Learned Legendre Predictive State Estimator for Control

by

Pawel Jaworski

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Systems Design Engineering

Waterloo, Ontario, Canada, 2022

Abstract

This thesis introduces a novel method for system model identification, specifically for state estimation. The method uses a 2 or 3 layer neural network developed and trained with the methods of the Neural Engineering Framework (NEF). Using the NEF allows for direct control of what the different layers represent with white-box modelling of the layers. NEF networks also have the added benefit of being compilable onto neuromorphic hardware, which can run on an order of magnitude or more less power than conventional computing hardware. The first layer of the network is optional and uses a Legendre Delay Network (LDN). The LDN implements a linear operation that performs a mathematically optimal compression of a time series of data, which in this context is the input signal to the network. This allows for temporal information to be encoded and passed into the network. The LDN frames the problem of memory as delaying a signal by some length θ seconds. Using the linear transfer function for a continuous-time delay, $F(s) = e^{-\theta s}$, the LDN compression is considered optimal as it uses Padé approximants to represent the delay, which has been proven optimal for this purpose. The LDN has been shown to outperform other memory cells, such as long short-term memory (LSTM) and gated recurrent units (GRU), by several orders of magnitude, and is capable of representