard to recognize the pole. Second, the heavier steel pole is deforming the gel more than the lightweight aluminum pole, which is beneficial for event generation as outlined in section 3.2.4. We noticed in the first dataset that the lightweight aluminum pole was causing very few events due to its low weight, which is why we opted for a heavier pole on the normal gel. Combined, these two changes enable us to see the pole more clearly in e.g. GIFs (see figure 4.7) of the recorded events, as more events are triggered and all events were caused by the pole (or noise).

Given that we can now see the pole isolated and clearly, we decided to turn the registered event stream into images, whose creation process is described in detail in section 4.6.2.

The second dataset consists of 10 trajectories of 20000 control steps on a normal gel. For each step in each trajectory, we create the images according to 1 and save them to disk as they are not feasible to be loaded in RAM all at once.

Classification-Dataset

In the third dataset, LQR control was disabled, and the robot maintained position using only the gravity compensation term. This causes the pole to eventually fall after the rope is released, ending the control loop and trajectory recording once it tilts beyond the set threshold. We aimed to capture falls in roughly every direction across the 360-degree space, constrained by the available lab area. This dataset was used to judge the tactile sensors' capabilities to classify the pole fall direction. For this dataset, similar to the

images of the collage in Figure 4.7, events of a single trajectory are only in certain regions, which makes the dataset the simplest and shortest of all three.

We captured 33 trajectories, each under 1000 control steps due to the pole's fall. The pole and gel setup match that of the image dataset. Figure 4.6 shows the pole tip positions for each trajectory in Cartesian coordinates, viewed from above and centered at (0, 0). As shown, the dataset is relatively small and doesn't cover all possible directions. We

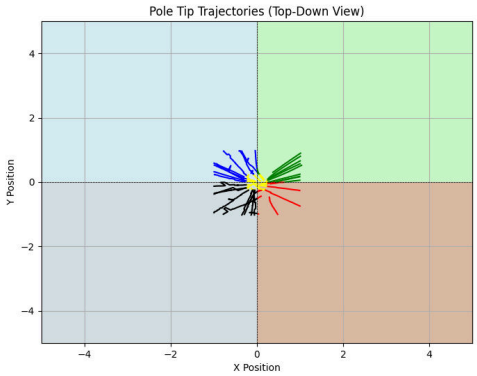


Figure 4.6.: Visualization of the trajectories in the classification dataset, displayed are the x and y coordinate evolution of each trajectory and the respective classification in e.g. the 4 quadrants as their ground truth label.

limited its size since classification inspection supports, but isn't central to, the main work, since classification networks cannot directly replace the continuous pole angle estimate required for LQR control. Thus, their application is limited, and the dataset mainly serves as a research tool for evaluating fall direction assessment via tactile sensor data.

4.6.2. Data Preprocessing

To enable easier training, we preprocess the raw events and dot locations coming from the tactile sensor. For the dot locations, we compute the mean location of each dot in our dataset and then compute the relative position of each dot to its mean position.

To create the images, we start with two empty gray-scale images initialized to 0 at the beginning of each trajectory. The image resolution of the Evetac event-based camera is

640×480, capturing the whole gel. Since images of this size are quite large, we down-sample the image size by a factor of 5, giving a resolution of 128×96. This moderate down-sampling provides a manageable amount of data while still preserving details. Pixels of the original event-based camera that fall under the same down-sampled image pixel are treated as equals.

We filter each event batch with a window-based filter that moves a 80×80 window over the original image grid and removes the respective events if there are fewer than 3 events in the window. This additional filtering is performed to improve the noise filtering that is applied by the tactile sensor ROS-node. After filtering, we split the events into positive and negative events by their polarity. We use both the negative and the positive events to change their respective image, i.e., we have an image of the positive events and an image of the negative events. We map the original pixel coordinates to their down-sampled coordinates and increase the value of the respective down-sampled image pixel by an event-gain-factor δ. Furthermore, to consider the recency of the pixel change, we decay the image by a decay factor γ, so that over time each image pixel returns to its initial value.

To ensure a valid gray-scale image, we clip the images to lie in $[0, 1]$. A pseudo-code description can be seen in algorithm 1. $I[\cdot]$ describes the pixel value at the pixel coordinates defined by the variable in the brackets.

Input : List of event-batches of a trajectory (list of list)
Output : List of corresponding images (List of images per time-step)
$Imgs \leftarrow \{\}$
$I_{pos} \leftarrow \mathbf{0}$
$I_{neg} \leftarrow \mathbf{0}$
foreach *$events$ in Event-batches* **do**
  $points \leftarrow filter(events)$
  $pos, neg \leftarrow split\_points(points)$
  $pos, neg \leftarrow map\_to\_downsampled\_pixels(pos, neg)$
  $I_{pos} \leftarrow \gamma I_{pos}$
  $I_{neg} \leftarrow \gamma I_{neg}$
  foreach *p in pos* **do**
    $I_{pos}[p] \leftarrow I_{pos}[p] + \delta$
  foreach *n in neg* **do**
    $I_{neg}[n] \leftarrow I_{neg}[n] + \delta$
  $I_{pos} \leftarrow clip(I_{pos})$
  $I_{neg} \leftarrow clip(I_{neg})$
  $Imgs.append((I_{pos}, I_{neg}))$
return $Imgs$

Algorithm 1: Convert event-batches to images

δ and γ are free to choose and thus are evaluated in the experiments section. δ configures how strongly an event changes its respective pixel value, and γ configures how fast we decay back to the initial state. In the extreme cases for $\gamma = 0$, only the events from the current time-step influence the image, whereas for $\gamma = 1$, all recorded events influence the image in the same way, leading to no forgetting of past events.

This image-based interpretation of the tactile events intuitively describes where events were triggered and how long ago they appeared, converting the sparse event-based information to a global and history-like representation of the data. To visualize the images to get an idea of what data we train on, we provide a collage of images in Figure 4.7.

For this collage, we merged the two training channels of the positive and negative event images, by creating only a single image, that starts at 0.5 (the initial gray value), incrementing the pixel values for positive events and decrementing them for negative events. Therefore, these images are different than the ones we train on, yet for human interpretation of the events, it is a useful visualization. The image collage was taken from a single GIF showcasing the manual tilt of a pole towards the bottom of the gel, including an

eventual fall of the pole, releasing any pressure on the gel. As you can see, the image is initially completely gray, i.e. all image pixels have the value of 0.5 (which again is not reflective of the training data).

Once the pole starts moving, around image 3, we can see a slight black edge where the pole's bottom part is pressing into the gel more heavily, and two slightly white regions where the light intensity increases. Light intensity increases in these regions as the pole edges lift due to the tilt, no longer blocking the light. This pattern is getting stronger over time (as more events occur at these places) until the second-to-last image. Furthermore, the two regions where the light increases (i.e. where the pole edges lift off from the gel) move closer together along the radius of the edge of the pole.

Finally, in image 15 the pole falls off from the gel completely, leading to a lot of surface being illuminated in a short time period resulting in a lot of positive events.

Since the timestamps of the tactile data(events and dot locations) and the OptiTrack angles are not the same, we interpolate the angles based on the tactile timestamps in a linear fashion. The interpolated angles are our ground truth data to learn the models on. Furthermore, we normalize these angles for learning stability and convenience with a Min-Max-Norm to lie in $[-1, 1]$.

4.6.3. Training And Evaluation

A central contribution of this thesis is the training and evaluation of models for tactile-based state estimation, including exploring various datasets and model configurations to find an effective setup. For learning the models, we create a PyTorch[65] based train script, configuring models and datasets. To regress the pole angles, an abstract class defines the learning, evaluation, and logging process, such as plotting, writing the losses to a JSON file, and organizing folders. Specific architectures, such as MLP, CNN, LSTM, or PointNet, inherit from the abstract class and define the data processing step, as well as the dataset itself. A visualization of this can be seen in Figure 4.8. All the specific architectures turn the data into a feature vector at the end, which is processed by a separate linear layer of the abstract regression net into the angle, which is a 2-dimensional vector (or, in the case of classification, the vector has the same number of dimensions as classes). The networks are learned via simple back-propagation on the error they predict. To evaluate each model, we use 5-fold cross-validation on the dataset. The data is split into 5 parts, and in each iteration, 4 parts are used for training and 1 for evaluation. This process is repeated with different splits until every part has been used for evaluation once, resulting

in 5 trained models. The process is visualized in Figure 4.9. The performance of the model as a whole is judged based on the mean performance of the individual trained models. This procedure is common to get a better understanding of how the model performs in general on the dataset, since evaluating only a single train-test split can often be biased for judging the performance.

To test the learned models in the real world control loop we import a just-in-time (jit) scripted version of the model in the C++ pipeline, loaded during initialization of the LQR controller. The jit conversion optimizes and serializes the PyTorch model so that we can use it in C++. We replace the Optitrack published pole angle with the prediction of our network, based on the tactile data given at the control time-step. The tactile data is continuously published by the ROS-node of the tactile sensor as mentioned in the previous section. The rest of the loop remains the same and can be seen in Figure 4.10. We evaluate the respective simulations and the real control loop, as well as the tactile-based state estimation, in the next section through a variety of experiments.

The code is available at:
https://git.ias.informatik.tu-darmstadt.de/tactile-sensing/tactile-pole-balancer.git.

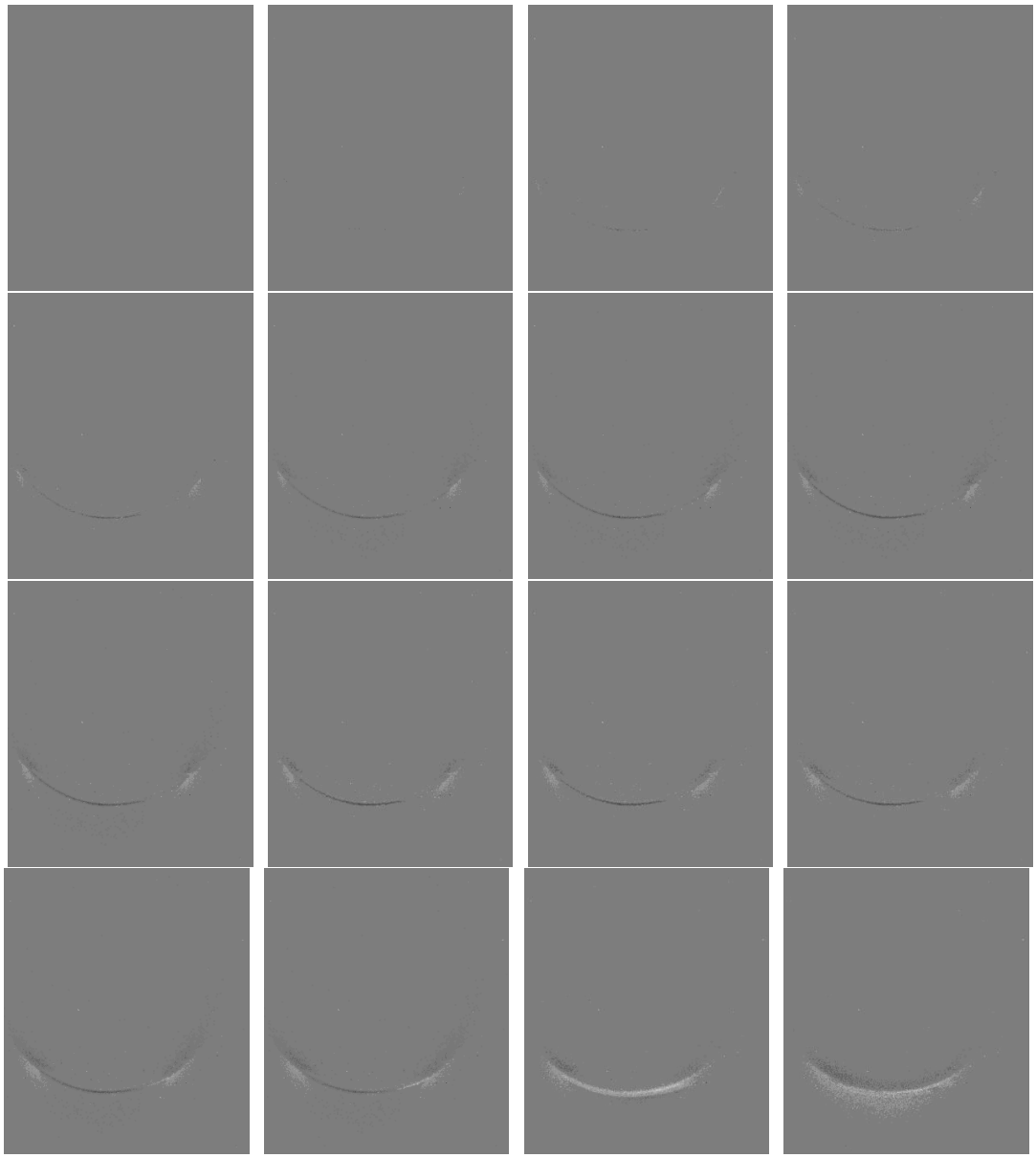


Figure 4.7.: Extracted frames from GIF (every 10th frame) of a pole falling to the bottom end of the sensor, white is close to 1, black is close to 0, gray is 0.5

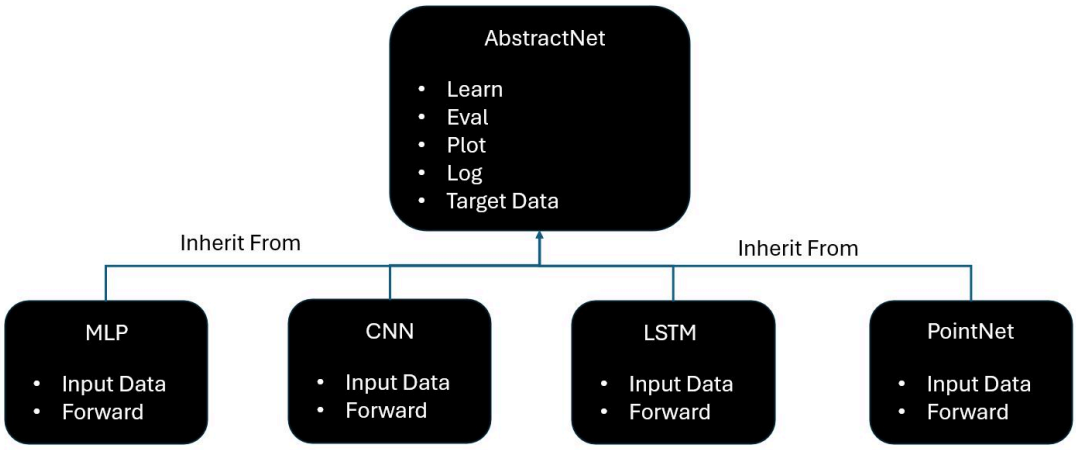


Figure 4.8.: Abstract net and concrete implementations

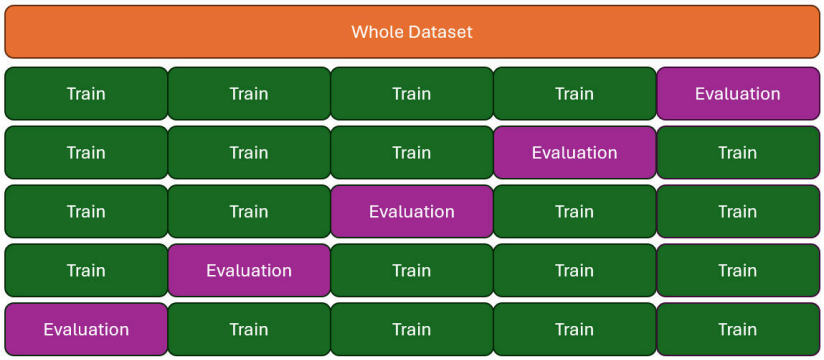


Figure 4.9.: Visualization of a 5-fold cross-validation

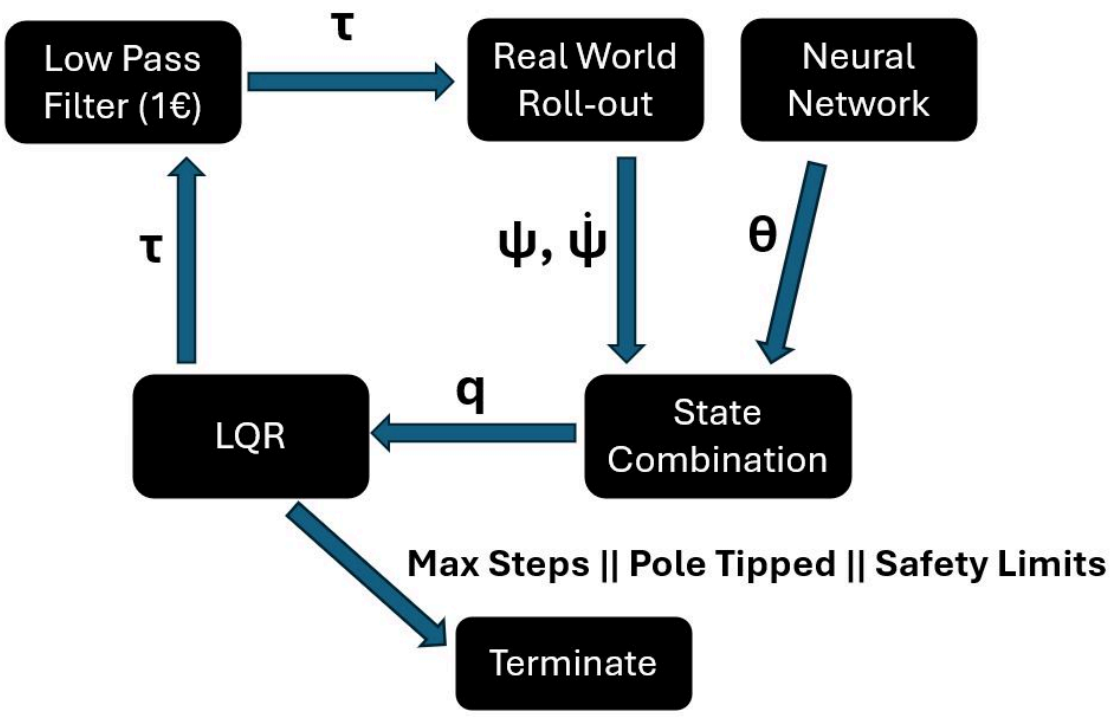


Figure 4.10.: Tactile-based control loop, using the neural network for predicting θ

5. Experiments

In this section, we outline the experiments conducted to evaluate the previously presented methods. These experiments include extensive LQR parameter grid search in simulation, a smaller-scale LQR parameter search on real hardware, and evaluation of various neural network configurations for tactile-based state estimation. All experiments were conducted on a AMD Ryzen 9 3900X 12-Core Processor with a NVIDIA GeForce RTX 2080 with CUDA Version: 12.2 as GPU. The C++ Version is 11.4.0, the Python version is Python 3.10.12.

5.1. Simulation LQR Experiments

As mentioned, to learn the tactile-based state estimation, we need a controller that balances the system to record tactile observations during balancing. For the controller, we choose an LQR and search for suitable parameters that result in a reliable stabilization of the pendulum. To evaluate different LQR parameters, we test them in our simulation to avoid damage to the real system and to make use of the faster inference in simulation. The inspected LQR parameter serves as a starting point for the real-world implementation of the setup on the WAM.

5.1.1. Setup

The tactile sensor is not modeled in the simulation. We create multiple URDF and XML files describing the robot-pole-system and its dynamic behavior we want to test. Especially we focus on different kinds of joints at the end-effector such as a 360 degree joint and a free joint. Like in [25], we perform experiments with an attached 360-degree joint allowing unrestricted rotation of the pole without translation along the x, y, or z axes, similar to the setup. Furthermore, we use a free joint that permits full translational and rotational movement in 3-D space. Both are realized for our real-world LQR experiments

in section 5.2. Simulation parameters like friction, armature of the joints, and the tension of the cables were estimated with a system identification on the real system, and the WAM URDF structure and content were taken from the WAM Data-sheet.

5.1.2. MuJoCo LQR Gridsearch

The goal of the MuJoCo LQR gridsearch is to find a well-performing LQR controller, meaning low velocities and deviations for $\mathbf{q_t}$ through a gridsearch over the different components of $\mathbf{Q}$ and $\mathbf{R}$, which are free to choose. Although not naively present in the simulation, we need to factor in the noise inherent in the observations of the state, in our case, the pole angle, for a more realistic balancing simulation. We assume that the noise is representable by a Gaussian with zero mean $\mathcal{N}(0, \sigma)$. As the amount of noise for the real system is hardly quantifiable, we considered $0.001, 0.005, 0.01, 0.02$ for σ. For $\mathbf{R}$ we choose a value of 0.025. For $\mathbf{Q}$ and its values d_1, d_2, d_3, d_4, we consider the following discrete values: $0.01, 0.1, 1, 10, 100, 1000$.

We researched the performance of any of the combinations of these LQR parameters on both the free and attached setups, described earlier. The free-moving pole introduces the problem that during control, the pole must not slip off the end-effector of the robot, which happens if the pole tilts for too long in a certain direction or if we abruptly change directions with our control signal so that the pole loses surface contact. Both situations are devastating for balancing and must be avoided. Thus, we cannot employ a bang-bang controller with high gains, but we also cannot use a very soft controller with low gains, since either we lose contact with the pole or let it slide off.

For each setup and configuration, we run 10 trials with 4000 steps in the MuJoCo simulation and log the position and velocity of the pole as well as the overall state error. Since we want to stabilize the pendulum around $\mathbf{q_t}$, we compute the mean state error of each configuration to $\mathbf{q_t}$. For this, we compute the sum of entries in $\mathbf{q}'$ at each time-step, average these sums over the full trajectory for each run, then average across all runs for a given LQR parameter configuration. The configuration with the lowest final score is considered the best.

For clarity, due to the amount of data, we limit the depiction of the results in this section to table 5.1, summarizing the performance of the top 20 configurations in terms of mean pole tilt and velocity around the X and Y axis as well as the state error for different levels of noise for both an attached and a free pole. We provide multiple figures of the respective configurations and their results in the appendix (see figures A.1 to A.16).

Setup/Noise	Pole Tilt x/y (rad)	Pole Velocity $\dot{x}/\dot{y}$ (rad/s)	State Deviation
Attached/0.001	0.0004 / 0.0006	0.0053 / 0.0042	0.0151
Free/0.001	0.0004 / 0.0003	0.0016 / 0.0022	0.0078
Attached/0.005	0.0021 / 0.002	0.0191 / 0.02359	0.06682
Free/0.005	0.0023 / 0.002	0.0072 / 0.0092	0.0362
Attached/0.01	0.0057 / 0.0041	0.0333 / 0.0458	0.1305
Free/0.01	0.004 / 0.0033	0.0151 / 0.0202	0.0727
Attached/0.02	0.0115 / 0.0092	0.0575 / 0.074	0.2301
Free/0.02	0.0147 / 0.0079	0.0250 / 0.0313	0.1456

Table 5.1.: Best respective configuration performance in terms of the mean state deviation, the mean pole tilt and velocity are explicitly listed and split in X and Y direction, separated by the "/" and are respective to $\mathbf{q_t}$

As an example, in Figure 5.1 and 5.2 we show the mean pole tilt (top) and velocity (bottom) evolution around the X and Y axis and the respective standard deviation across 10 trials during roll-out of 1000 steps for the best performing configuration for the attached setup with a noise level of 0.001 and 0.02 for σ. Again, the plots for the other configurations are in the appendix in Figures A.17 to A.22. As you can see in both the table and the plots, with an increase in noise, the position and velocity errors, as well as the overall state error, increase.

For a low level of noise, we achieve an almost perfect stabilization with low tilts along both axes and minimal velocity. For higher levels of noise, the orders of magnitude for both velocity and pole angle increase. This is expected because an inaccurate estimate of the state leads to a suboptimal action performed by the LQR, thus resulting in worse performance than for an accurate state estimate.

Noticeably, the free pole achieves comparable position accuracy to the attached pole, but usually has very different LQR parameters, resulting in less velocity. The lower velocity is the main reason why the free pole setup has lower overall state error. A possible explanation for this could be that the attached pole and the free pole have different behaviors due to friction and other physical effects. For the attached pole, the friction is determined by the friction of the joint, whereas for the free pole, it is determined by the

surface interaction with the end-effector. Furthermore, the free pole is a separate entity, whereas the attached pole is dependent on the robot's movement and position, which also likely results in different tilt angles for LQR roll-outs of the same parameters, explaining the difference in performance.

5.1.3. Effect of Non-LQR Parameter

Building the simulation of the pole balancing task comes with a few non-LQR parameter choices regarding the setup. We considered the following variations to the pole balancing setup overall: First, a high and lower configuration of the robotic arm shown in Figure 5.3. Second, we explored the effect of a longer pole on the balancing task. Third, we investigated the impact of positioning the pole off-center on the robotic arm, simulating real-world misplacement.

Balancing Configurations

The two configurations we inspected can be seen in Figure 5.3. In the first configuration, the WAM is holding a plate rigidly attached to its end-effector, much like a human holds and transports a plate, and the pole is put on top of it. In the second image, the pole stands on a plate mounted directly above the end-effector of the WAM, effectively extending it. In both cases, the plate works as an abstract surface on which the pole stands. We were interested in the influence of these different robot configurations for balancing, to find a suitable pose for the real WAM balancing configuration.

We again perform a grid search for the two setups with a free pole setup and compare the results both qualitatively and quantitatively. Noticeably, for the free pole setup, the lower configuration performs worse than the higher one, as can be seen in Figure 5.4, given the much higher mean state deviation sum of each configuration. Furthermore, for the higher configuration, multiple different LQR parameters perform very well, whereas for the lower configuration, there is a significant difference between the first configuration and the others, meaning the higher configuration is much more robust.

This behavior is reflected in the MuJoCo simulation roll-outs by excessive movement of the second joint in the lower position scenario, leading to a back-and-forth joggling of the pole, with possible divergence to joint limit values. This behavior is likely because in this configuration, the robotic arm only effectively uses 3 out of the 4 joints to control the pole, as the third joint axis does not point in either of the pole tilt axes.

Furthermore the movement of the arm in this setup introduces a significant slip on the plate over time, which can lead to a fall of the pole from the plate in the worst case. For these reasons and because in the higher setup the robot is below the pole, preventing occlusion for the real roll-outs, we chose the high configuration.

Pole Length

A longer pole makes the balancing of a pole easier for humans, because the higher inertia of a longer pole leads to a later fall of the pole as more force is required to rotate it, and thus the human observer has more time to react to falls. Therefore, we also inspect the effect of balancing a longer pole with the WAM. For our setup, we originally used a pole of length $0.374\,\mathrm{m}$ and decided to test the behavior for a pole of length $0.5\,\mathrm{m}$ as well, which increases the inertia around the relevant axis by a factor of approximately 2.389. We performed the same gridsearch and tested the resulting LQR behavior for multiple levels of noise for the longer pole. Again, each configuration in the gridsearch was rolled out 10 times for 4000 steps.

For a low level of noise $\mathcal{N}(0, 0.001)$, the longer pole helps stabilize, since we achieve a lower mean state deviation as can be seen in Figure 5.5. Noticeably, the LQR parameter for the longer pole is quite different from the shorter pole, despite the rest of the setup being equal. The longer pole has lower absolute gains for its LQR values, but both pole lengths can be stabilized sufficiently.

For a higher level of noise $\mathcal{N}(0, 0.01)$, the longer pole no longer helps stabilize, as can be seen in Figure 5.6. This behavior is likely caused by the fact that even a perfectly balanced pole has a decent amount of perceived tilt due to the added noise. Therefore, the controller needs to move the pole more often to balance it. For the longer pole, it is harder to correct state deviations that resulted from suboptimal controls, given the noisy input, requiring more force and thus motion of the robotic system, which leads to higher state deviations in the mean.

Position Offset

Accurately positioning the free pole at the center of the robotic arm, or later the tactile sensor, is challenging and unrealistic unless done in a simulation. Therefore, we were interested in the effect of positioning the pole with an offset to the center regarding the balancing task. We found that for higher offsets there is a slight but not significant impact

to the balancing task in terms of state error as long as the pole did not lose contact or slipped off of the plate's surface as can be seen in Figure 5.7 and 5.8. We tried offsets in multiples of $0.009\,\mathrm{m}$ to the center in both x and y direction, so that the pole initially still stands on the end-effector, which is of size $0.05\,\mathrm{m}$ in both x and y direction. We ran 100 trials for each offset, and as you can see, the further away we get from the center, the higher the mean state deviation becomes, although the error is not significant. A possible reason for this is that the gravity compensation term and control matrix no longer fit the actual setup, as in the URDF, the pole is always at the center, leading to different fall directions and speeds, resulting in more error.

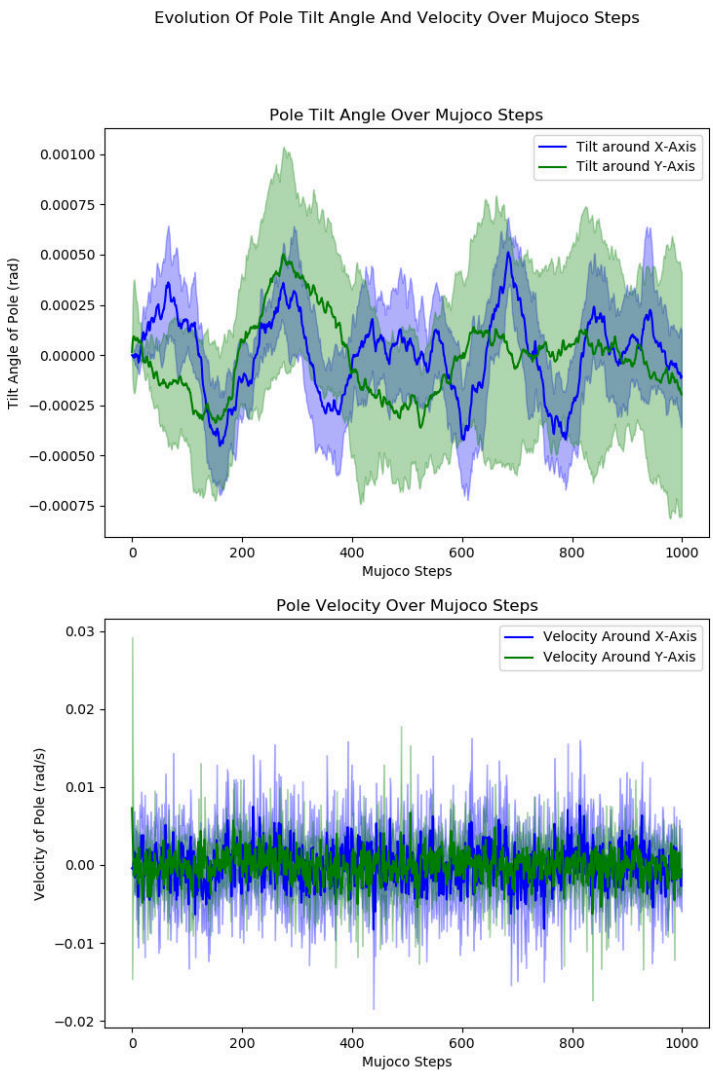


Figure 5.1.: Pole tilt angle and velocity – attached, noise 0.001

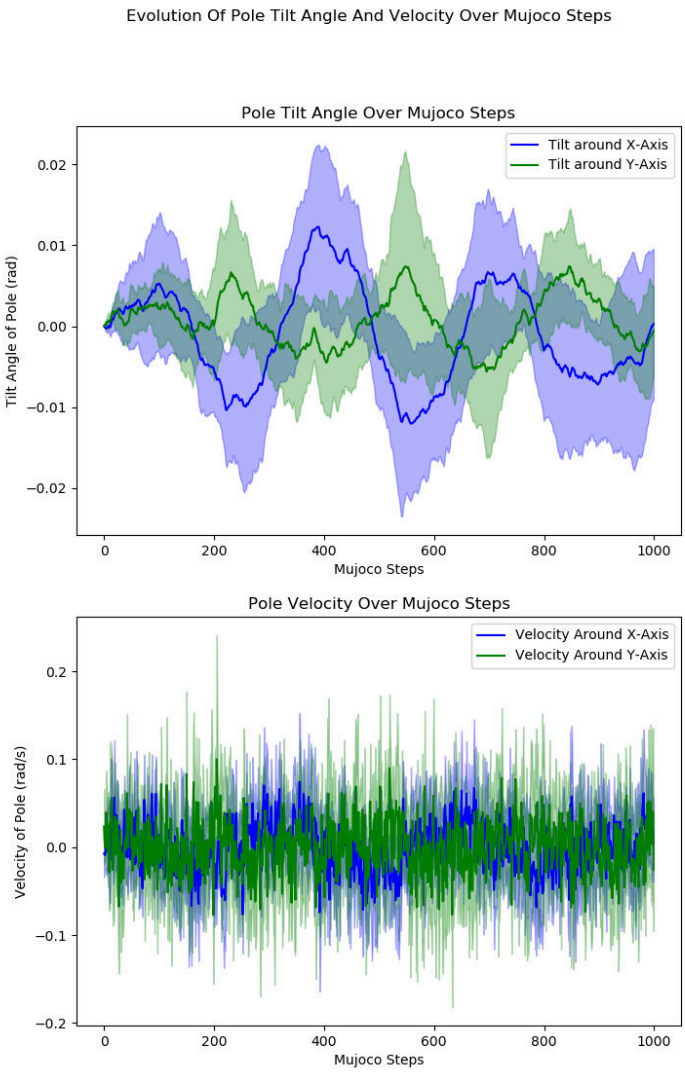
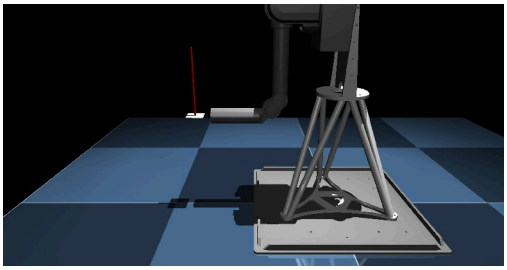
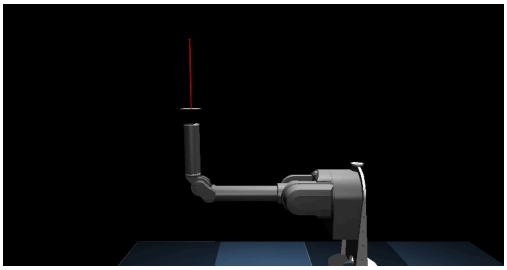


Figure 5.2.: Pole tilt angle and velocity – attached, noise 0.02

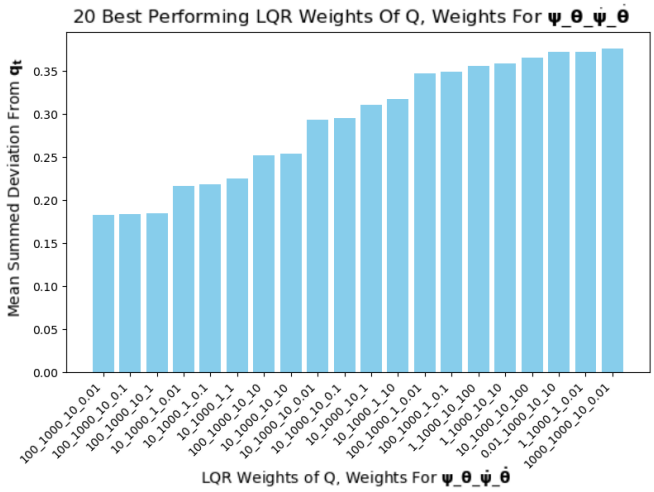


(a) low



(b) high

Figure 5.3.: MuJoCo screenshots of the different robot configurations



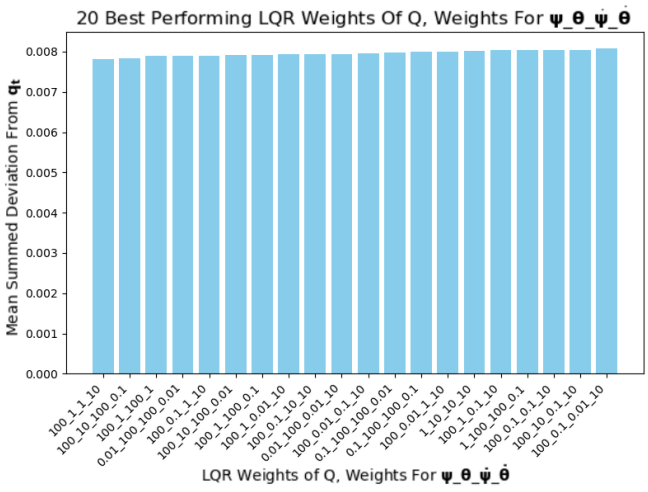


Figure 5.4.: Deviation from $\mathbf{q_t}$, noise 0.001 for low (left) and high (right)

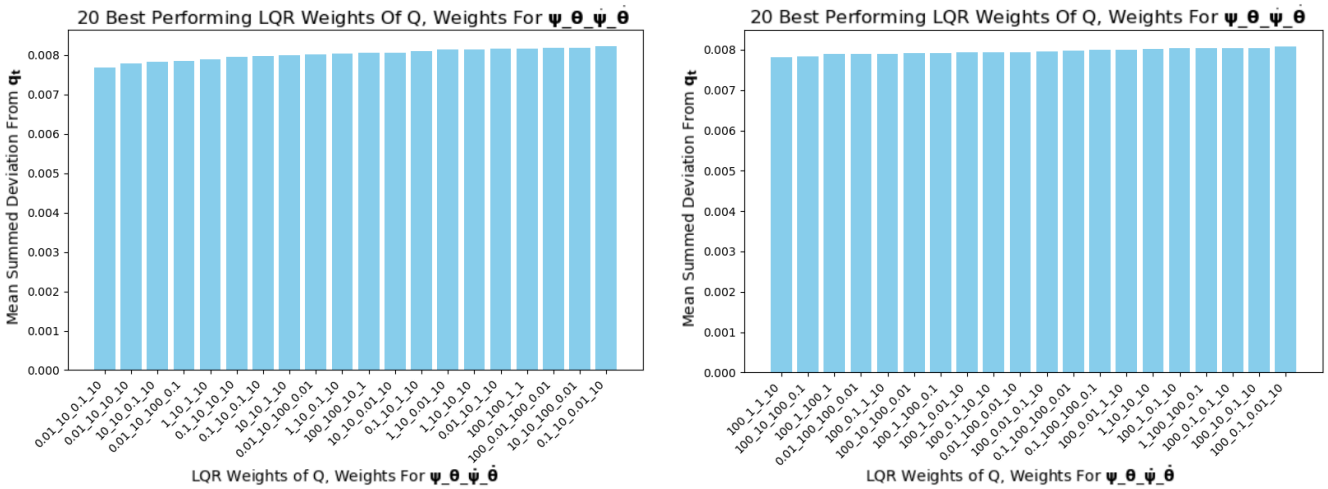


Figure 5.5.: Deviation from $\mathbf{q_t}$, noise 0.001 for longer (left) and shorter pole (right)

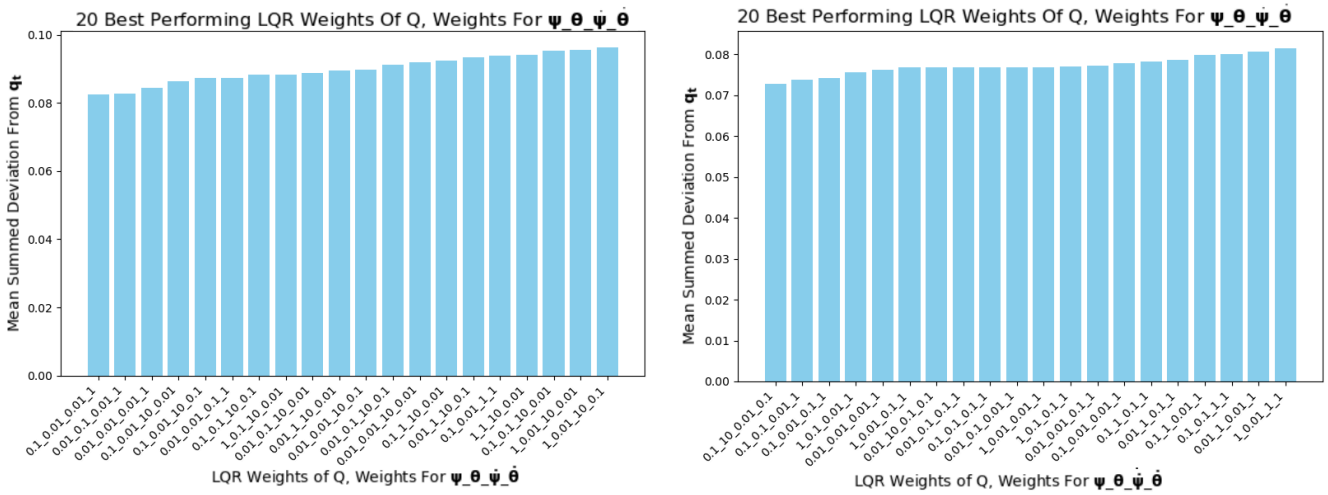


Figure 5.6.: Deviation from $\mathbf{q_t}$, noise 0.01 for longer (left) and shorter pole (right)

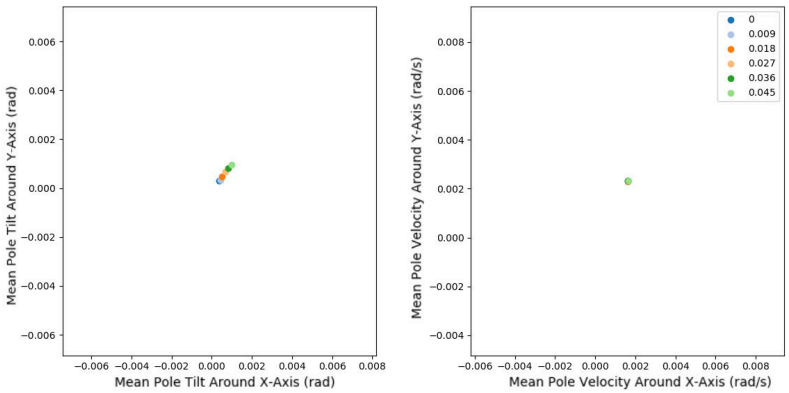


Figure 5.7.: Pole tilts and velocity for different offsets

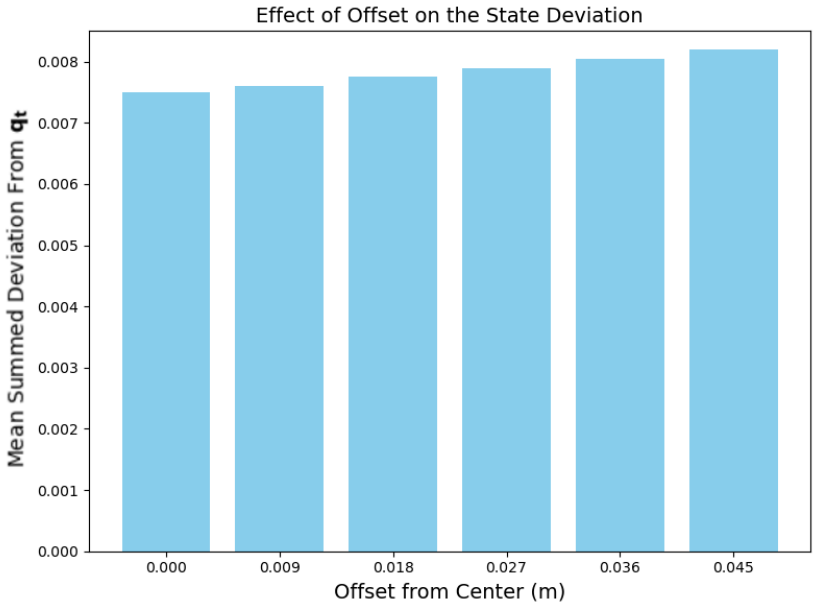


Figure 5.8.: State error per offset in m

5.1.4. Effect of Luenberger-Observer Parameter

Before deploying our LQR on the real WAM, we evaluated and stabilized the pole in the C++ pipeline simulation environment. This pipeline uses an Extended Luenberger-Observer for complete state estimation. As described in Chapter 3.1.4, the observer incorporates a system model and reacts to deviations via tunable gains. While this observer was eventually replaced with a finite difference approximation due to empirical instability during control of the real WAM, we first attempted to tune its performance by changing values for $\mathbf{L_P}, \mathbf{L_D}, \mathbf{L_I}$. We start by setting all diagonal values of $\mathbf{L_P}$ to 0.56569, $\mathbf{L_D}$ to 10 and $\mathbf{L_I}$ to 0 as proposed by [25].

To counteract the disproportionate velocity of the pole around the x-axis, as displayed in Figure 5.9, caused by jitter in the robotic arm, we reduced the L_d of the Luenberger-Observer. Instead of using a uniform value of 10, we applied alternating values of d and 10, where d specifically influences the joints affecting velocity in the x-direction. Figure 5.10 compares the performance for $d = 0.2$ and $d = 1$. With $d = 0.2$, we observe lower state error and a more balanced distribution across components. However, further reducing the D-gains (e.g., $d = 0.1$) leads to instability in the Y-direction.

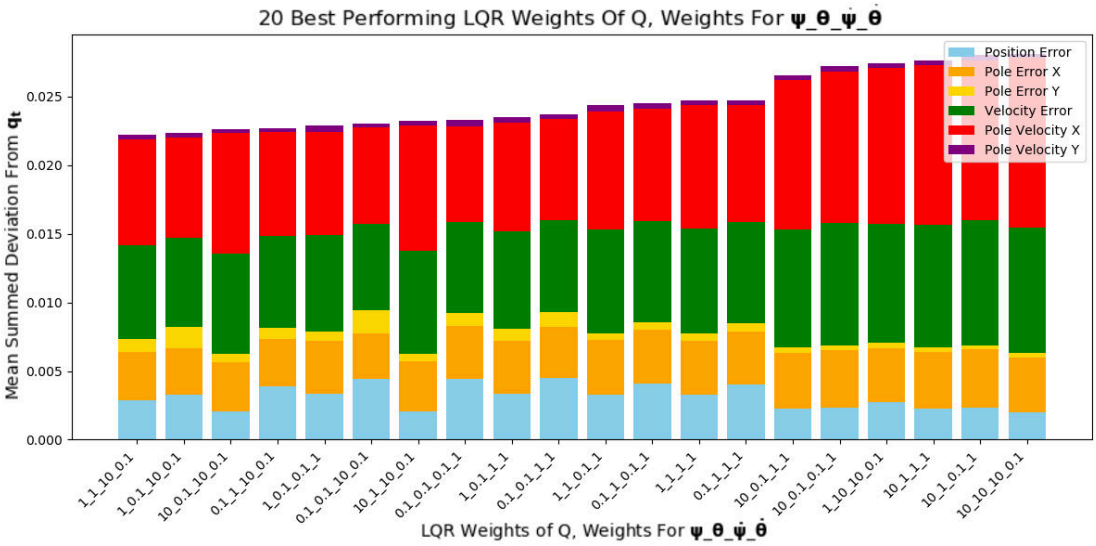


Figure 5.9.: State deviation split for different LQR configurations

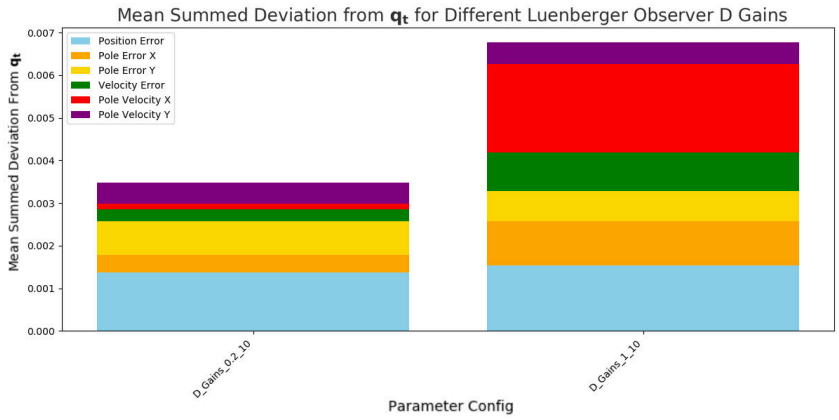


Figure 5.10.: Tuning $\mathbf{L_D}$ impacts stabilization

The state deviation for the best Luenberger Configuration is lower than the state deviations for the MuJoCo simulation (e.g., listed in 5.1), likely because the model-based update is improving the noisy state observations, leading to better controls. However, for the real roll-outs, the Luenberger-Observer caused issues. Specifically, the Luenberger-Observer state estimation leads to abrupt WAM movements, causing falls or throws of the pole, likely due to model inaccuracies or imperfect parameters for the real robot. Therefore, we ultimately removed the Luenberger-Observer and replaced its velocity estimation with a simple finite difference approximation of the last OptiTrack estimation and the current estimate.

5.2. Real Robot LQR Experiments

After analyzing the LQR behavior in simulation, we now focus on the real WAM. We shortly introduce the setup and evaluate a few LQR controller roll-outs.

5.2.1. Setup

As shown in the simulation, given low noise, a longer pole is beneficial for stabilizing. Since OptiTrack is capable of sub-millimeter accuracy, instead of a short pole of length $0.374\,\mathrm{m}$, we also use poles of length $1\,\mathrm{m}$ for the real setup, visualized in Figure 5.11. The

Figure 5.11.: Poles of different lengths with markers (duct-tape)

longer poles are hollow cylinders, the silver one in the middle is made out of aluminum, and the pole at the bottom is made from steel. The one on the top is the shorter pole and is fully made out of steel. Since steel is much denser than aluminum, we also have different weights of the poles. All of the poles, as well as the marker duct tape, were bought in a local hardware store, and their physical properties, such as mass and inertia, were measured and computed and put into respective URDF files, describing the joints and poles.

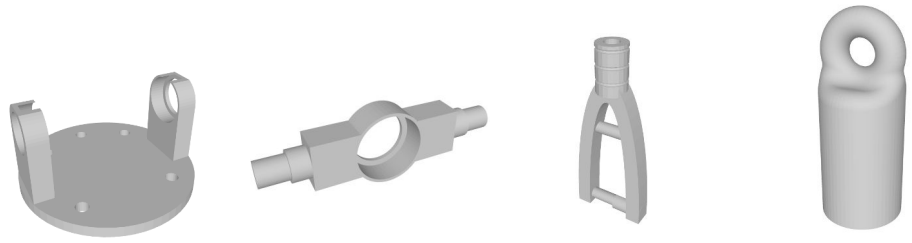
OptiTrack

Regarding the pole estimate in real life, we place a set of cameras around the scene, filming the Cartesian position of $\mathbf{q_t}$. For markers, we rely on UV-reflecting duct-tape stripes placed on the pole. The coordinates of our markers captured are estimated relative to the OptiTrack frame, which is defined by a calibration procedure.

Attached Pole through Spherical Pendulum

We create a spherical pendulum with 3-D-printing like in [25]. The pole is plugged into the top of the assembled joint and the entire joint is attached to the end-effector via screws. The joint consists of rotating x and y-axis parts, enabled by ball bearings. To start the stabilization with an upright pole, a cable pull system is used. A 3-D-printed torus with a cylindrical base is placed on the pole, and a rope through the torus passes over the cable pull, allowing manual alignment. The rope is held until the LQR controller activates.

Figures 5.12a–5.12d show the 3 STL files for the 360-degree joint and the torus. Figures 5.13a and 5.13b display the full setup on the WAM with short and long poles upright.



(a) Stand for joint (b) Rotating x-axis (c) Rotating y-axis (d) Torus for rope

Figure 5.12.: Components of real 360-degree joint and rope hold

Free Standing Pole

For the free joint, we 3-D-print the Evetac case using files from [9], attach a silicon gel cartridge from the GelSight Mini at the top, and position the pole centrally on top before each iteration. The components and the setup can be seen in Figures 5.14 and 5.13c. A 3-D-printed add-on at the pole's base protects the gel and concentrates pressure, causing deeper, sharper deformation and more events, as discussed in 3.2.4. The add-on is shown in Figure 5.14d. As before, the rope holding the upright pole is released when the LQR controller starts.

5.2.2. LQR Parameter Search

Unfortunately, the parameters found in the MuJoCo simulation gridsearch did not result in satisfying performance for the real WAM setup, leading to fall-overs of the pole or velocity limit violations. An example of the LQR configuration performance on the real WAM is visualized in Figure 5.15. As you can see, not long after the beginning, the task fails, due to a sudden motion of the pole and thus the robot, which violates the velocity limits.

This mismatch in performance from simulation to real life likely occurs for mainly two reasons: First, in simulation we always have perfect access to all states, whereas in reality we can lack a complete pole state estimate and rely on approximations of all values. Second, there seems to be a mismatch between the URDF and the actual WAM, meaning our model of the robot and its dynamics are different from the real WAM. For example, we need a bigger force for holding the robot in a position than naively computed via

(a) Attached short pole (b) Attached long pole (c) Free long pole

Figure 5.13.: Setup examples with short and long pole

Pinocchio on the URDF. Both reasons lead to a poor performance of the parameters found in the simulation.

However, performing a gridsearch in real life on the LQR parameters to find the best values would be taking an infeasible amount of time, given the complexity of the gridsearch and the runtime per trial. Therefore, we limited our search to a manual inspection and improvement of parameter gains, so that the balancing works out, and omitted an exhaustive analysis of possible LQR parameter values. For this manual search, we go over different setups and provide a few example plots of the trajectories' evolution over time. All control loops were set to 20000 steps, with the usual aborting constraints on safety and pole tilt as mentioned in chapter 4. If a trajectory ends before 20000 steps in the plot, this means the balancing failed at that step due to any of the stop conditions.

(a) USB hull (b) Camera housing (c) Gel, from GelSight (d) Pole add-on

Figure 5.14.: Components of the free setup

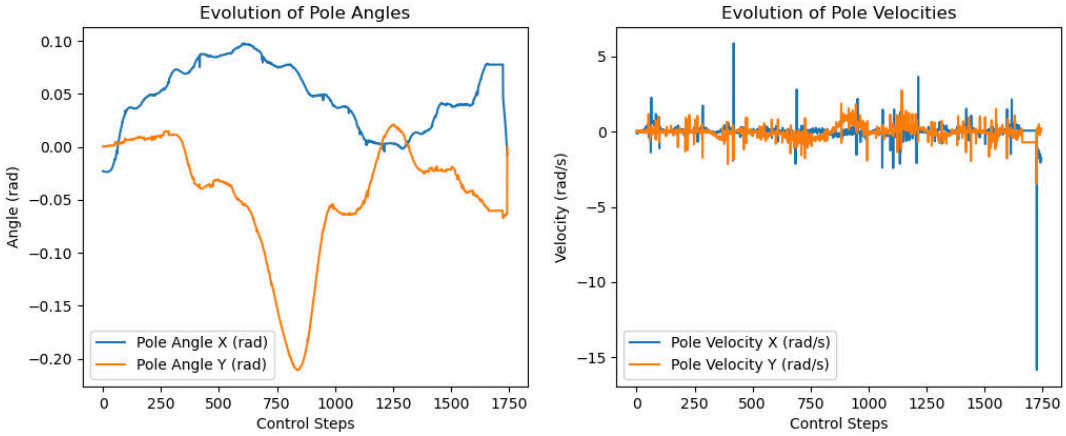


Figure 5.15.: Trajectory for real LQR parameter 100,1,1,10 for the short pole

Attached Pole

To make the real-life balancing as easy as possible, we start with the attached setup and initially choose the smaller pole to mimic the research in [25]. With this setup, we perform only a coarse search to inspect the behavior of the real WAM. Given these insights, we hope to find parameters more easily for the free pole setup, which is the desired setup for our tactile balancing task.

As a starting point for the search, we set all $\mathbf{Q}$ gains d_1, d_2, d_3, d_4 to 1. As can be seen in Figure 5.16, during balancing, the trajectory is interrupted, indicated by a stop of the line plot. This occurred due to a violation of the velocity limits of the robot and is reflected in

the plot by the excessive tilt and velocity values for the last values around step 14000. For this LQR configuration, the system becomes unstable during roll-out and experiences significant velocity and tilt values.

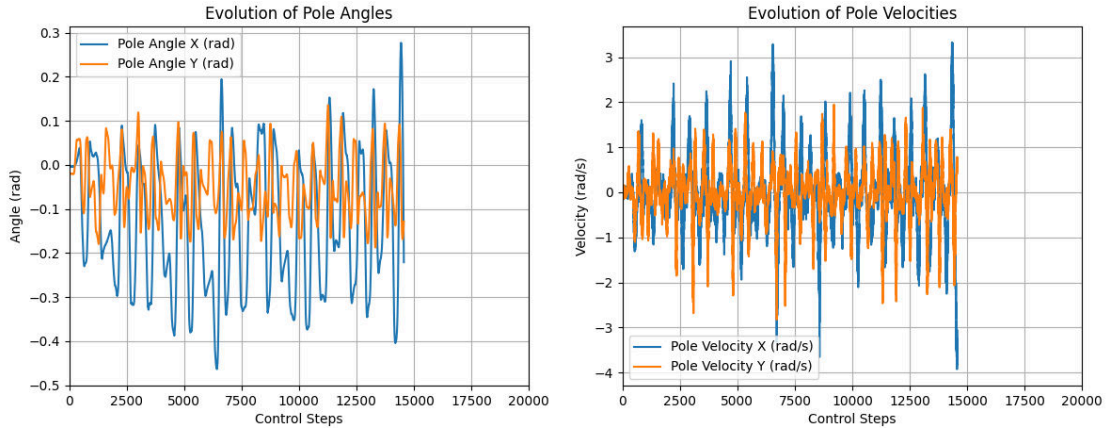


Figure 5.16.: Trajectory for real LQR parameter 1,1,1,1 for the short pole in the attached setup, with eventual violation of the velocity limits of the joints

To avoid these high velocities and excessive pole tilts while also staying close to $\mathbf{q_t}$, we increase the LQR parameters to 10. An example roll-out for this configuration is plotted in Figure 5.17. The trajectory no longer fails during roll-out as previously. Furthermore, the velocity and tilt angle values are significantly less extreme. However, there is still a lot of room for improvement for better stability. For example, the pole experiences an unproportional amount of tilt in the x-direction, and the velocities have sharp peaks.

To achieve more precise control, we set all LQR parameters to 100. As can be seen in Figure 5.18, the angle and velocity have less extreme values than before. The disproportionate tilt in the x-direction and the spikes in the velocity remain, and although they are less significant in terms of absolute values, the relative difference between x and y direction seems to be a persisting problem in the real setup. This may be related to the setup itself, meaning the pole is not starting in the ideal position ´facing upward, but with a bias with respect to the x-axis, due to position inaccuracies of the WAM and the cable pull, or how the LQR is capable of balancing the pole in this setup.

If we increase the parameter even further, e.g., to 1000 for all LQR parameters, these high controller gains perform slightly worse in terms of pole tilt as visualized in the roll-out in 5.19. The pole is tilting with more extreme values, especially around the x-axis. Since

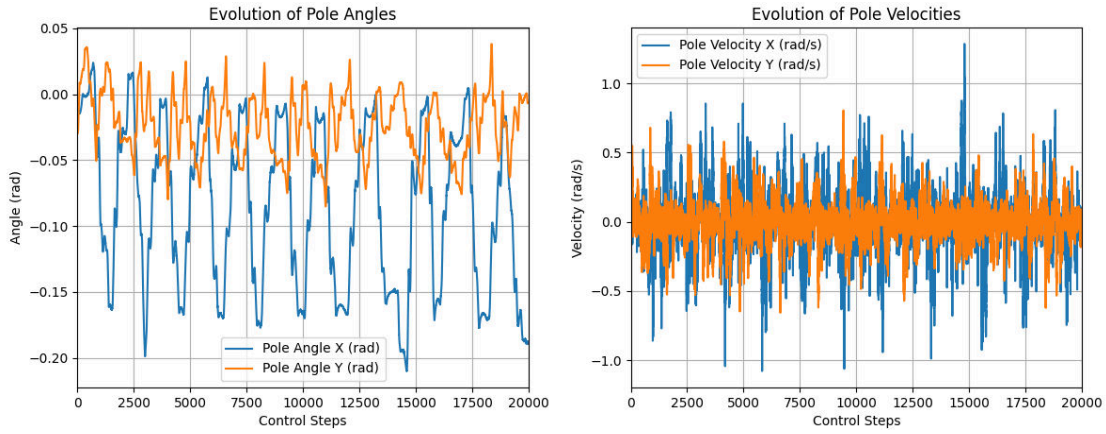


Figure 5.17.: Trajectory for real LQR parameter 10,10,10,10 for the short pole in the attached setup

large tilts are undesired for our free setup for the tactile sensor, where the pole could slip, increasing the parameter further seems undesired.

For the longer steel pole in the attached setting, the trajectories are quite similar to the short pole trajectories as displayed in Figure 5.20. According to our MuJoCo research on the different pole lengths 5.1.3, this validates that the pole angle estimate must be accurate, since the longer pole could also perform worse given noisy estimation of the pole angle.

Concluding, the best configurations for our tactile sensor setup seem to lie between 100 and 1000, as for values below 100, the balancing can fail or move unnecessarily much, and for values around 1000, our controller is becoming too stiff.

Free Pole

For the pole freely standing on the tactile sensor, we use the longer pole due to its more favorable balancing properties compared to the shorter one. In the free setup, the controller must ensure continuous contact, avoiding scenarios where the pole slips on the gel or falls off due to abrupt movements. Losing contact leads to inevitable control failure, as the robot cannot catch the pole once it fell. Furthermore we also lose any tactile information in this case which would make control impossible in the tactile scenario.

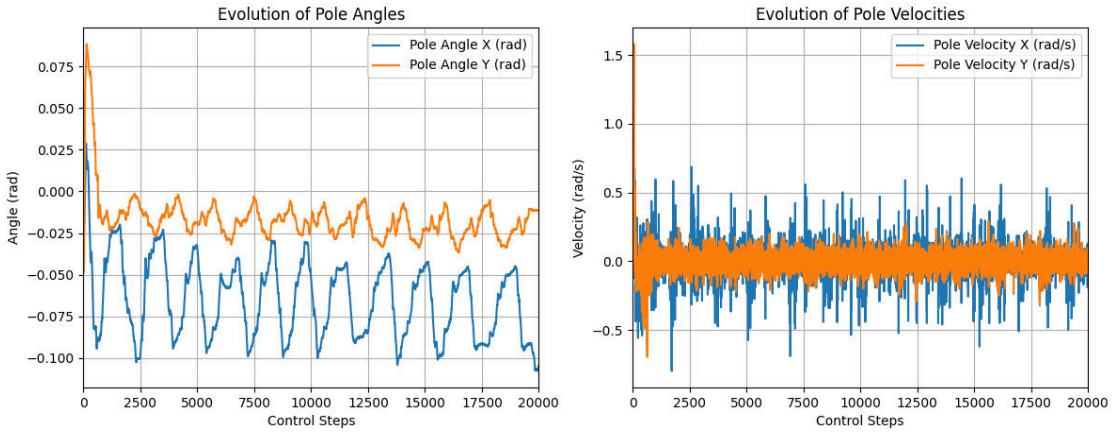


Figure 5.18.: Trajectory for real LQR parameter 100,100,100,100 for the short pole in the attached setup

However, as outlined in 3.2.4, we also want to avoid control parameters that cause the pole to barely move, because then next to no events are triggered. Therefore, we need to strike a balance between movement and stability, avoiding both overly aggressive and overly smooth trajectories. This enables effective learning of the mapping from tactile data to pole angle.

We evaluate a multitude of possible LQR configurations. We manually tune each parameter using the insights from the attached setup LQR search, and finally end up with 500,200,40,10 as parameters for d_1, d_2, d_3, d_4, which resulted in the best performance when evaluated. As depicted in Figure 5.21, this controller configuration ensures the robot and pole position is held quite strictly with high parameter gains acting on d_1 and d_2. Velocity is relatively encouraged through the lower gains acting on d_3 and d_4. The motion of the pole is necessary for event generation, which is why we punished velocity less.

These LQR parameter provide a good stabilization in real life, leading to no fall of the pole over a 5 minute horizon of balancing (longest tested horizon), while also ensuring movement of the pole for the tactile sensor. With this configuration we capture the datasets used for learning a mapping from tactile sensor data to pole angle.

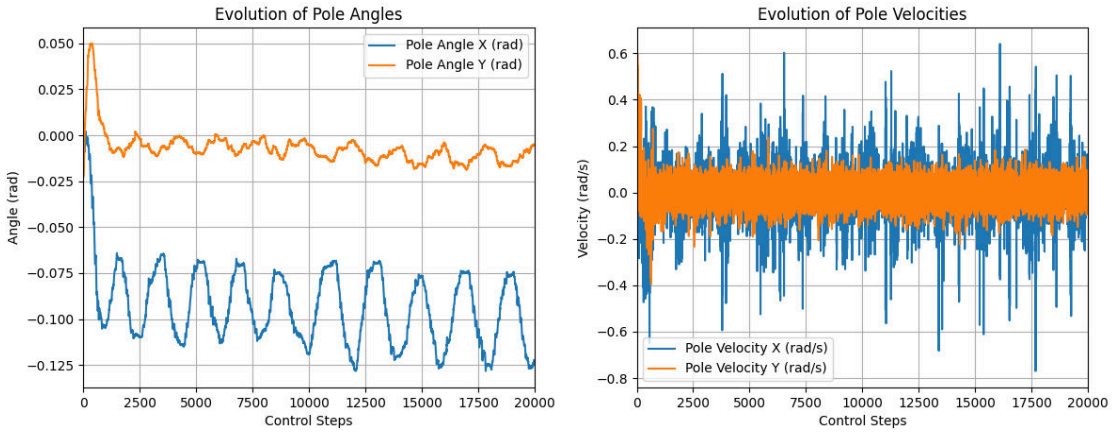


Figure 5.19.: Trajectory for real LQR parameter 1000,1000,1000,1000 for the short pole in the attached setup

5.3. Tactile Based Pole State Estimation

Our goal is to replace the OptiTrack estimate of the pole's global tilt vector with the tactile sensors' observations, precisely the estimated angle from the tactile events, predicted by a neural network. For this we try to learn a mapping in a supervised fashion instead of e.g. reinforcement learning techniques for two reasons: First, for reinforcement learning we would need constant interaction with the real system, since the simulated environment does not map to the real WAM, as outlined prior, making simulation roll-outs and their results not usable for the real WAM. Furthermore, in simulation, the tactile sensor does not exist, so simulation is not an option.

Second, continuously rolling out a policy on the WAM would take a long time, and it is unclear how many samples we need to fully cover the space, especially because for a fixed LQR configuration, the trajectories of different roll-outs are quite similar, limiting exploration. Therefore, we choose the supervised learning approach as it provides a simple, straightforward framework for learning by capturing datasets and regressing the angles based on the data.

In the following we go over each of our individual approach to learn a mapping from tactile sensor data. We highlight the individual structures of the network, the training procedure, and present the results, which include the loss evolution over time and examples of

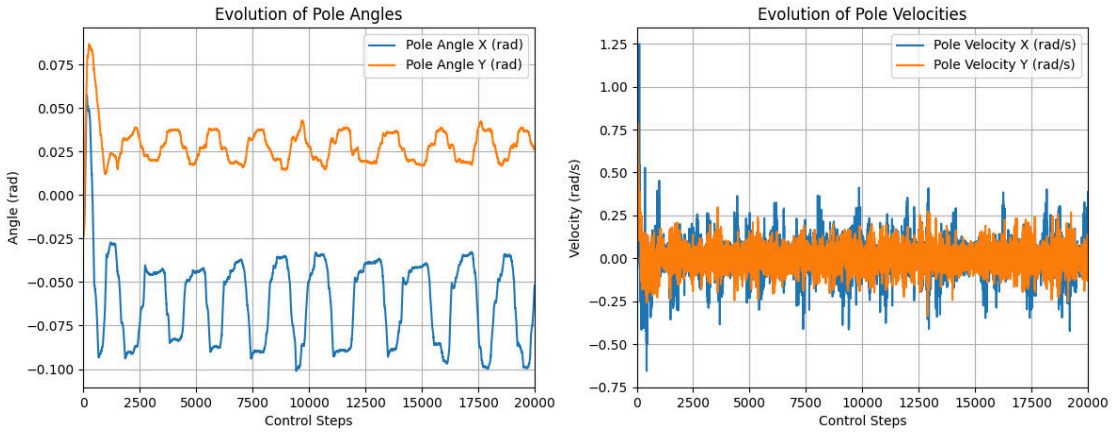


Figure 5.20.: Trajectory for real LQR parameter 100,100,100,100 for the long pole in the attached setup

inference performance. At the end of the section we provide a summarizing table to compare the individual approaches.

5.3.1. Setup

For learning any mapping, we used an initial learning rate of 0.0001, which gets reduced every 50 epochs by a decay factor of 0.5. If not specified otherwise, we choose a batch-size of 128, striking a balance between memory management and the size of each batch. For optimization, we use the Adam[66] algorithm, regulated with a weight decay of 0.0001. For the regression, we train for a total of 200 epochs on the angles, using the MSE of the prediction and the actual value. For the classification task, we train for only 50 epochs using the CE loss. Other hyper-parameters are dependent on the individual network architecture and are mentioned in the respective sections. Remember that each network is trained 5 times on different train/test splits of the data according to a 5-fold cross-validation. The results are based on the mean of these runs, if not otherwise specified.

We either shuffle the data explicitly when loading into RAM or we shuffle the content of each loaded file when loading from disk, depending on the dataset. For regression, the angles are normalized based on the min and max values found in the respective dataset, pushing all values in the range $[-1, 1]$. For classification during the dataset during training

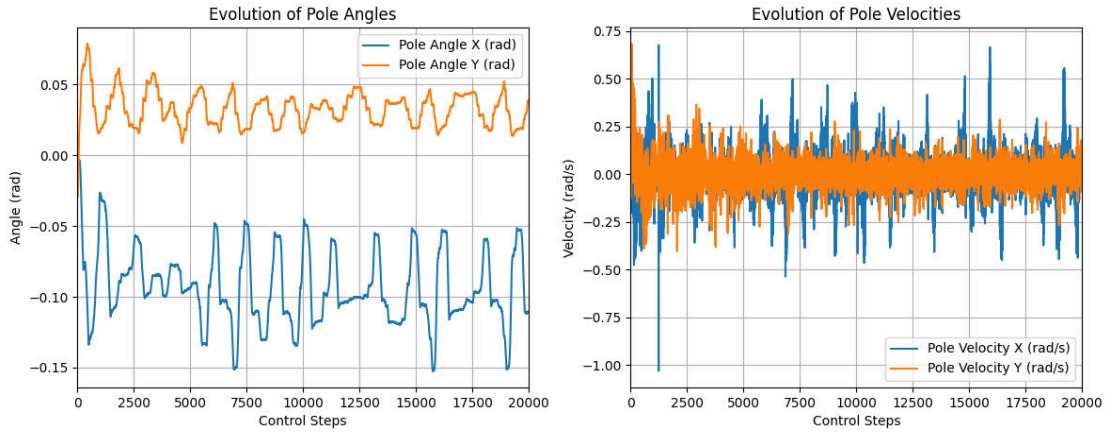


Figure 5.21.: Trajectory for real LQR parameter 500,200,40,10 for the long pole in the free setup

is balanced by super-sampling the individual images so that we have an equal number of images per class.

5.3.2. Classification

As an introduction and to demonstrate the potential of learning from tactile sensor data, we begin by focusing on classifying the fall direction in the classification dataset. This serves as a simple entry point for two reasons:

1. The task of classifying fall direction simplifies the problem, as it reduces the challenge of estimating the pole's angle from continuous values ($\mathbb{R}$) to discrete class labels ($\mathbb{N}$).
2. The classification dataset is smaller than the regression datasets, making the task more manageable and focused.

In this dataset, the pole falls freely while the WAM is instructed to maintain its position using a gravity compensation term. This setup ensures that the trajectories and events are independent of any robot motion and primarily occur in the region where the pole is tilting.

However, since we cannot infer the pole state from a fall direction label, we only demonstrate that learning from the sensor data is feasible, i.e., there is meaningful data in the

tactile sensors' observations. We train a CNN, with its architecture detailed in Section 5.3.4, as we perform a more detailed analysis there, on images as created in 1. The network's output dimension is set to match the number of fall direction classes, and we use Cross Entropy Loss for training.

As a simple test, we split the directions around the pole into 4 quadrants and a region around 0,0. Together, they form the 5 classes we train to classify. An image of this setup and the ground truth label for the trajectories can be seen in Figure 4.6. We train for 50 epochs and showcase that it is possible to learn, as can be seen in the example confusion matrices displayed in Figure 5.22.

The CNN manages to learn from the training set and achieves good prediction performance for 4 out of the 5 classes that appeared in the evaluation trajectories (a single fall trajectory is typically in a single class as visualized in Figure 4.6). While the classification is imperfect,

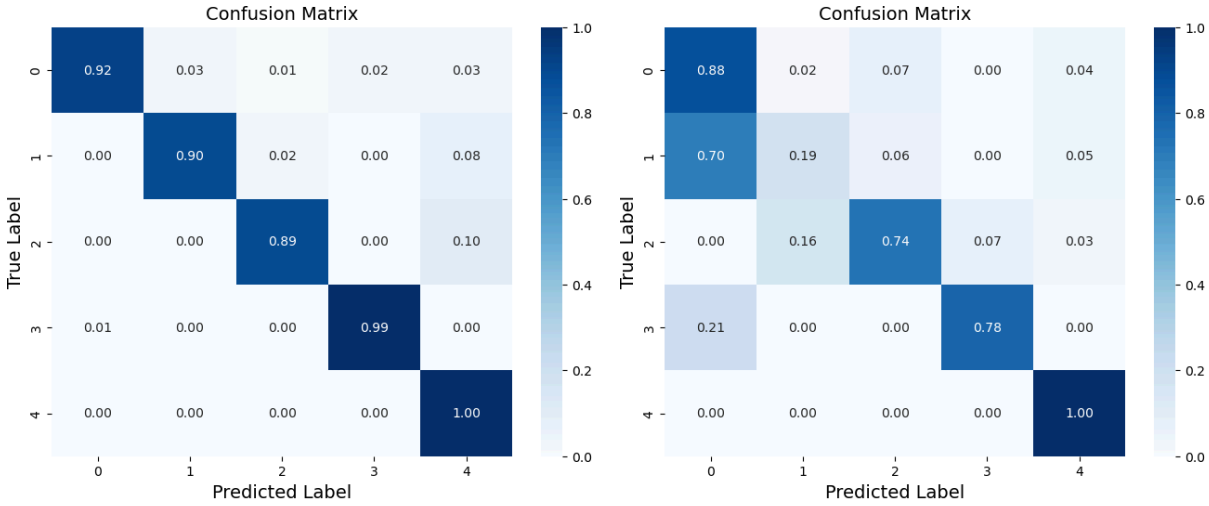


Figure 5.22.: Single Split Example Confusion Matrices of a CNN trained to classify the fall direction based on the tactile observations. Training (Left), Evaluation (Right)

this confirms that the tactile sensor can provide enough information for learning purposes and deciding the fall direction of a pole.

5.3.3. MLP Regression

Moving on from classification, we focus on the original problem, regressing the pole angle from the tactile data. Initially, as a simple start, we used an MLP as a neural network for regression. Since in [9] the authors were able to regress forces applied to the gel based on the tracked dots with a simple MLP alone, the architecture seems reasonable. We use a 4-layer MLP with a hidden-size of 256, resulting in 65.792 parameters, which is slightly bigger than the one used in the Evetac paper. Each layer of our network consists of a combination of a linear layer, an activation function, and a dropout of 0.1 for regularization.

Dots

First, we try to regress the pole angle based on the dots given in the first dataset. For the input, we flatten the 63 dot locations (x and y coordinates) to a single 126-dimensional vector. A visualization of this network can be seen in Figure 5.23.

Figure 5.23.: Dots network

We compute the mean of each dot's location on the dataset and feed the neural network the relative positions of the dots to their mean positions. As you can see in Figure 5.24, the regression is not precise. Seemingly, the dot locations do not sufficiently move around during roll-outs, as the network predicts simple flat lines at times, corresponding to similar input. This is likely caused by the pole being too lightweight, which applies insufficient pressure on the gel.

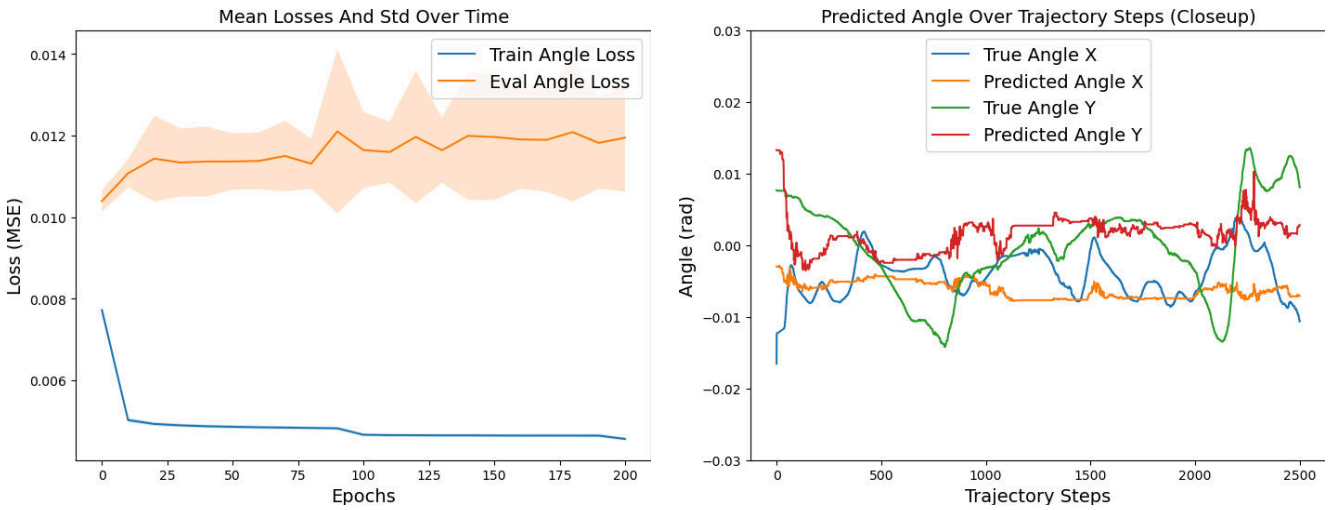


Figure 5.24.: Loss evolution (left) and predicted vs. true values (right) for network MLP using Dots

Dots And Robot Data

As shown in the previous section, the model fails to accurately learn and generalize pole angle regression from the dot locations. To address this, we augment the input with the robot's position and velocity (robot data) at each time step and introduce an additional 4-layer MLP branch to process this data, which is visualized in Figure 5.25.

Afterwards, the feature vector from the dot locations and the feature vector from the robot angles and velocity are combined into a single feature vector, which is regressed to the pole angle. As visible in Figure 5.26, the MLP is capable of predicting the angles relatively closely now, following the true angles more closely, although the evaluation loss is not improving during training.

Robot Data

To assess the effect of the robot data on learning, we also train a Multi-Layer Perceptron (MLP) that only processes the robot data and predicts the pole angle. If the robot data alone is adequate for prediction, we should avoid conditioning any network on it, as doing so may lead to the tactile input being disregarded. As can be seen in Figure 5.27, the

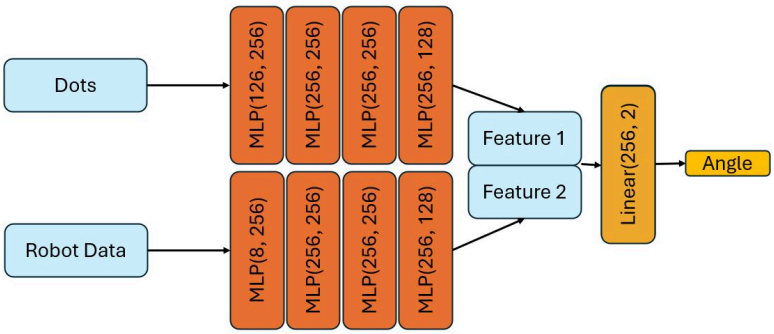


Figure 5.25.: Dots and robot data network

robot data alone suffices for learning the angles. The estimate gets worse if we use the prior structure, which includes the dots.

Judging these results, we conclude two things. First, the dot locations are not sufficient for estimating the pole's angle. Second, the network exploits the correlation between robot state and pole angle that likely emerges from our LQR formulation, when conditioned on the robot data. This learned correlation (likely) never works on the real system, as the robot should react to the pole tilts, changing the robot's state based on the control law. However, when learning from the robot data, we learn the inverse relation, what robot state corresponds to what pole state, which is not usable for later control.

5.3.4. CNN Regression

After failing to regress the angle on the dots with a simple MLP, we turn to a novel approach, utilizing CNNs. As highlighted in section 2, several works such as [45] use CNNs to interpret the images from optical tactile sensors. Following this line of research, we also try to use CNNs for our problem, but for our sensor, we first need to generate the images from the events we capture. For each set of events recorded at a time-step t, we create an image as described in 1. We do this for all 10 trajectories we captured on the second dataset. We choose to keep the dataset limited to that size (around 200000 data-points consisting of image and pole angle) due to the limited time scope of the thesis,

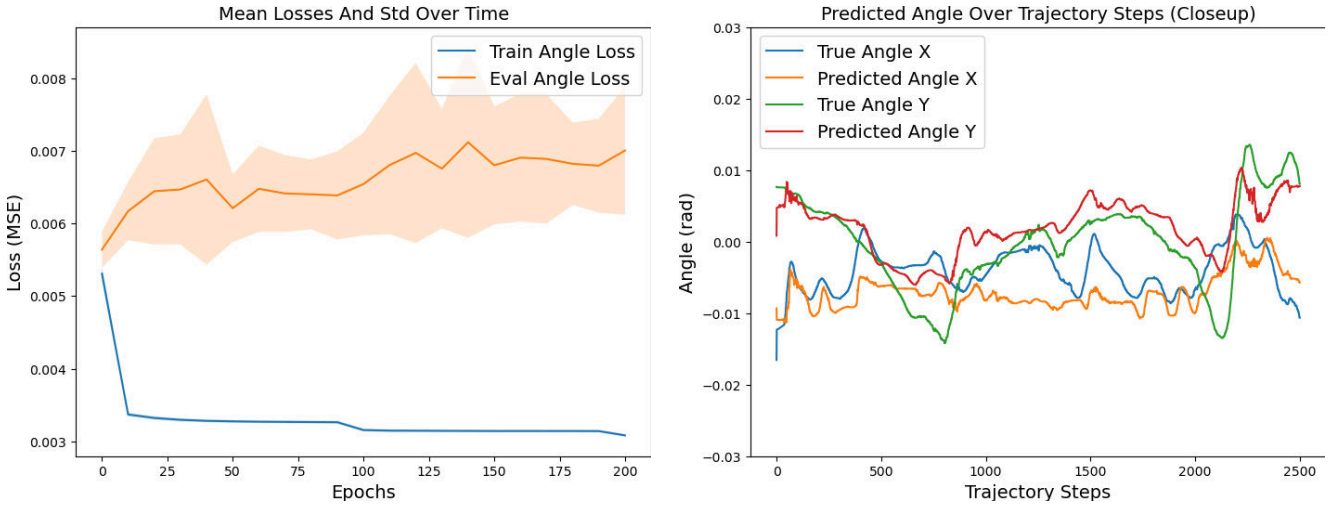


Figure 5.26.: Loss evolution (left) and predicted vs. true values (right) for network MLP using Dots And Robot

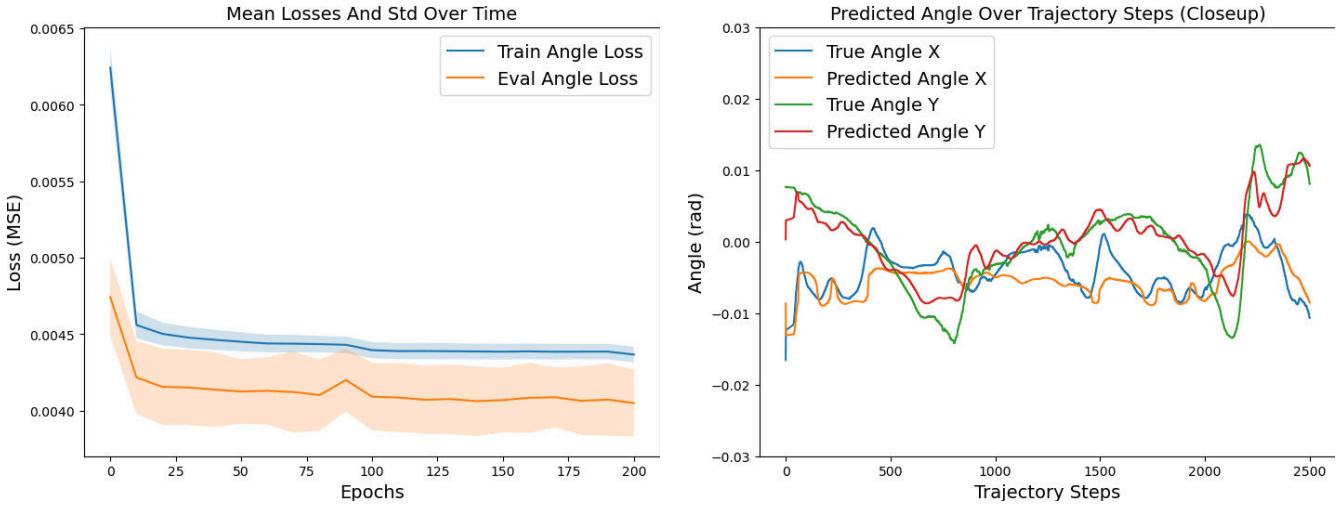


Figure 5.27.: Loss evolution (left) and predicted vs. true values (right) for network MLP using Robot

which became a bottleneck for tuning the CNN architecture, because they train much longer than the previous MLP structures.

For our CNN architecture, the network consists of multiple CNN-layers, which in turn consist of a convolution, a ReLU, a MaxPool, and a layer norm. We evaluate the usage of an additional dropout layer to regularize the network. The MaxPool reduces the dim of the images per layer, the ReLU helps to learn non-linear functions, and the convolution is used to interpret the image values. The inside of each CL can be seen in Figure 5.28a. An example of such a 4-layer-CNN can be seen in Figure 5.28b. Note that in these images, we include the dropout layer although we reason about it in the next section. Furthermor,e it is important to know that we double the channels per CL layer, which makes the input size of the final linear layer dependent on the number of CL in the CNN.

(a) 4-CNN layer components with dropout.

(b) Example of 4-layer CNN

Figure 5.28.: Visualizations of CNN architectures.

We initially chose a moderately sized 4-layer CNN with a total of 389,602 parameters, including a regression head for predicting the pole angle. This architecture served as a solid starting point for our experiments, allowing us to evaluate performance and tune hyper-parameters, such as kernel-size, dropout, feature aggregation and image creation parameters effectively. We also report results using alternative network sizes to inspect the trade-offs between model complexity and performance.

As discussed in Section 4.6.1, the original images are down-sampled by a factor of 5 to reduce data size while preserving important visual details. This results in a reduced input resolution of 128×96, which strikes a balance between efficiency and information retention. The down-sampled image is then processed through our 4-layer CNN, which produces a feature map of size $C \times 16 \times 12$, where C is the number of output channels. We start with 32 channels in the first layer and double the number of channels in each subsequent layer. This progressive increase allows the network to capture more complex features at deeper levels.

Effect of Introducing Dropout

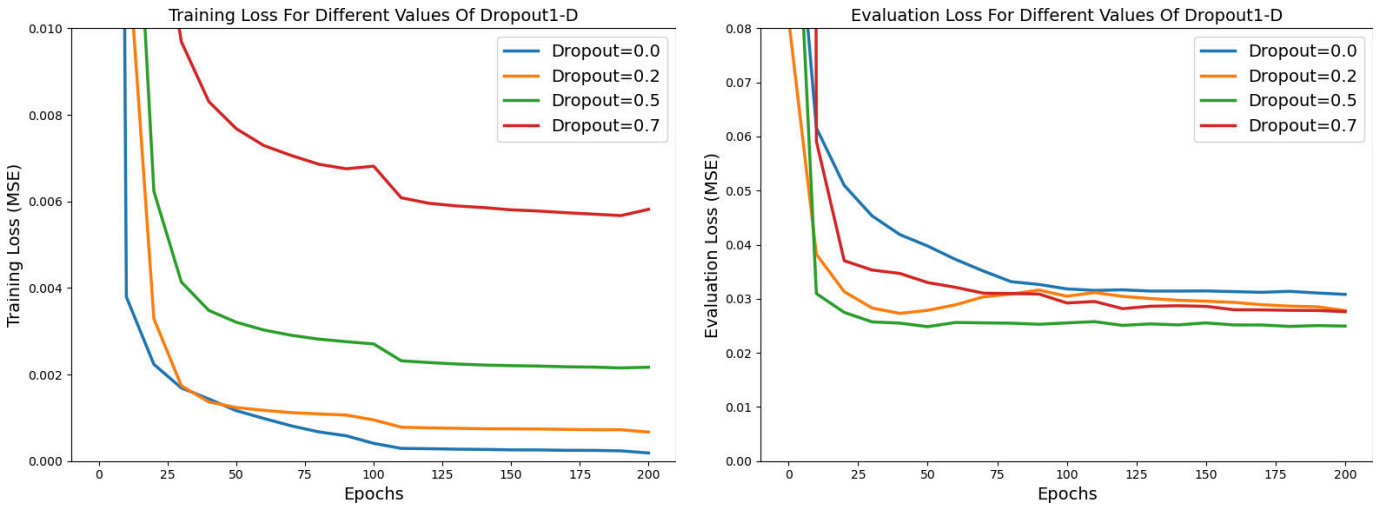


Figure 5.29.: Effect of 1D-dropout on the loss. Training (Left), evaluation (Right)

As can be seen in Figure 5.29, our 4-layer CNN is capable of reducing the training loss to near zero (blue line). This is a big improvement over the performance of the MLPs in the prior section, working on both the robot data and dots, which only achieved a training loss of around 0.003 (as seen in the previous section). Therefore, this image-based approach and the CNN architecture seem to be promising.

However, the network fails to reduce the evaluation loss equally well, being stuck on a loss of around 0.03 (see figure 5.29). This over-fitting to the training data can occur due to several reasons, such as insufficient regularization, a small dataset consisting of insufficient samples for learning a general function, or an overly complex network structure. First we focus on regularization. Dropout is a common method used to reduce over-fitting in machine learning. We examine the effect of 1-D- and 2-D-dropout, i.e. dropout on the individual pixels or on entire feature maps. For the 1-D-dropout, we test the values of $0.0, 0.2, 0.5, 0.7$. The respective evaluation loss results are depicted in Figure 5.29 and compared to the baseline of 0 dropout.

For the 2-D-dropout, we considered smaller values, namely $0.0, 0.1, 0.2, 0.3$. The results are depicted in Figure 5.30 and again compared to the baseline with 0 dropout.

As can be seen, the dropout has a mild effect on reducing the loss by approximately 15 percent for 0.5 on the 1-D-dropout and for 0.1 on the 2-D-dropout, but the issue of

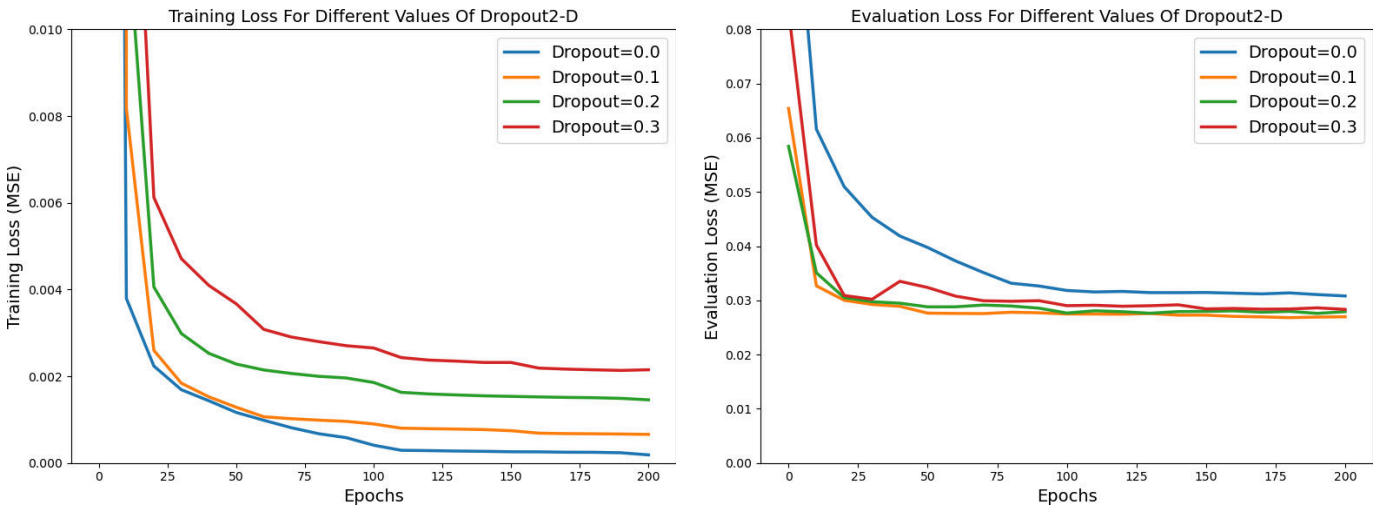


Figure 5.30.: Effect of 2D-dropout on the loss. Training (Left), evaluation (Right)

over-fitting remains, since the training loss is order of magnitude lower than the evaluation loss. Since 2-D-dropout is beneficial for performance and generally deemed more suitable for image processing than the 1-D variant, we use a 2-D-dropout of 0.1 for the CNN in the following.

Effect of Event-gains δ and Decay-Rates γ

Another option for improving the learning in general is to try different values for the event-gains δ and decay-rates γ in hopes of finding better images for learning the angles. To qualify the influence of δ and γ we perform a small grid-search over them, comparing the evaluation losses on the respective image datasets for the same CNN architecture.

For our search, we consider $\delta = 0.1, 0.5, 1$ and $\gamma = 0.999, 0.99, 0.9$. This offers a variety of combinations, from forgetting past events rather fast with a decay-rate of 0.9 to a longer memory with a decay-rate of 0.999. Also, the sensitivity to new events with 0.1 is much lower than for 1, where any event causes the maximum image value. We mix any pair of δ and γ, create the respective dataset, and train our CNN architecture on it.

The different evaluation losses are depicted in Figure 5.31. As you can see, the main difference between the datasets is caused by the decay-rate γ, as for 0.9, irrespective of the δ, all CNNs have a high evaluation loss. For $\gamma = 0.99$ and $\gamma = 0.999$, the evaluation

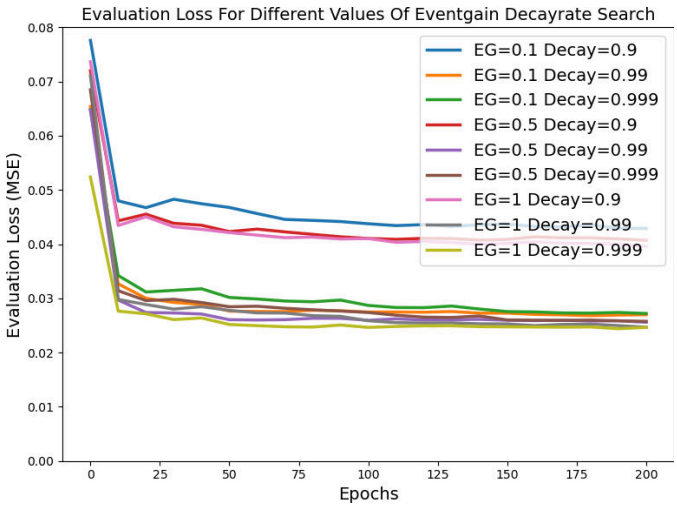


Figure 5.31.: Effect of learning on images generated by different event-gains δ and decay-rates γ

loss is lower, which means remembering older events seems to be important for predicting the angle. For 0.99 it takes around 450 time-steps until an image pixel is approximately reset to its original value ($0.99^{450} = 0.0108$), which corresponds to approximately 0.9 seconds in real life. For 0.999 it takes around 4500 time-steps ($0.999^{4500} = 0.01108$), which corresponds to 9 seconds in real life.

Although close, the best performance is achieved for images created with $\delta = 1$ and $\gamma = 0.999$, meaning a high sensitivity to new events as well as remembering older ones for a longer time horizon seems to be beneficial for predicting the angle. This result can be explained by the fact that we need a piece of global information for the absolute value and sensitivity to recent events to react to fast changes.

Effect of Feature Aggregation Techniques

To aggregate the feature maps produced by the CNN we can use different techniques. Previously, we relied on global pooling, taking the maximum value of each feature map. However, we could also take the average or just flatten the feature maps. Especially the last operation results in a very large vector, but it preserves all the information gathered

in the feature maps. To inspect the impact of the different feature aggregation techniques, we compare the three in terms of evaluation loss.

As can be seen in Figure 5.32, using the average pooling leads to a stagnation of the evaluation loss during training. For flattening and max pooling, the evaluation loss decreases initially but then fails to continuously decrease, as seen before. Max pooling yields a slightly lower evaluation loss (0.024 vs. 0.027) compared to flattening, despite using a smaller network with lower-dimensional aggregated features. This better result may be because in the training trajectories, the pole position on the gel is not constant, leading to different positions of the events in terms of image location. Since the max pooling is position-agnostic compared to flattening the final feature maps, the max pooling seems to be the superior technique, as solidified by the results of our trials. Therefore, we choose the max pooling.

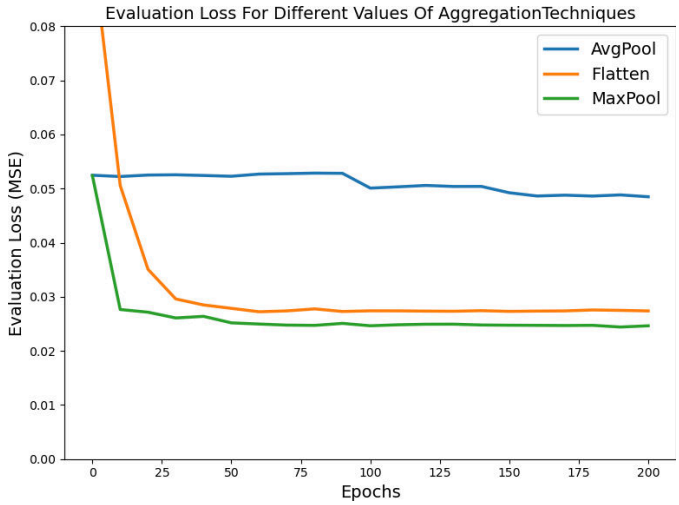


Figure 5.32.: Influence of different aggregation techniques on the evaluation loss

Effect of Kernel Sizes

Another hyper-parameter of the CNN that heavily influences performance is the kernel-size of each layer. The kernel-size determines the context of each pixel when using a convolution. For sufficient edge detection, it might be necessary to consider bigger

kernel sizes. We test different kernel sizes on the 4-layer CNN and provide the resulting evaluation loss, depicted in Figure 5.33.

As you can see the different kernel sizes hardly influence the final performance on the evaluation set, as all variants achieve similar results. Therefore, we decide to use a 3×3 convolution as this has the fewest parameters.

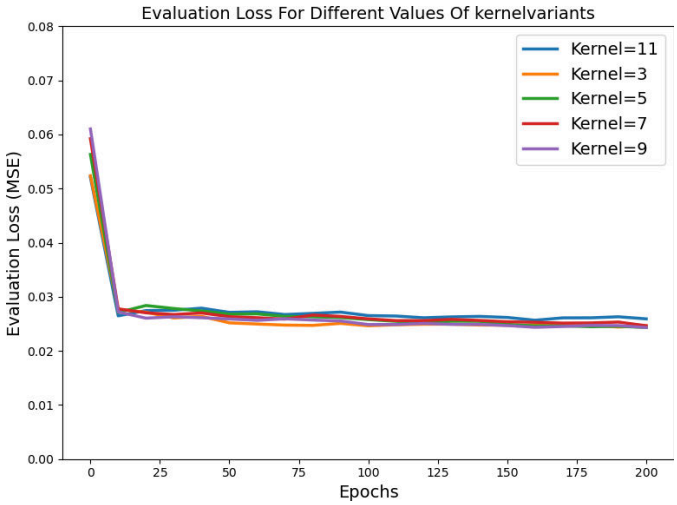


Figure 5.33.: Influence of different kernel-sizes on the evaluation loss

Effect of Number of Layers

Our network might be too large or too small for the dataset. We test the performance of different network sizes, going from 1 layer to 7, using a convolution of 3×3. 7 is the maximum amount of layers we considered, because for 7 layers the final feature map already has the dimensions of 2×1. We compare the individual network performances in Figure 5.34.

As you can see, the network depth influences the evaluation set performance. We see significant decreases in loss on the evaluation setup until the 4-layer CNN, but for more layers, the benefit is stagnating, negligible, or the network performs worse. This performance may be because, for bigger networks, the final feature maps are small, as explained above. Therefore, the 4-layer CNN seems to be the best choice, as it has fewer parameters

than bigger networks, while performing on par, and for fewer layers, we get worse results.

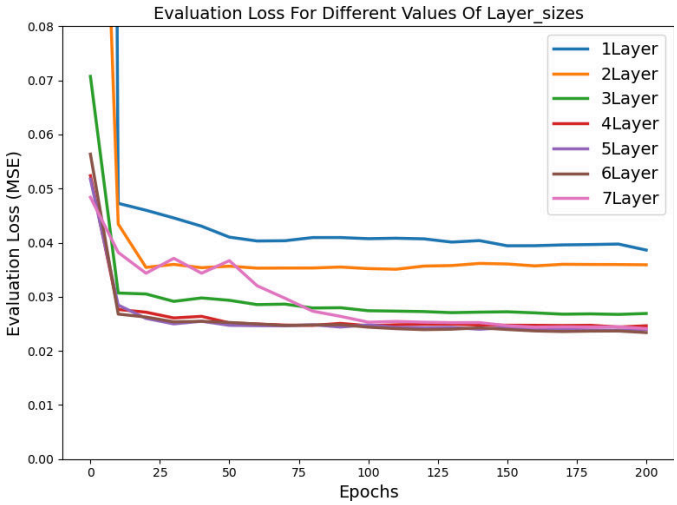


Figure 5.34.: Influence of network depth on performance on the evaluation loss

Effect of Stacking Multiple Images of Different γ and δ

Multiple images with different decay-rates and event-gains may allow the CNN to learn different time dependencies in the data. To test this, we compare the evaluation loss of the CNN trained with a single pair of images created by $\delta = 1$ and $\gamma = 0.999$ and images consisting of 4 pairs for each combination of $\delta = 1, 0.1$ and $\gamma = 0.999, 0.9$. These values were chosen as they represent the respective maximum and minimum values we tested, spanning the biggest difference between the images.

However, as can be seen in Figure 5.35, stacking multiple images is not beneficial to learning. This may be related to what we showed in Figure 5.31, where some combinations of δ and γ result in images we can learn on better, while others seem to lack the information we need, and combining images does not change that.

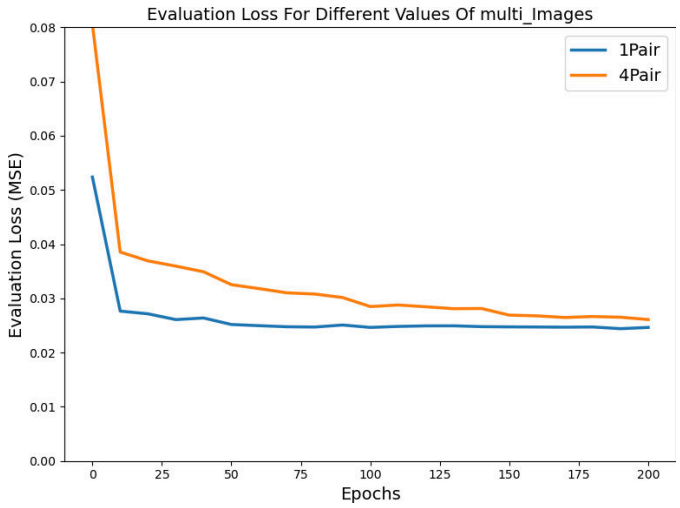


Figure 5.35.: Influence of stacking multiple images of different δ and γ on the evaluation loss

Effect of Shifting Images during Training

As mentioned, we place the pole manually on the gel of the tactile sensor, which is why the pole is not necessarily in the center of the tactile sensor. Thus, the actual event location of corresponding parts of the pole varies between trajectories. To make the CNN more robust to these circumstances and to focus on the shapes in the images and not the exact location of the events, we shifted the images during training randomly in the x and y directions and tested the effect of different absolute values for the random shift. The shifts themselves are uniformly distributed, and we test different ranges.

As can be seen in Figure 5.36, adding a shift during training lowers the evaluation loss quite significantly, from around 0.024 to around 0.018. We achieve a better performance for a smaller shift range of 10 pixels in either direction than for 20 pixels in either direction. The reason for a better performance of a smaller shift may be because the down-sampled image we shift is only 128×96, so a shift of 20 is very significant, leading to possibly shifting the pole events out of the image, ultimately losing data.

Amending the dataset by shifting the images shows two other things: First it showcases that the network needs to learn that the pole position in the trajectories during training is not always centric, due to slip or non-centric initialization, and thus it is beneficial for

evaluation to be position-agnostic to a certain degree. This also coincides with the results from the aggregation technique inspection results for max-pooling.

Second, shifting the images and achieving better results likely indicates that our dataset is not large or diverse enough. Given that we only captured 10 trajectories and that for a single trajectory its data-points are not independent, this likely harms training a generalizing neural network significantly.

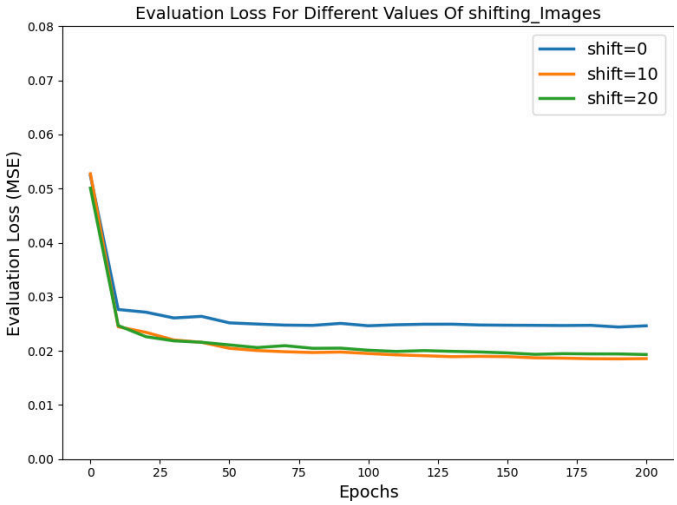


Figure 5.36.: Influence of shifting images during training on the evaluation loss

A final example of the prediction performance for this CNN can be seen in Figure 5.37. As you can see, the CNN can be capable of following the true trajectory, but oftentimes it does not fit the peaks but rather outputs a mean regression value, especially for the x-axis angle, which is prevalent and likely detrimental to using the output for control.

5.3.5. LSTM Regression

In the CNN-predictions (see figure 5.37) we can see that the network fails to reliably follow the curve of the angles, especially for peaks. This may be related to the fact that for a single image, it is hard to differentiate what exactly the pole angle might be, because of the similarities of individual pictures. Given more context of the image, i.e., a sequence of images leading up to the final one, and a subsequent prediction, might be a straightforward solution to improve the angle prediction. To process such a sequence, we use an LSTM.

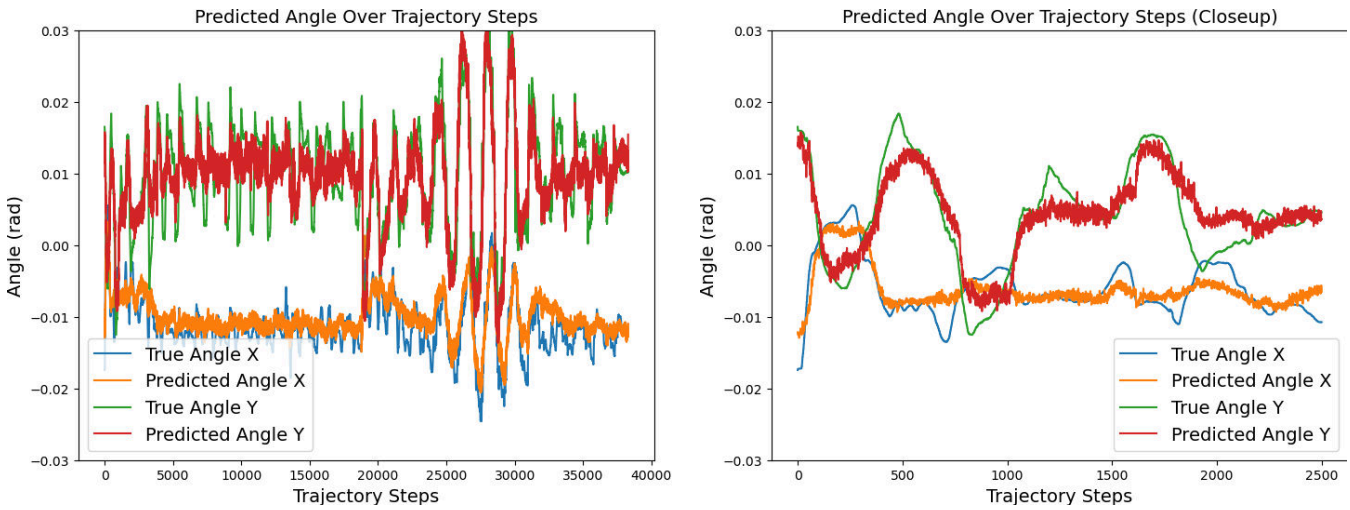


Figure 5.37.: Example of the prediction performance on the evaluation set of the CNN trained on shifted images with dropout

An LSTM is capable of learning from a series and predicting its continuation. Since our trajectories are an example of such a series, we try to learn the angles with the help of a LSTM, processing the features of a CNN that interprets the image data like in the previous step. For the LSTM, we choose a single layer, with a hidden-size of 256, determined by the channels of our previously tuned 4-layer-CNN output and learn on different sequence lengths. the architecture can be seen in Figure 5.38, where the 4-layer CNN process the input images, the LSTM takes the respective sequence of feature vectors and finally an angle is predicted.

Differently to the previous CNN training we need to lower the batch-size from 128 down to 64 due to memory concerns of the GPU . Furthermore, due to timing constraints we train only for 100 epochs, because each epoch now takes more than 700 seconds a single cross validation run is taking up to 4 days.

Sequence Length 10 for $\gamma = 0$ and $\delta = 1$

We initially choose $\gamma = 0$ and $\delta = 1$ so that we only consider the recent events of each time-step per image. This means that the LSTM needs to learn how to combine the individual events in each image instead of us providing the dependency via a $\gamma > 0$. However, this

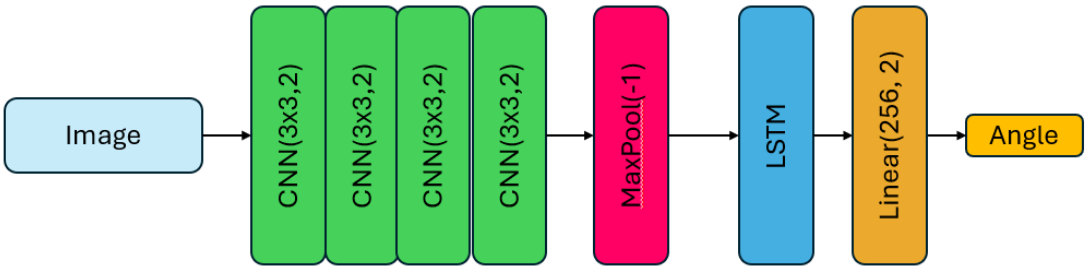


Figure 5.38.: LSTM architecture, the 4-layer CNN process the input images, the LSTM takes the respective sequence of feature vectors and finally an angle is predicted

processing step completely fails to learn a good model on the dataset for a sequence length of 10, as can be seen in Figure 5.39. Initially, the training loss is decreasing to around 0.032, but over time during training, the loss increases to a level comparable to the start. Similarly, the evaluation loss initially decreases but stagnates during learning at around 0.052. Both these values are the worst recorded so far.

This issue in performance may be caused by the CNN being unable to extract meaningful features from the images with $\gamma = 0$ and $\delta = 1$, thus hindering learning. As seen in Figure 5.31, a lower γ leads to a worse prediction. Therefore, it is reasonable to assume that for $\gamma = 0$ the CNN performance would be worse than for $\gamma = 0.9$. Another reason could be that the sequence of images is too short to gather meaningful knowledge with the LSTM. To counteract this issue, we would need to increase the sequence length.

Sequence Length 20 for $\gamma = 0$ and $\delta = 1$

To investigate whether the performance improves if we learn on a longer sequence of length 20, we train a respective LSTM. As can be seen in Figure 5.40, the longer sequence achieves a worse training loss value than for sequence length 10, as the training loss is increasing over the training period, ending around 0.042. The evaluation loss is decreasing slightly during training, but the error is still high, with 0.052 at the end. Furthermore, for the evaluation loss between the different runs of the 5-fold, there is a very high standard deviation.

These results suggest that the issue in learning is not due to insufficient sequence length, but rather that the features produced by the CNN fail to capture the necessary information,

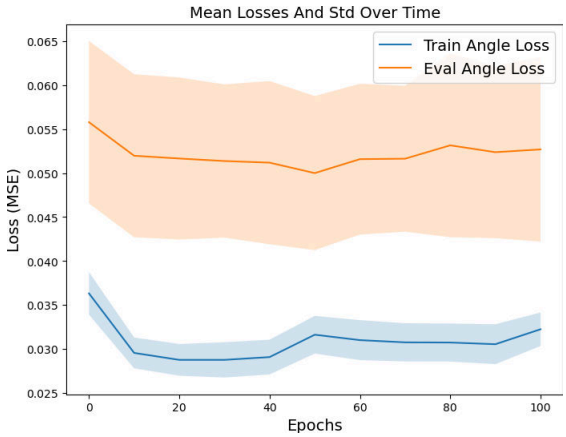


Figure 5.39.: Training and evaluation loss evolution over time for the LSTM with $\gamma = 0$ and $\delta = 1$ for a sequence length of 10

making it hard or impossible for the LSTM to learn.

Sequence Length 10 for $\gamma = 0.999$ and $\delta = 1$

Lastly, we investigate the performance when we create images with the previously best configuration of $\gamma = 0.999$ and $\delta = 1$ for CNN-based regression. As you can see in Figure 5.41, both the training and evaluation losses are much lower than for $\gamma = 0$. However, the evaluation loss of 0.03 is still higher than for the simple CNN setup, which quickly drops to around 0.025.

This higher error may be related to the fact that LSTMs are harder to train than CNNs, as LSTMs can suffer from a vanishing gradient problem. This is reflected in the higher training loss of the LSTM. Furthermore, the LSTM is also trained less than the CNN variants, although the CNNs do only experience a slight improvement for training longer than 100 epochs. However, the LSTM has more parameters and more complex inputs, which may require training for more than 100 or even 200 epochs to achieve better results. In Figure 5.41, you can see that both the evaluation and training loss are still significantly decreasing with more training epochs, but given the current progress per step, it might take another 100 epochs to reach the losses seen in Figure 5.36. As mentioned, LSTM evaluations were limited to 100 epochs due to time constraints.

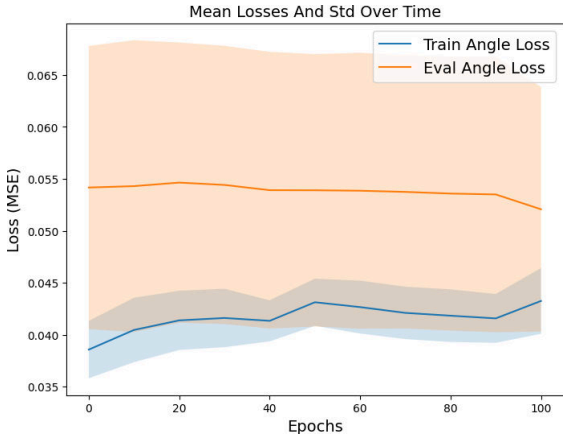


Figure 5.40.: Training and evaluation loss evolution over time for the LSTM with $\gamma = 0$ and $\delta = 1$ for a sequence length of 20

5.3.6. PointNet

Another, completely unrelated approach to processing the events is to interpret the event data at time step t as a point cloud. We process this pointcloud with PointNet into a global vector representing the shape as outlined in [59]. In [59] they sampled pointclouds of size 1024 from the surface of an object to classify and segment the object.

In our scenario, there are many fewer events, also heavily depending on the pole movement, which can result in 10 events or fewer, whereas timestamps with high movement can have up to 300 or more events. A histogram approximating the distribution of the number of events per time-steps taken from a single trajectory of the image dataset is displayed in Figure 5.42. Notice that the histogram is limited to the range of $[0, 50]$, since higher event counts occur rarely.

We evaluate the performance of a simple PointNet adaptation seen in Figure 5.43 on a cloud-size of 50, randomly sub-sampling bigger clouds and adding multiple points if there are too few points in a cloud. A cloud consists of all event locations captured at the time-step, normalized around the center of the image to be in [-1,1]. The events are taken from the image dataset.

As you can see in Figure 5.44, the PointNet-based regression fails to be accurate on both the train and test sets, which likely means the representation is insufficiently capturing the data or the network is not capable of learning the problem. This may be because the

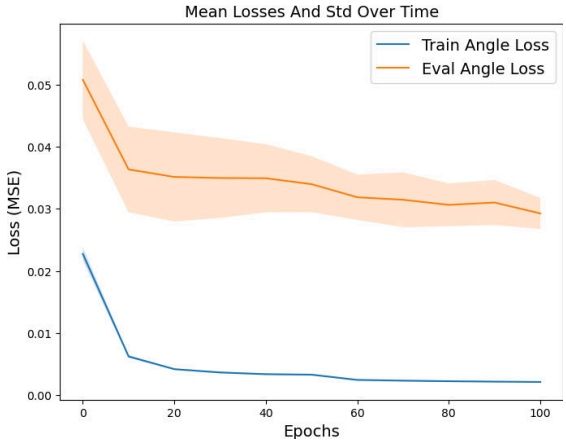


Figure 5.41.: Training and evaluation loss evolution over time for the LSTM with $\gamma = 0.999$ and $\delta = 1$

pointcloud size is magnitudes of orders lower than for which the original architecture was designed (1024 points vs. less than 10 in the mean, see figure 5.42). Judging the results, we conclude that this architecture is not suitable for the problem.

5.3.7. Real Roll-Outs

Before testing out our trained neural networks, we investigate the performance of our LQR controller given a constant offset on the pole angle estimate. This serves as a reflection of the inaccuracies of the neural network prediction of the pole angle, as seen in Figure 5.37. We chose a constant offset for noise contrary to the normally distributed noise in section 5.1.2, because our predictions are not reflective of adding Gaussian noise to the original pole observations.

In Figure 5.45, you can see the performance of the LQR configuration chosen in section 5.2 for constant offsets of 0.01 and 0.03 rad added to the pole angle. As you can see, with increasing noise, the balancing becomes less stable, as we balance the pole around a point further away from $\mathbf{q_t}$. However, we manage to balance a noise level of absolute values of up to 0.03 rad, added to both direction x and y. Empirically, for an offset of 0.04 rad, the balancing fails due to the significant tilt in the balancing position, causing the pole to slip and fall.

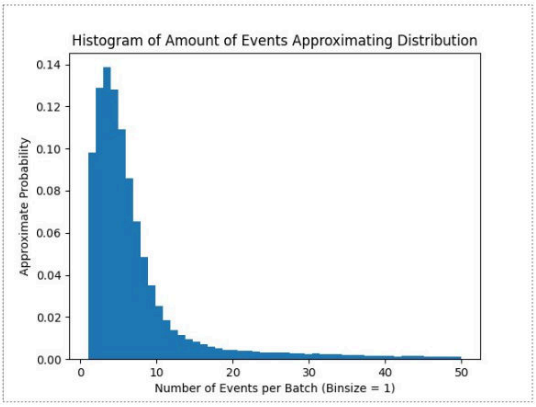


Figure 5.42.: Histogram approximating distribution of number of events per time-step of an example trajectory taken from the image dataset

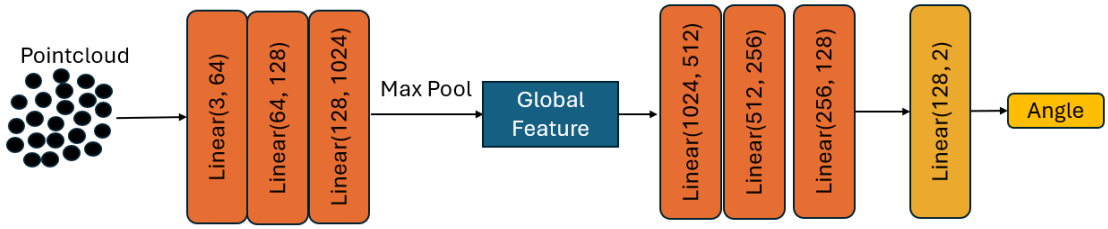


Figure 5.43.: Architecture of our PointNet example, the orange linear layers are paired with a ReLU afterwards

After validating the LQR being robust to offsets in prediction, we investigate the performance of our 4-layer CNN, predicting the angle for control.

We choose to evaluate the 4-layer CNN model, although it does not achieve the lowest evaluation error. However, as mentioned before, the CNN qualitatively performed the best, following the pole angle plot curves and their ups and downs better than, for example, the MLP trained on the dots (see figures 5.37 and 5.24). We import the model into the C++ control loop to replace the OptiTrack estimate of the pole state. We used LibTorch, the PyTorch variant for C++, and tested the runtime on CPU and GPU. A low runtime is crucial for real-time control, because we frequently need to predict and then correct the pole tilts for a working stabilization.

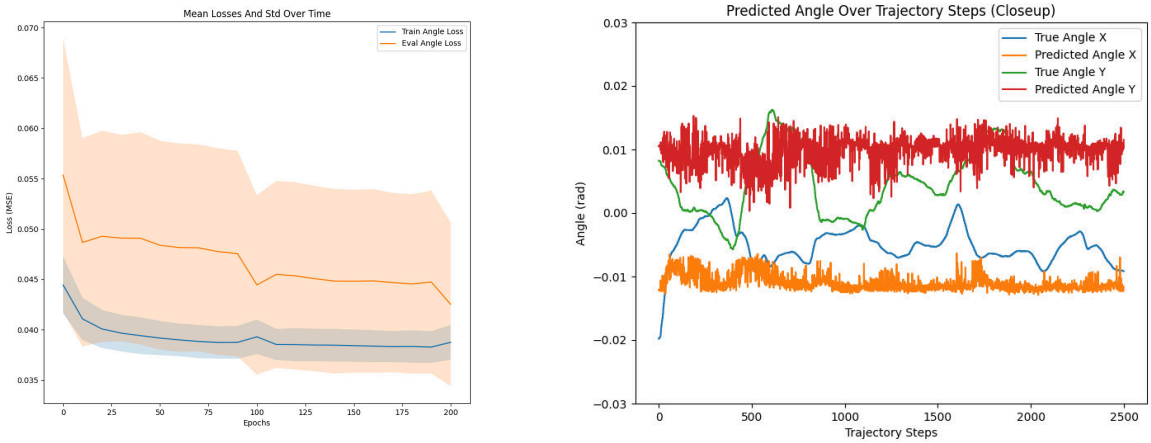


Figure 5.44.: Loss evolution (left) and predicted vs. true values (right) for network Pointnet cloudsize 50

The control loop of the WAM operates at a frequency of 500 Hz, i.e., every 2 ms. Since we want to be able to control the robot with that interval, we need to ensure that the inference time of our model is sufficiently short. In Figure 5.46, you can see the runtime for our model running on the CPU and the GPU, averaged across multiple roll-outs each. As you can see, the inference time on the CPU is quite high, with a mean execution time of around 1.7ms. On the GPU, we achieve a faster inference of around 0.48ms, which is a significant improvement.

Although the CNN inference time is relatively long compared to the other operations in the control loop, which take up to 0.04ms combined, it would still be feasible to execute the CNN during the control loop, as it takes less than 2ms. However, it would significantly increase the runtime, and to avoid this increase, the CNN inference is performed asynchronously in a separate callback. This callback is triggered by new ROS messages from the tactile sensor containing new events. The control loop simply accesses the variable updated by the callback, which holds the pole tilt. The callback function processes the events as outlined in Algorithm 1.

A crucial step to reducing runtime latency is "warming up" the CNN, using dummy inputs. This allows the weights and kernels to be initialized and loaded into the GPU cache, thus ensuring faster execution during actual inference.

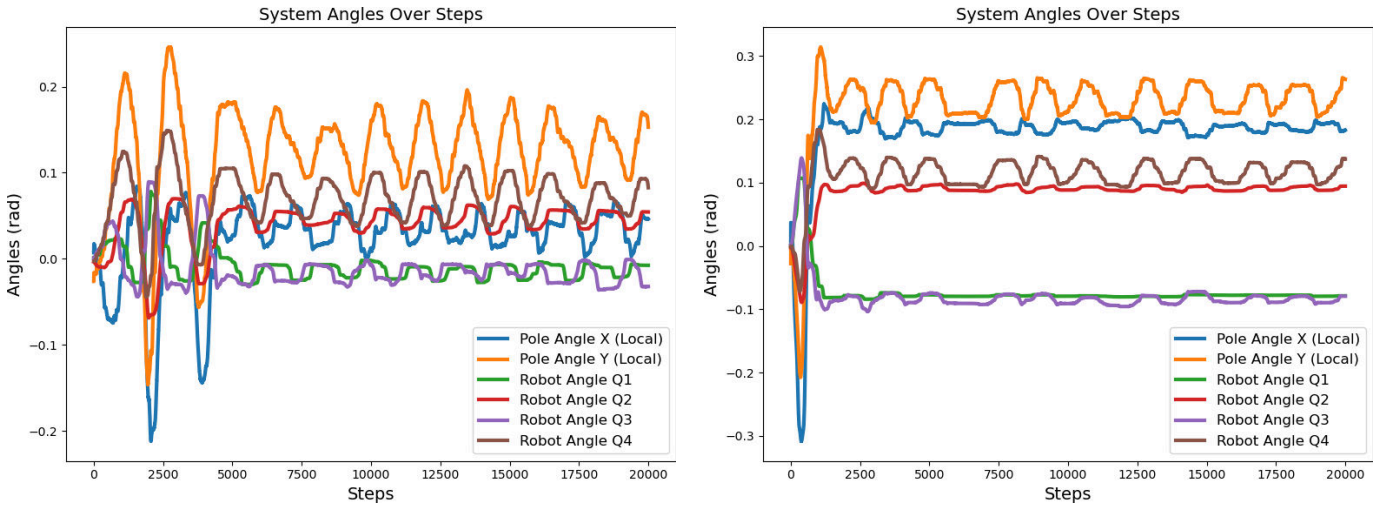


Figure 5.45.: Example of LQR performance for a constant offset of 0.01 rad (left) and 0.03 rad (right) added to the pole angle in both x and y direction.

We tested the performance of our LQR controller pipeline when approximating the pole state with our trained 4-layer CNN, as it achieved the best qualitative performance. However, during roll-out, the balancing is quickly interrupted because the generated torques lead to a violation of the joint velocity limits, since the robot is moving rapidly. An example of a roll-out and its angles and velocities can be seen in Figure 5.47.

As shown in the image, the fourth joint and the pole are moving at an unsafe velocity, leading to the termination of the control loop. The estimate of our network is quite far away from the Optitrack estimate, which is to be expected given the inaccurate results in section 5.3.4. Another problem is that initially the prediction does not change (flat line at the beginning), which is because we only observe events that are not considered noise, after 35 control steps, due to the few motions of the pole at the beginning of the LQR control loop. Afterwards, we receive an increasing number of events per control-step, but the regressed angles are not close to the Optitrack estimate (ground truth). In the end, we failed to balance the pole with the CNN.

Just to verify, we also tested if the MLP trained on the dot-dataset is capable of balancing the pole. However, the MLP is also unable to balance the pole and the behavior is nearly identical.

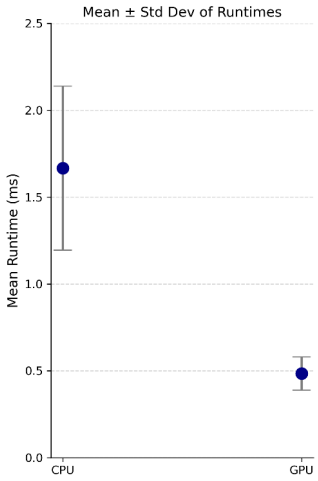


Figure 5.46.: Average inference runtime in ms for our 4-layer CNN on the CPU and GPU

5.3.8. Summarizing Table

Lastly, as a conclusion to summarize the results for different architectures and datasets, we provide a table of all experiments and their performance in table 5.2:

As you can see, the only method that achieves a low evaluation error for regressing the pole angle is the MLP on the Robot data. However, it is disqualified for actual control since it just simply learned the correlation of robot data and pole tilt. For all other networks, the evaluation loss is high and the prediction performance is improvable. The CNN architecture enables a lower training error than the MLP architectures but performs worse on the evaluation set than the simple dot based approach regarding the MSE.

However, qualitatively, as shown in Figure 5.37, the CNN predicts the ups and downs of the angles more accurately, doing less over-smoothing, but the total offset of the CNN prediction to the angles is higher, resulting in a higher MSE. The real roll-outs of the CNN and the MLP failed to stabilize the pendulum, which is to be expected given the evaluation loss and prediction plot examples.

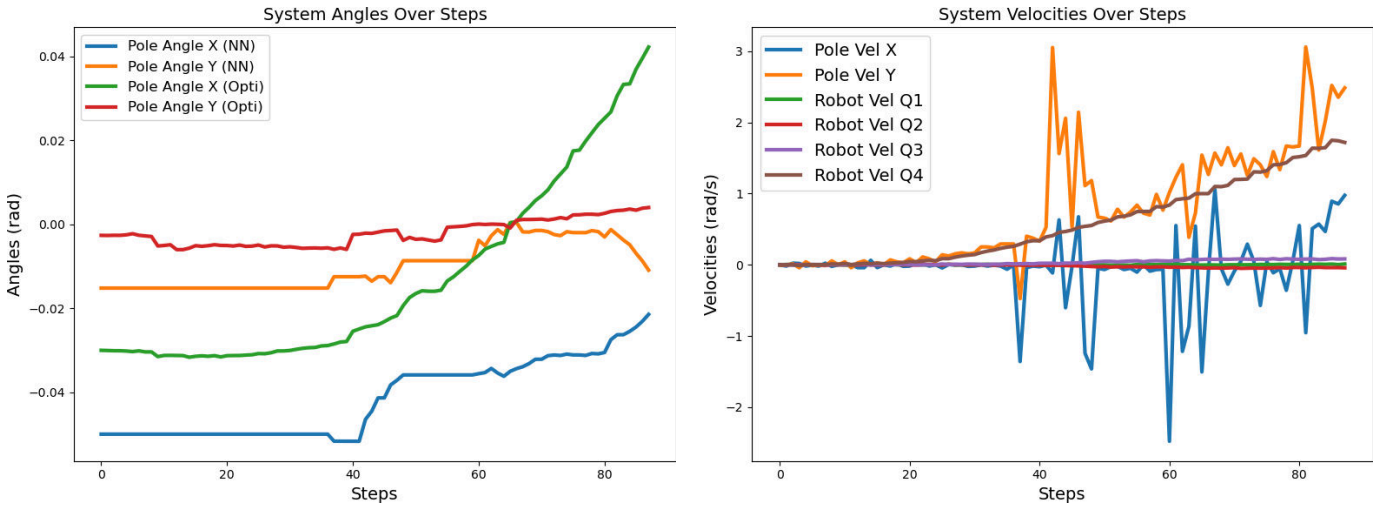


Figure 5.47.: Example of the performance of our trained CNN predicting the angle during roll-outs of the control pipeline. Angle prediction and OptiTrack estimate (left), velocity estimate and joint velocity evolution (right)

Network structure	Training Loss (MSE)	Evaluation Loss (MSE)
MLP Dots	0.0046	0.012
MLP Dots + Robot	0.003	0.007
MLP Robot	0.004	0.004
CNN naive	0.0001	0.031
CNN Dropout	0.0005	0.0246
CNN Dropout + Shifting	0.0012	0.0186
LSTM $\gamma = 0$	0.032	0.052
LSTM $\gamma = 0.999$	0.0021	0.029
Real Roll-out CNN/MLP	fails	fails

Table 5.2.: Comparisons of the different approaches

6. Conclusion

To conclude this thesis, we will summarize the results of our work and list limitations and improvement opportunities.

6.1. Summary of Experiment Results

As seen in the previous section, we successfully balanced the pole in simulation as well as with a visual estimate of the pole on the real system. However, the main research goal of tactile pole balancing was not achieved. Although we proposed and evaluated several approaches on different datasets, actively balancing a pole with the robot remains an open challenge. Regressing the pole angle during learning is already inaccurate, and the real-time inference leads to a quick stop due to safety limit violation, as our pole predictions do not reflect the true pole state.

6.2. Limitations and Improvements

The quality of the real-world balancing and training of the models is negatively impacted by a few circumstances we list in the following paragraphs.

6.2.1. Position of the Free Pole

For the real free pole setup, it is hard to guarantee that the pole is standing perfectly upright. In fact, for most of our datasets, the pole does not start upright as can be seen in Figure 5.37, where the starting angle value of both x and y is around absolute values of 0.015 for the evaluation. Starting in the upright position would be beneficial because

we assume that the pole is starting in the equilibrium point and has no velocity or tilt. Violating this assumption requires us to apply torques to the robot directly when switching to the LQR, which is not ideal. Furthermore, it makes learning harder as for an initial tilt angle that is not zero, the initially empty image or the starting dot locations must predict a value that is not zero. Improving our setup with the cable pull and the manual release when starting the LQR could therefore be beneficial to the stabilization and data recording. For example, future research could enforce the upright position with some form of hold around the pole to limit its movement. Perfectly releasing the hold once the LQR is started by a mechanism, so that the movement of the pole is never artificially influenced by the human holding the rope, would also improve the initial behavior at the start. Related, it is hard to manually ensure the pole is in the middle of the gel, since the pole is neither prevented from translating by the cable pull setup, nor is its position fixed. Centering the pole was performed manually, but of course, this is imperfect, and a machine routine to position the pole in the middle of the gel would be better. Especially for learning from the data and inference, it would be a significant improvement if the event locations were in related locations across different trajectories.

6.2.2. Sim-to-Real Gap

The URDF of the WAM did not reflect the real-world behavior of the WAM perfectly. For example, the gravity compensation term for the second joint needed to be increased artificially to avoid a drift of the second joint once switching to the LQR. Likely, this is not the only inaccuracy in the model, as the behavior of the WAM in simulation and real world for different LQR values differs a lot, as seen in section 5.1 and 5.2. A more accurate model would help bridge the sim to real gap, improving the applicability of LQR parameters determined in the MuJoCo simulation. Furthermore, with a more accurate model, the LQR control would improve, because we would have a more accurate linearization of the behavior of the robot. Additionally, the Luenberger-Observer might be helpful for stabilization, given an accurate model, replacing the finite difference approximation. During the thesis, we shortly investigated a system identification of the WAM, but ultimately could not find better performing configurations than the already used ones.

Another option would be to simulate the tactile sensor, which would enable us to learn in simulation and not with the real software, which has the benefit of faster and safer roll-outs, enabling different learning techniques such as reinforcement learning.

6.2.3. Event Generation and LQR Controller Parameters

Since all neural networks need data i.e., events for learning the mapping, another area of improvement would be the amount of events generated by the LQR-roll-out. As outlined in chapter 3.2.4, e.g., a heavier pole causes more events because it deforms the gel more. Additionally, more movement creates more events. Therefore, a LQR controller that results in continuous motion of a heavy pole would be advantageous for the amount of events generated, which in turn would help detect the edges of the pole and thus improve Deep Learning results. However, empirically, a heavy pole that moves on the gel can easily cause damage to the gel even with the 3-D-printed add-on on the bottom we designed. This resulted in a frequent need for replacing the gels for data generation because the damaged gels would have constant events where the gel was damaged, making them unsuitable for learning. A more robust gel or a smoother contact point of pole and gel could eliminate this problem, which was detrimental to capturing more data, as running out of undamaged gels became a bottleneck.

Furthermore, we do not claim that the LQR parameters used to generate the dataset are optimal, since the field of optimization was not explored completely. There might be better configurations for our task, especially robustness to, for example foreign disturbance (e.g. hitting the pole while balancing to cause external pole tilt) or more plentiful event generation with more or maybe also less movement of the robotic arm. Inspecting the LQR parameter and capturing new datasets may benefit training as it could limited the range of tilt angles or provide a wider variety of trajectories enabling better learnability and generalization of the learned models.

6.2.4. OptiTrack and Velocity Estimation

Our ground truth data relies on OptiTrack to accurately capture the pole angle precisely at all times. However, OptiTrack may lose a marker during the roll-out of the LQR controller, leading to an imprecise estimate of the pole angle. This issue can be seen in the image of the third data set 4.6, where sudden jumps in the pole angle occur, due to its fast motion. These kinds of jumps in the pole angle estimation are wrong and negatively impact the capability to train on them. The problem could be resolved by adding more cameras capturing the pole and adding more markers, so that it is harder to completely lose track of multiple or all markers at any time, thus ensuring a robust line estimate. Furthermore, regarding the velocity estimate, we relied on a finite difference computation, which is a simple but noisy method that is also heavily dependent on the OptiTrack estimation to be

smooth and accurate. Improving the accuracy and robustness of both pole position and velocity estimation would result in cleaner data to train on. Estimating the velocity in a more sophisticated approach could help as well, e.g., relying on a model-based prediction like the Luenberger-Observer, which in turn relies on accurate modeling. Alternatively, a sensor that measures velocity could be used.

Similarly, it might be beneficial to learn also the velocity and acceleration of the pole instead of simply the angle, as humans use their tactile sense for these properties as stated in [37]. A big problem for this would be that currently there is no ground truth data for both the velocity or acceleration to learn a neural network on, only our approximations via the finite differences.

6.2.5. Time Constraints and Hyper-Parameters For Neural Networks

Training on images and LSTMs requires a large amount of data to be stored in RAM and eventually on disk. Loading from the disk is a costly operation in terms of time, when done during training, substantially increasing the runtime. With more available RAM or faster loading techniques for when we need to load the data partially from disk could improve learn speed greatly, enabling a faster search for network configurations or longer training iterations. In general, a bigger memory for both RAM and GPU could lead to more efficient loading and processing of data.

Improving the datasets by capturing more trajectories to learn on, maybe also with more variance in roll-outs and angles seen, likely helps the generalizing capabilities of the presented architectures. Another option is to amend the datasets by considering different permutations besides shifts, like rotations, during training. Different types of datasets, than images, pointclouds, or dot locations, may also be better suited for learning the pole angle.

Another possible improvement would be regarding the neural network architectures and training procedures. For example, for the LSTMs, we only considered a smaller training horizon, which may be problematic because LSTMs can suffer from the vanishing gradient problem and might take longer to train to achieve better results. Similarly, different MLP or CNN structures, learning rates, batch-sizes, and other hyper-parameter than the ones we explored might be better for performance. Since we were limited by the computational power of our hardware and the timing constraints of the thesis, we could not explore all the options available.

6.2.6. Alternative Learning Techniques

Our supervised learning approach may be an insufficient or ineffective way to tackle the balancing task. Commonly, for robotic tasks, reinforcement learning algorithms are used to achieve state-of-the-art performance. However, reinforcement learning would require frequent and reliable interaction with the environment and roll-outs of the policy. In the current state of both the simulation and the real setup, the data generation process for reinforcement learning would be difficult, because the simulation does not model the tactile sensor, and the simulation seems to be an inaccurate reflection of the real WAM and its behavior, as mentioned prior. For the real setup, we face the issue that the starting conditions are not the same between roll-outs, and each roll-out takes a significant amount of time. Nevertheless, if these obstacles can be overcome, reinforcement learning might be a promising way to improve inference results. For example, instead of learning the angle prediction, the reinforcement algorithm could also learn to predict a motor control signal directly based on the events. We shortly investigated learning the torques instead of a pole angle regression with our supervised setup, but decided against it. This is because the performance of the resulting control would be hard to compare to the visual-inferred LQR controller. After all, the pipeline would be vastly different, and we wanted a similar control loop for comparability. However, training a policy to predict the necessary motor control torques might be a generally better approach to the issue.

6.2.7. Limits of the Sensor

Lastly, the sensor has some inherent limitations. For example, due to the setup of the LED stripe, the sensor is detecting events caused by edges rotated towards the parallels of the sensors x-axis worse, as the effective areas of events are considerably smaller than the ones for edges facing the parallels of the y-axis, as depicted in 3.2.4. As seen in Figure 5.37, for both angles the prediction is not accurate over the entire prediction, but especially for the x-angle the prediction is overly smooth, missing most of the peaks and lows of the angle curves. Only with sufficient motion of the pole and the resulting events generated by that, can we closely predict the angle, as can be seen for the trajectory steps 22500 to 30000 in the same figure on the left. Furthermore, the LED is a single light color, which is uncommon for visual-tactile sensors, as they mainly work on an RGB camera and illumination, where the image is split into the usual input channel as mentioned in [1]. Multiple colors enable a depth estimate of the gel deformation through stereo, which may be essential for determining the pole tilt angle accurately. However, for the Evetac, we discard this information and solely rely on events, which may be detrimental.

6.3. Outlook

To summarize, this work leaves a lot of open research fields in order to improve or to achieve tactile pole balancing, including runtime reduction of networks, training time and techniques, more sophisticated architectures and datasets, more accurate models of the robotic hardware and the dynamics, improved state estimation techniques and setup suggestions.

We believe, that tactile pole balancing is possible to achieve and we hope to contribute to the task in a meaningful manner with the insights and experiments in this thesis.

Bibliography

[1] S. Dikhale, K. Patel, D. Dhingra, I. Naramura, A. Hayashi, S. Iba, and N. Jamali, "Visuotactile 6d pose estimation of an in-hand object using vision and tactile sensor data," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 2148–2155, 2022.

[2] N. Kuppuswamy, A. Castro, C. Phillips-Grafflin, A. Alspach, and R. Tedrake, "Fast model-based contact patch and pose estimation for highly deformable dense-geometry tactile sensors," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 1811–1818, 2020.

[3] N. F. Lepora, "The future lies in a pair of tactile hands," *Science Robotics*, vol. 9, no. 91, p. eadq1501, 2024.

[4] R. D. Howe, "Tactile sensing and control of robotic manipulation," *Advanced Robotics*, vol. 8, no. 3, pp. 245–261, 1993.

[5] C. Atkeson and S. Schaal, "Learning tasks from a single demonstration," in *Proceedings of International Conference on Robotics and Automation*, vol. 2, pp. 1706–1712 vol.2, 1997.

[6] S. Schaal, "Learning from demonstration," in *Advances in Neural Information Processing Systems* (M. Mozer, M. Jordan, and T. Petsche, eds.), vol. 9, MIT Press, 1996.

[7] M. I. Fadholi, Suhartono, P. S. Sasongko, and Sutikno, "Autonomous pole balancing design in quadcopter using behaviour-based intelligent fuzzy control," in *2018 2nd International Conference on Informatics and Computational Sciences (ICICoS)*, pp. 1–6, 2018.

[8] J. Conradt, M. Cook, R. Berner, P. Lichtsteiner, R. Douglas, and T. Delbruck, "A pencil balancing robot using a pair of aer dynamic vision sensors," pp. 781 – 784, 06 2009.

[9] N. Funk, E. Helmut, G. Chalvatzaki, R. Calandra, and J. Peters, "Evetac: An event-based optical tactile sensor for robotic manipulation," 2024.

[10] J. S. Brownlee, "The pole balancing problem: a benchmark control theory problem," 2005.

[11] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834–846, 1983.

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[13] D. Hougen, J. Fischer, and D. Johnam, "A neural network pole balancer that learns and operates on a real robot in real time," 07 1994.

[14] B. D. O. Anderson and J. B. Moore, *Optimal control: linear quadratic methods*. USA: Prentice-Hall, Inc., 1990.

[15] R. Banerjee and A. Pal, "Stabilization of inverted pendulum on cart based on pole placement and lqr," in *2018 International Conference on Circuits and Systems in Digital Enterprise Technology (ICCSDET)*, pp. 1–5, 2018.

[16] F. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuro-evolution.," pp. 1356–1361, 01 1999.

[17] F. Gomez and R. Miikkulainen, "2-d pole balancing with recurrent evolutionary networks," in *ICANN 98* (L. Niklasson, M. Bodén, and T. Ziemke, eds.), (London), pp. 425–430, Springer London, 1998.

[18] J. Harris, *An Introduction to Fuzzy Logic Applications*. USA: Kluwer Academic Publishers, 2000.

[19] M. J. Er, B. H. Kee, and C. C. Tan, "Design and development of an intelligent controller for a pole-balancing robot," *Microprocessors and Microsystems*, vol. 26, no. 9, pp. 433–448, 2002.

[20] D. Nguyen-Tuong, M. Seeger, and J. Peters, "Learning models for control," *Networks*, vol. 18, no. 1, pp. 71–90, 2005.

[21] H. Lee, J. Gil, S. You, Y. Gui, and W. Kim, "Arm angle tracking control with pole balancing using equivalent input disturbance rejection for a rotational inverted pendulum," *Mathematics*, vol. 9, p. 2745, 10 2021.

[22] R. S. Bhourji, S. Mozaffari, and S. Alirezaee, "Reinforcement learning ddpg–ppo agent-based control system for rotary inverted pendulum," *Arabian Journal for Science and Engineering*, vol. 49, no. 2, pp. 1683–1696, 2024.

[23] O. Saleem and J. Iqbal, "Phase-based adaptive fractional lqr for inverted-pendulum-type robots: Formulation and verification," *IEEE Access*, 2024.

[24] M. Safeea and P. Neto, "A q-learning approach to the continuous control problem of robot inverted pendulum balancing," *Intelligent Systems with Applications*, vol. 21, p. 200313, 2024.

[25] P. Klink, F. Wolf, K. Ploeger, J. Peters, and J. Pajarinen, "Tracking control for a spherical pendulum via curriculum reinforcement learning," 2023.

[26] Barrett Technology, "Barrett wam robotic arm." `https://barrett.com/wam-arm`. Accessed: 2024-11-27.

[27] J. Baek, C. Lee, Y. S. Lee, S. Jeon, and S. Han, "Reinforcement learning to achieve real-time control of triple inverted pendulum," *Engineering Applications of Artificial Intelligence*, vol. 128, p. 107518, 2024.

[28] OptiTrack, "Optitrack - motion capture systems." `https://www.optitrack.com/`, 2024. Accessed: 2024-11-26.

[29] W. Ye, Z. Li, C. Yang, J. Sun, C.-Y. Su, and R. Lu, "Vision-based human tracking control of a wheeled inverted pendulum robot," *IEEE transactions on cybernetics*, vol. 46, no. 11, pp. 2423–2434, 2015.

[30] F. Casali, *Real-Time State Estimation of a Two-Wheeled Inverted Pendulum Robot for Motion and Navigation Control*. PhD thesis, Politecnico di Torino, 2021.

[31] D. Tang, "Flying inverted pendulum," in *2018 5th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*, pp. 30–34, 2018.

[32] S. Schmidt, "Using reinforcement learning to teach a humanoid robot new behaviors." `https://tams.informatik.uni-hamburg.de/publications/2023/BSc_Sven_Schmidt.pdf`, 2023. Bachelor's Thesis, University of Hamburg.

[33] R. Prakash, L. Behera, and S. Jagannathan, "Adaptive critic optimal control of an uncertain robot manipulator with applications," *IEEE Transactions on Control Systems Technology*, pp. 1–11, 2024.

[34] P. Thiha Kyaw, A. V. Le, L. Yi, P. Veerajagadheswar, M. R. Elara, D. T. Vo, and M. Bui Vu, "Greedy heuristics for sampling-based motion planning in high-dimensional state spaces," *arXiv e-prints*, pp. arXiv–2405, 2024.

[35] E. Das, T. Touma, and J. W. Burdick, "Bayesian optimal experimental design for robot kinematic calibration," *arXiv preprint arXiv:2409.10802*, 2024.

[36] E. Tzorakoleftherakis, F. A. Mussa-Ivaldi, R. A. Scheidt, and T. D. Murphey, "Effects of optimal tactile feedback in balancing tasks: A pilot study," in *2014 American Control Conference*, pp. 778–783, 2014.

[37] "Experimental estimation of tactile reaction delay during stick balancing using cepstral analysis," *Mechanical Systems and Signal Processing*, vol. 138, p. 106554, 2020.

[38] Z. Kappassov, J.-A. Corrales, and V. Perdereau, "Tactile sensing in dexterous robot hands," *Robotics and Autonomous Systems*, vol. 74, pp. 195–220, 2015.

[39] M. Meribout, N. A. Takele, O. Derege, N. Rifiki, M. El Khalil, V. Tiwari, and J. Zhong, "Tactile sensors: A review," *Measurement*, p. 115332, 2024.

[40] J. Lenz, T. Gruner, D. Palenicek, T. Schneider, and J. Peters, "Analysing the interplay of vision and touch for dexterous insertion tasks," *arXiv preprint arXiv:2410.23860*, 2024.

[41] W. Yuan, S. Dong, and E. H. Adelson, "Gelsight: High-resolution robot tactile sensors for estimating geometry and force," *Sensors*, vol. 17, no. 12, 2017.

[42] J. Bimbo, S. Luo, K. Althoefer, and H. Liu, "In-hand object pose estimation using covariance-based tactile to geometry matching," *IEEE Robotics and Automation Letters*, vol. 1, no. 1, pp. 570–577, 2016.

[43] M. Bauza, A. Bronars, and A. Rodriguez, "Tac2pose: Tactile object pose estimation from the first touch," *The International Journal of Robotics Research*, vol. 42, no. 13, pp. 1185–1209, 2023.

[44] H. Li, J. Akl, S. Sridhar, T. Brady, and T. Padir, "Vita-zero: Zero-shot visuotactile object 6d pose estimation," 2025.

[45] G. M. Caddeo, N. A. Piga, F. Bottarel, and L. Natale, "Collision-aware in-hand 6d object pose estimation using multiple vision-based tactile sensors," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 719–725, 2023.

[46] D. J. Nagy, J. G. Milton, and T. Insperger, "Controlling stick balancing on a linear track: Delayed state feedback or delay-compensating predictor feedback?," *Biological Cybernetics*, vol. 117, no. 1, pp. 113–127, 2023.

[47] Wikipedia contributors, "Linear–quadratic regulator — Wikipedia, the free encyclopedia," 2025. [Online; accessed 30-April-2025].

[48] Wikipedia contributors, "Proportional–integral–derivative controller — Wikipedia, the free encyclopedia," 2025. [Online; accessed 30-April-2025].

[49] S. Irfan, L. Zhao, S. Ullah, A. Mehmood, and M. Fasih Uddin Butt, "Control strategies for inverted pendulum: A comparative analysis of linear, nonlinear, and artificial intelligence approaches," *Plos one*, vol. 19, no. 3, p. e0298093, 2024.

[50] L. Ntogramatzidis, V. Arumugam, and A. Ferrante, "On the structure of the solution of continuous-time algebraic riccati equations with closed-loop eigenvalues on the imaginary axis.⁎⁎partially supported by the australian research council under the grant dp190102478.," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 6875–6880, 2020. 21st IFAC World Congress.

[51] "A new approach to linear filtering and prediction problems," 2002.

[52] D. Luenberger, "An introduction to observers," *Automatic Control, IEEE Transactions on*, vol. 16, pp. 596 – 602, 01 1972.

[53] G. Casiez, N. Roussel, and D. Vogel, "1€ filter: A simple speed-based low-pass filter for noisy input in interactive systems," *Conference on Human Factors in Computing Systems - Proceedings*, 05 2012.

[54] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system,"

[55] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, "The pinocchio c++ library – a fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *IEEE International Symposium on System Integrations (SII)*, 2019.

[56] J. Carpentier, F. Valenza, N. Mansard, *et al.*, "Pinocchio: fast forward and inverse dynamics for poly-articulated systems." https://stack-of-tasks.github.io/pinocchio, 2015–2021.

[57] Wikipedia contributors, "Universal approximation theorem — Wikipedia, the free encyclopedia," 2025. [Online; accessed 28-March-2025].

[58] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[59] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “Pointnet: Deep learning on point sets for 3d classification and segmentation,” 2017.

[60] S. Linnainmaa, “Taylor expansion of the accumulated rounding error,” *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976.

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” in *Advances in Neural Information Processing Systems*, vol. 3, pp. 1–9, 1991.

[62] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[63] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016.

[64] ISO, *ISO/IEC 14882:1998: Programming languages — C++*. Sept. 1998. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.

[65] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[66] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

[67] P. Research, “Prusaslicer,” 2025. Accessed: 2025-03-28.

[68] PrusaSlicer, “Prusaslicer,” 2025. Accessed: 2025-03-31.

[69] O. Inc., “Onshape,” 2025. Accessed: 2025-03-28.

A. Appendix

A.1. LQR Configuration Plots

In this section we provide a few plots to showcase the evaluation of the LQR parameter search in MuJoCo. Note that the configuration closest to 0,0 in terms of pole angle in e.g. figure A.1 is not necessarily the best configuration overall as we also need to consider the mean deviation to $\mathbf{q_t}$, which is not represented in these plot. For the overall best performance please refer to respective figures about the mean deviation from $\mathbf{q_t}$. In these figures we can see the top 20 best performing LQR configurations and their mean state error, i.e. the mean summed deviation form it. The sum consists of the error to the robot configuration, the desired pole angle, the robots velocity and the poles velocity.

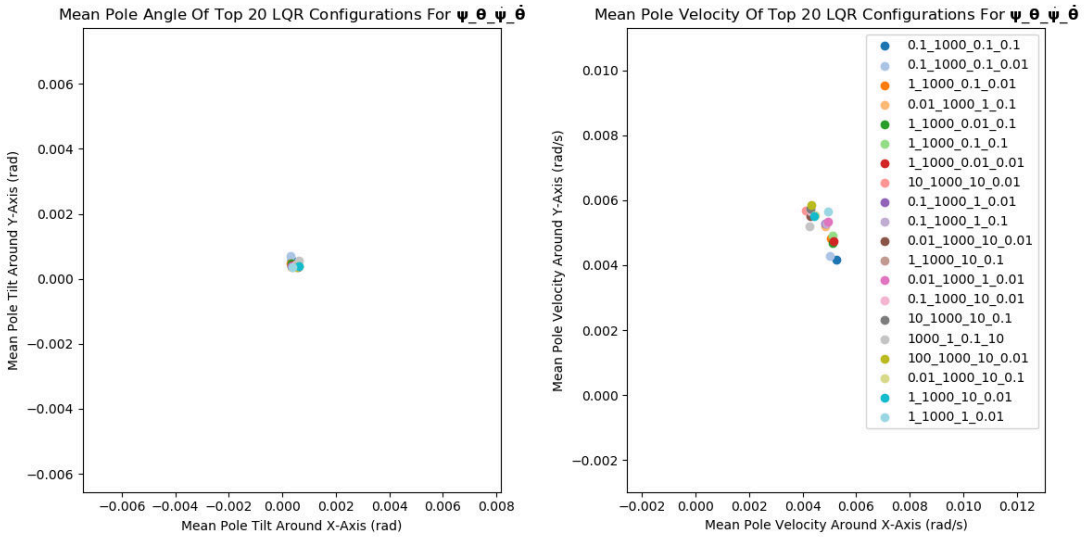


Figure A.1.: Mean pole angles and velocity, pole attached, noise 0.001

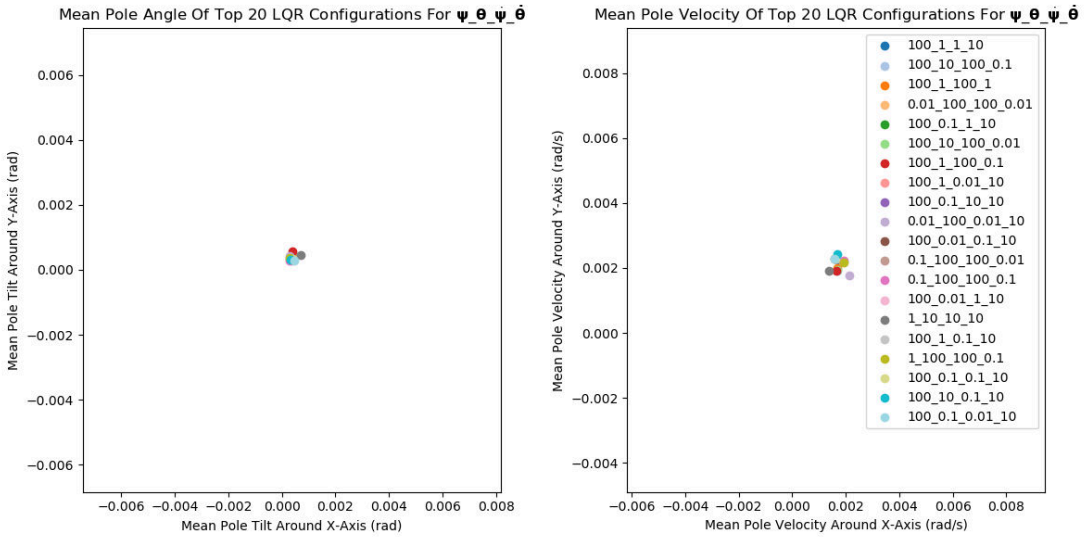


Figure A.2.: Mean pole angles and velocity, pole free, noise 0.001

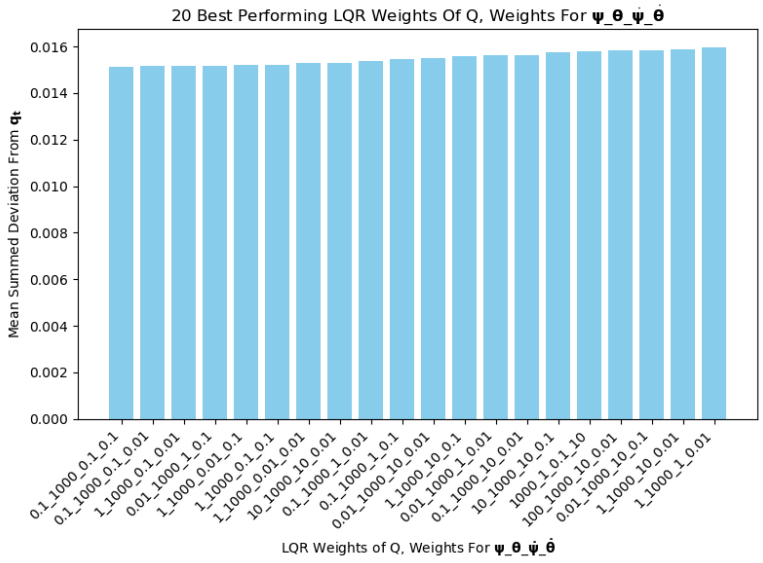


Figure A.3.: Deviation from $\mathbf{q_t}$, pole attached, noise 0.001

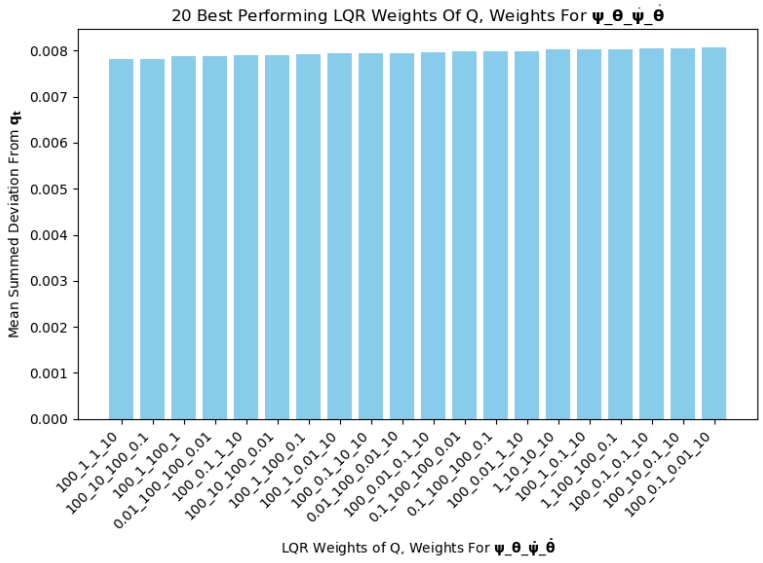


Figure A.4.: Deviation from $\mathbf{q_t}$, pole free, noise 0.001

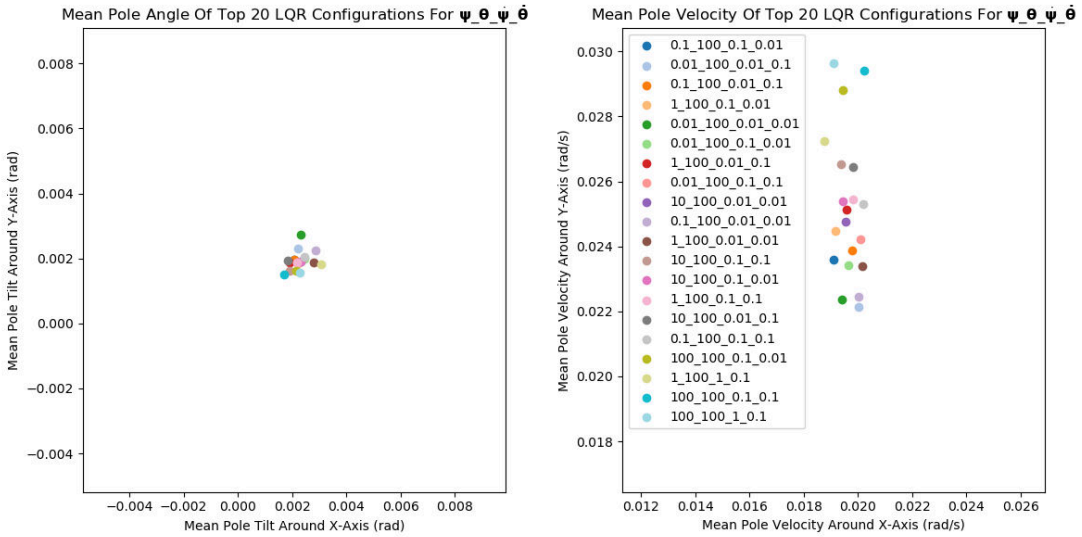


Figure A.5.: Mean pole angles and velocity, pole attached, noise 0.005

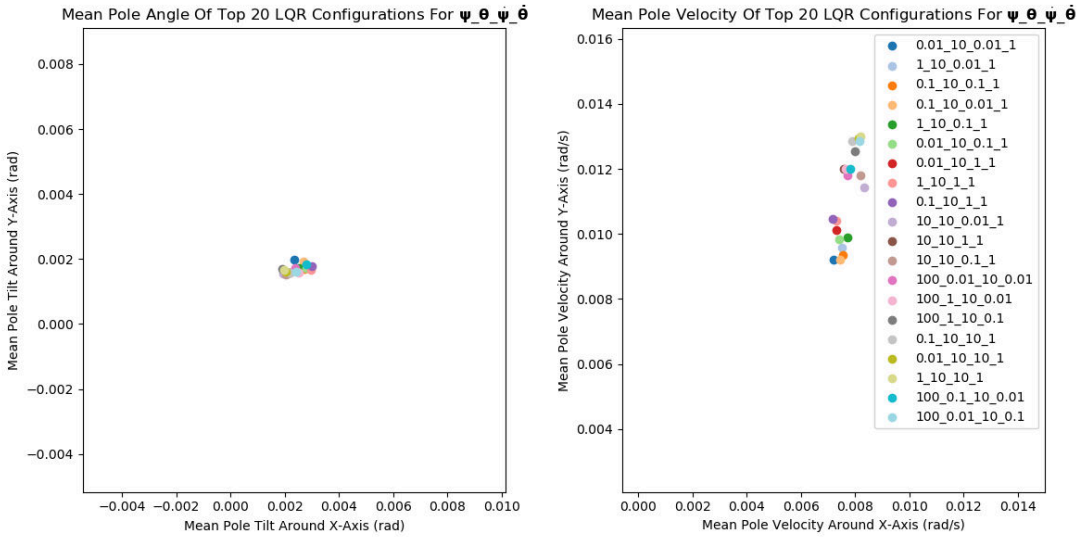


Figure A.6.: Mean pole angles and velocity, pole free, noise 0.005

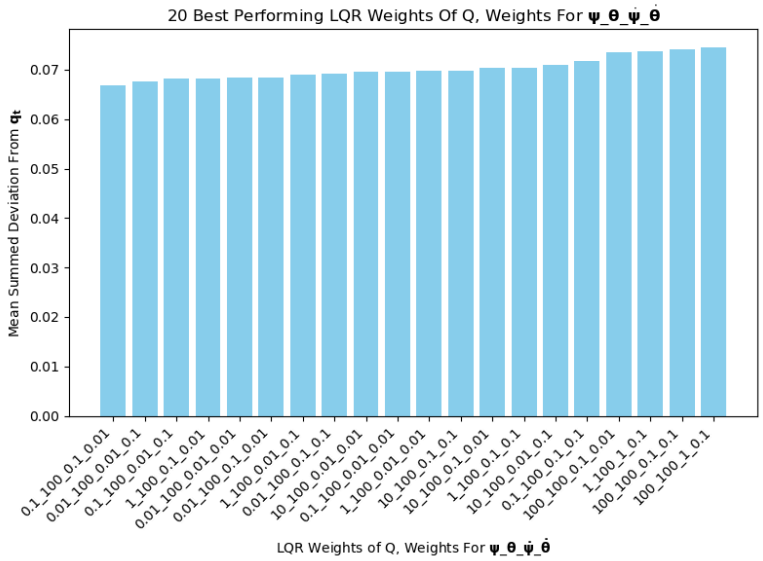


Figure A.7.: Deviation from $\mathbf{q_t}$, pole attached, noise 0.005

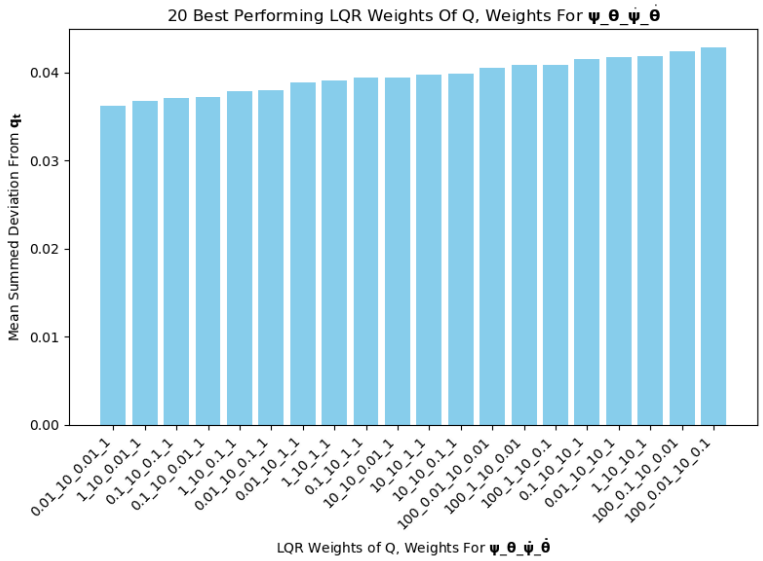


Figure A.8.: Deviation from $\mathbf{q_t}$, pole free, noise 0.005

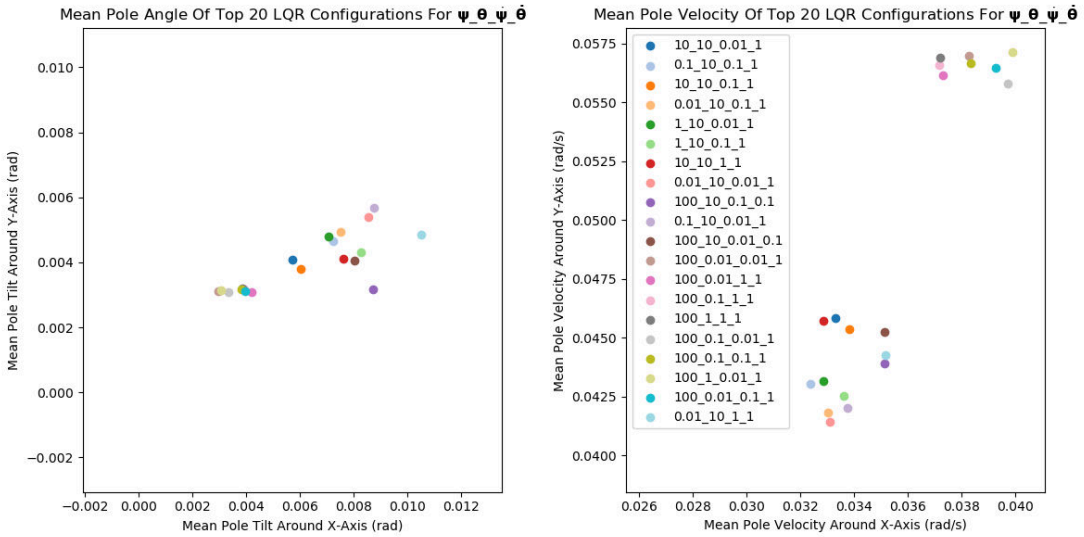


Figure A.9.: Mean pole angles and velocity, pole attached, noise 0.01

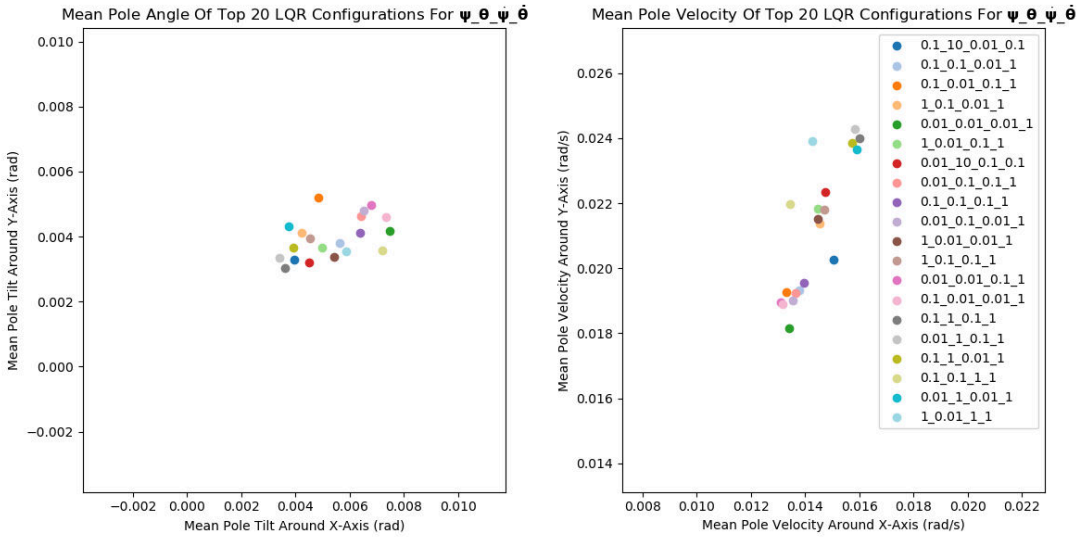


Figure A.10.: Mean pole angles and velocity, pole free, noise 0.01

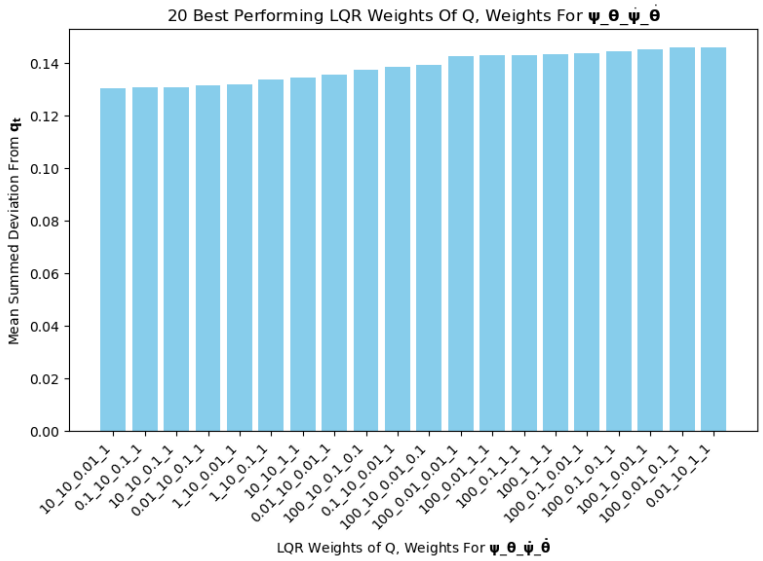


Figure A.11.: Deviation from $\mathbf{q_t}$, pole attached, noise 0.01

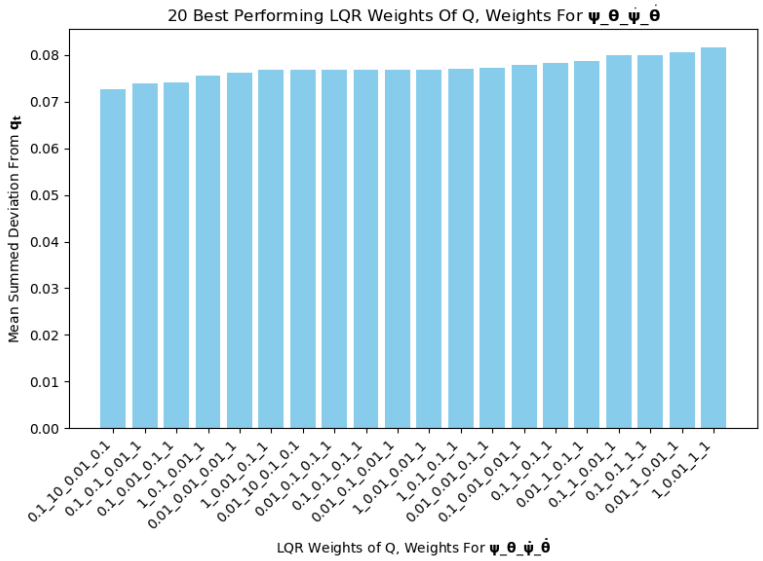


Figure A.12.: Deviation from $\mathbf{q_t}$, pole free, noise 0.01

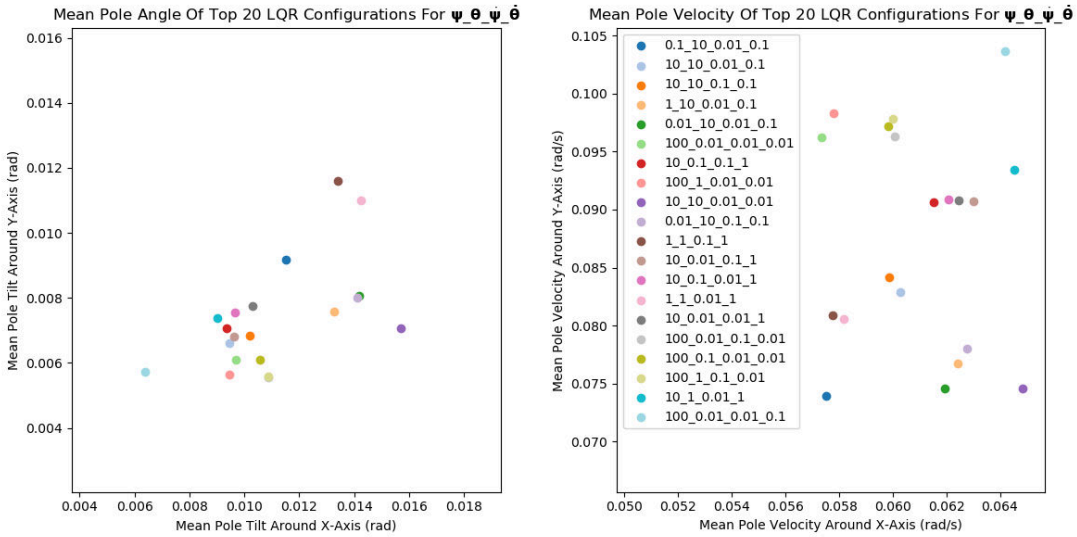


Figure A.13.: Mean pole angles and velocity, pole attached, noise 0.02

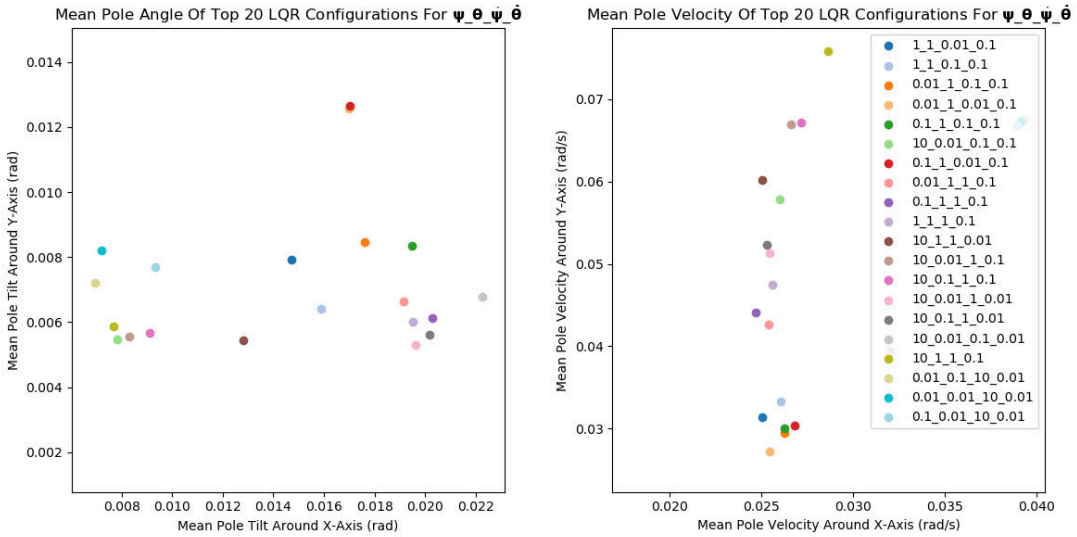


Figure A.14.: Mean pole angles and velocity, pole free, noise 0.02

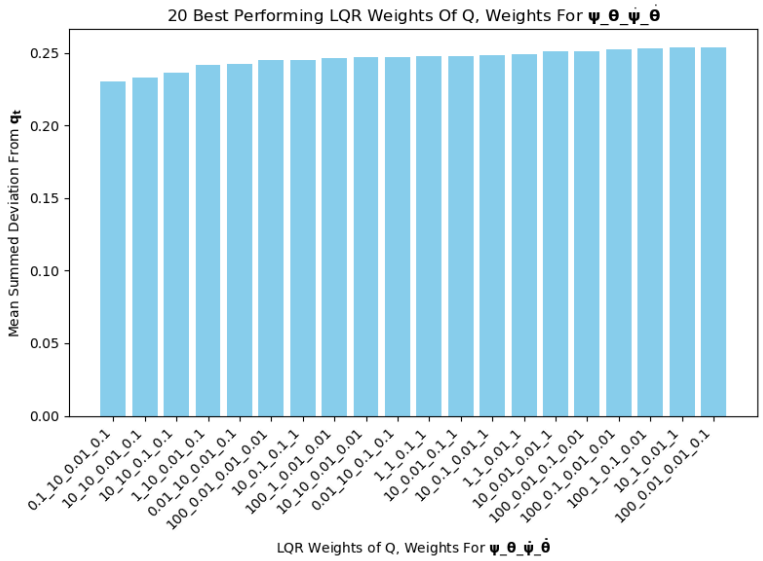


Figure A.15.: Deviation from $\mathbf{q_t}$, pole attached, noise 0.02

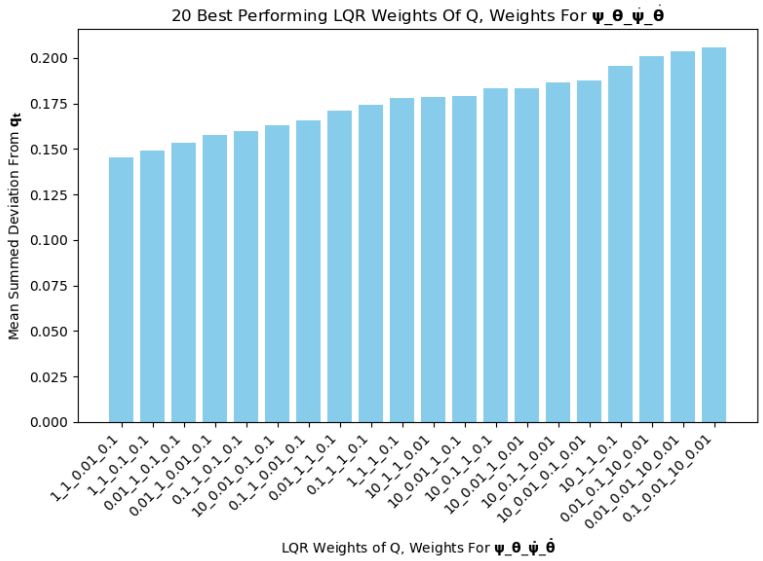


Figure A.16.: Deviation from $\mathbf{q_t}$, pole free, noise 0.02

A.2. Pole Angle And Velocity Over Time

In these plots we provide example trajectories when rolling out LQR controller in MuJoCo. As you can see in e.g. figure A.17 the pole is barely tilting before the stabilization of the LQR controller corrects the error. Furthermore, for higher noise levels the error as well as the standard deviation increase. The velocity is centered around 0 and is a lot more jittery than the angle itself.

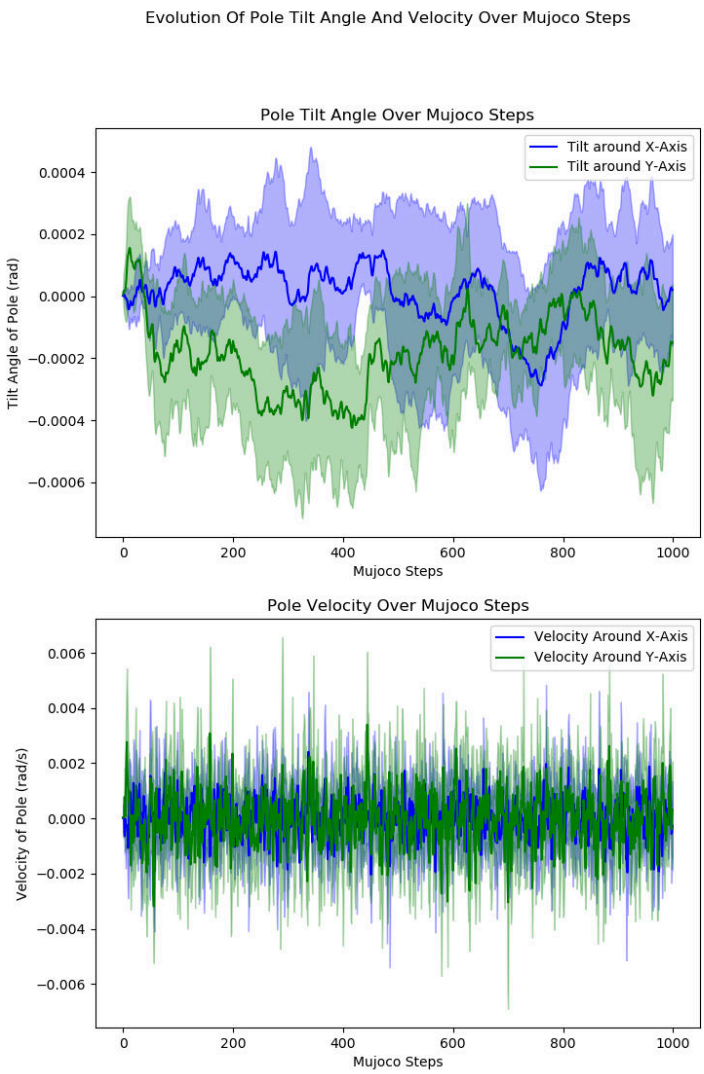


Figure A.17.: Pole tilt angle and velocity – free, noise 0.001

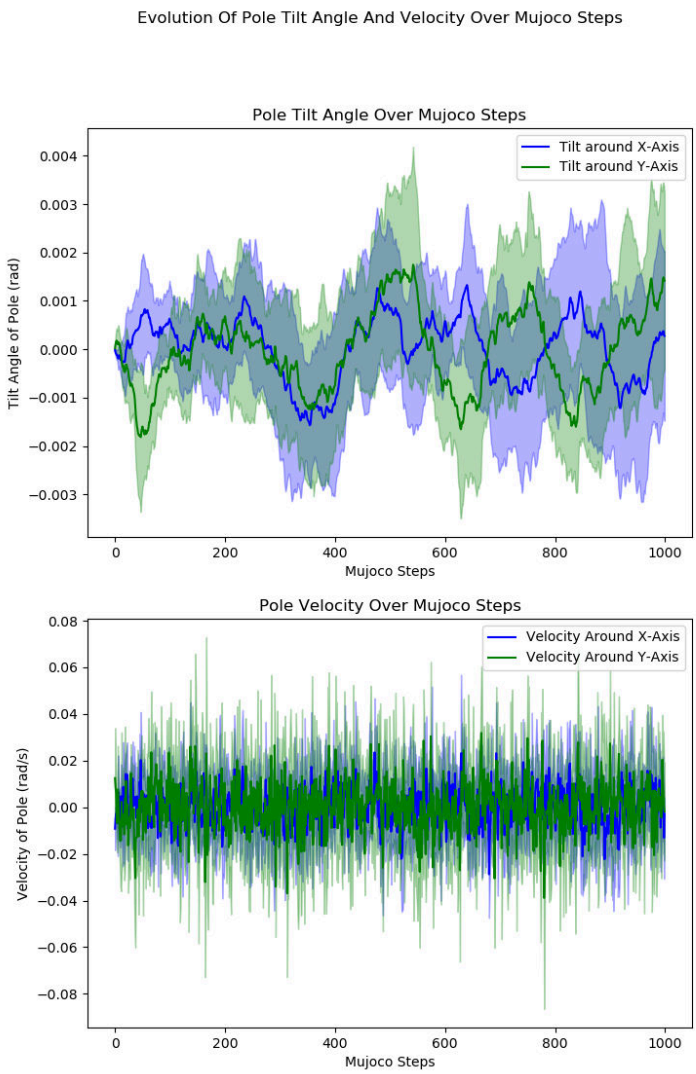


Figure A.18.: Pole tilt angle and velocity – attached, noise 0.005

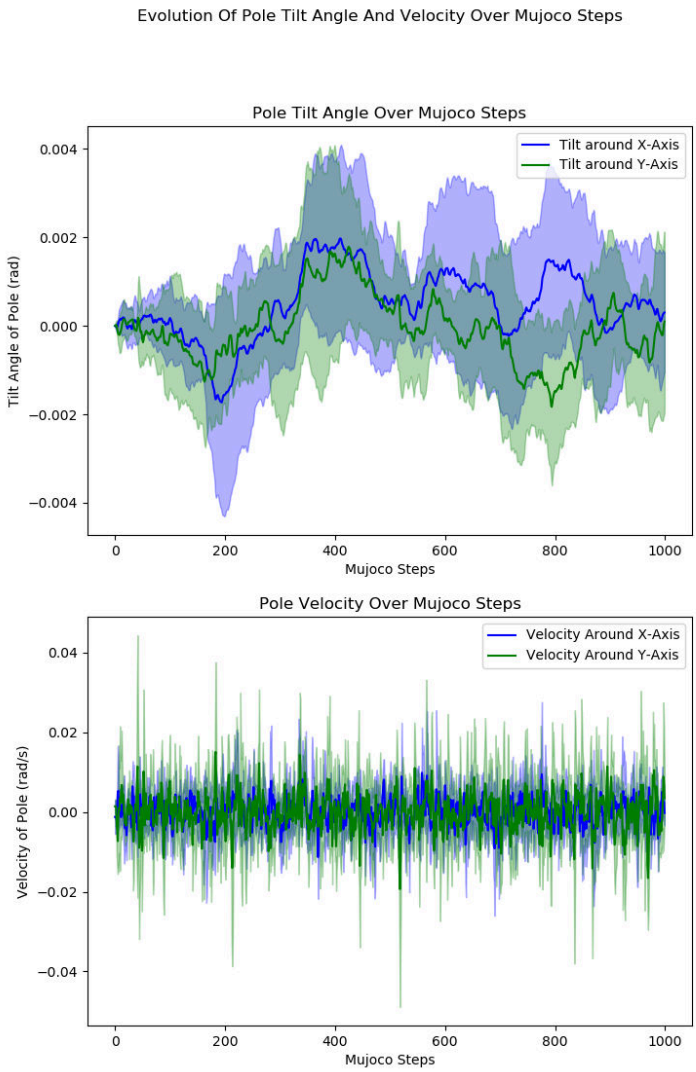


Figure A.19.: Pole tilt angle and velocity – free, noise 0.005

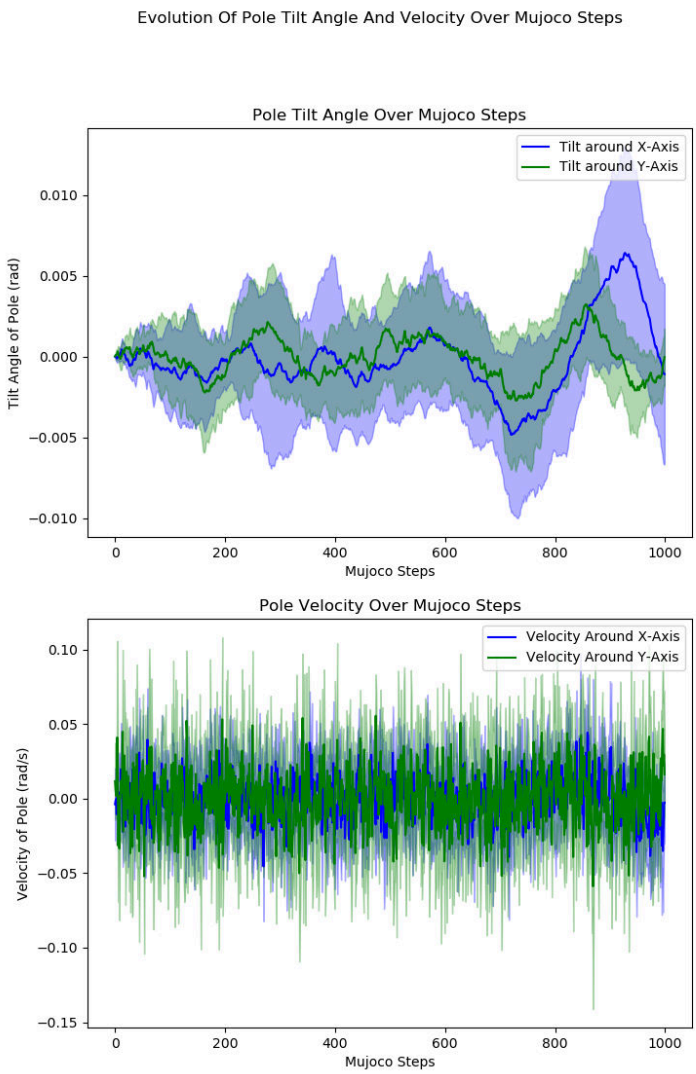


Figure A.20.: Pole tilt angle and velocity – attached, noise 0.01

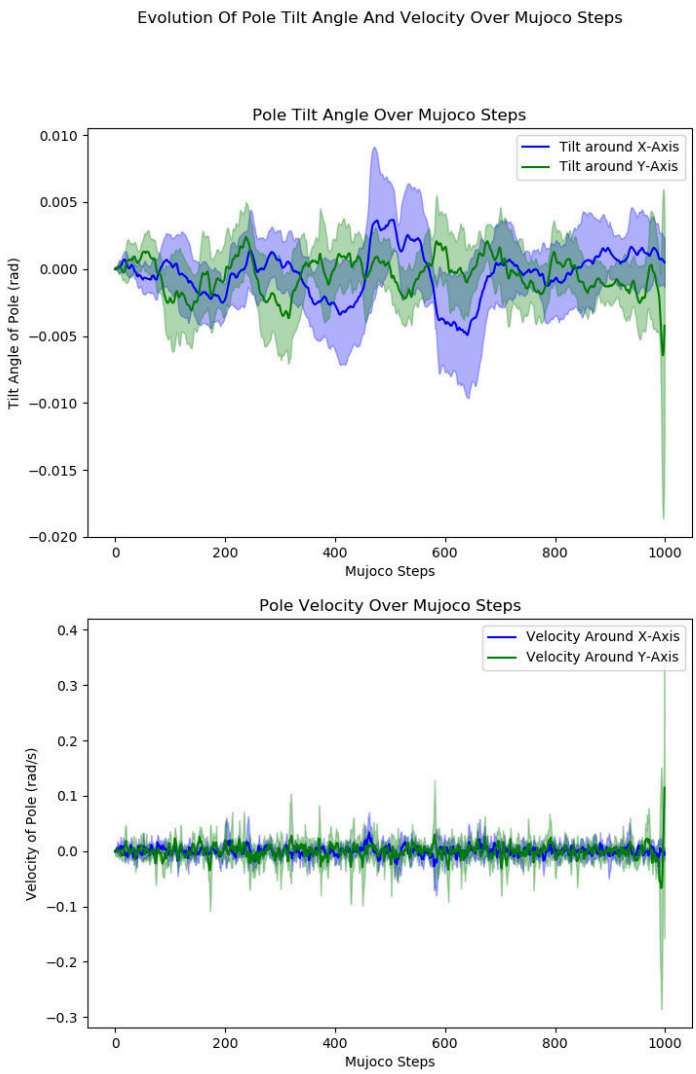


Figure A.21.: Pole tilt angle and velocity – free, noise 0.01

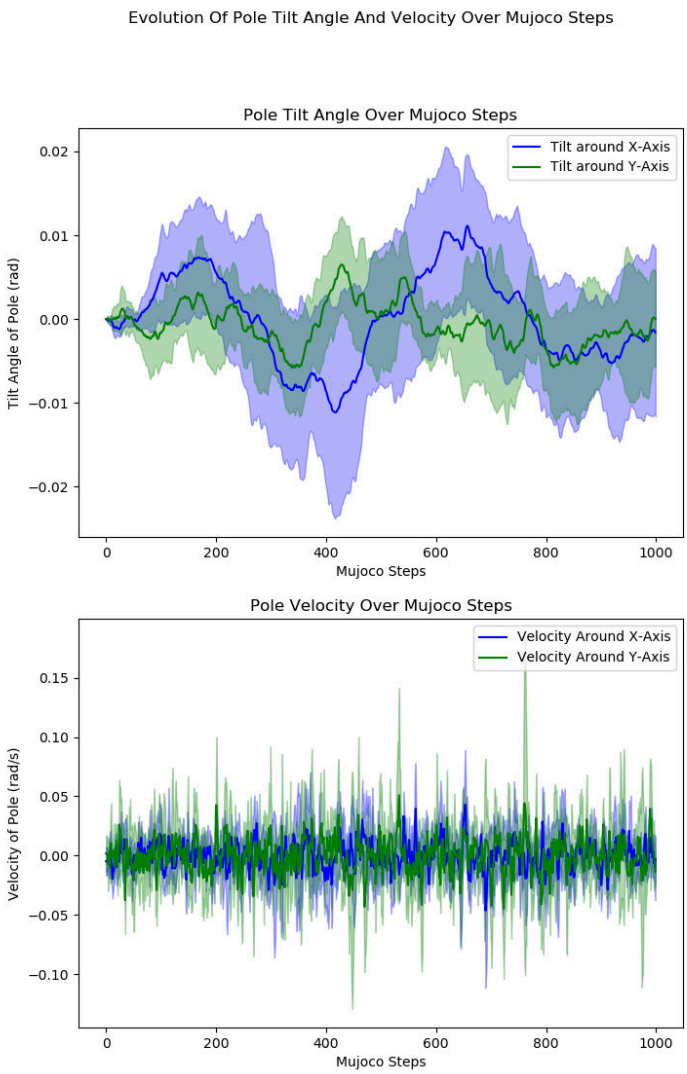


Figure A.22.: Pole tilt angle and velocity – free, noise 0.02

A.3. Implementation details

The pole type, endeffector, physical properties, cable and current modeling are dynamically loaded and compiled by Xacro scripts creating the URDF describing the WAM. The hardware interface defines the access to the robots joints. Our launch file loads the URDF into a robot description parameter, launches nodes responsible for RViz, pole state publishing, tactile sensor logging, and the WAMDriver, for acting on the robot and reading its state. Furthermore, it loads the controller specified on the list.

Before using the WAM for the first time since turning on the electricity, we need to initialize the joint encoders of the WAM by manually moving the WAM to key positions (see WAM-Manual). The robot indicates when all joint encoders are initialized and gets fully visualized in RViz.

To create the 360 degree joint or the tactile sensor housing, we relied on 3-D-printing. 3-D-printing allows for precise control over spatial dimensions, enables customization based on specific needs, and produces lightweight yet stable objects. For 3-D-printing, we used the PrusaSlicer [67, 68]. PrusaSlicer processes 3-D-models (STL, OBJ, AMF) and converts them into G-code instructions executed by a 3-D-printer, in our case the Original Prusa i3 MK3s+. In the print preview settings, parameters such as infill percentage can be adjusted to control the weight and robustness of the object. PrusaSlicer is a free and open-source software stack. For the filament we used Polylactic Acid (PLA) rolls.
We designed the STL files in Onshape [69] and imported them into PrusaSlicer. Onshape is an online, free CAD tool that allows users to model arbitrary shapes by creating and combining basic geometric elements such as rectangles and circles, as well as performing operations like extrusion, Boolean intersection, and edge smoothing.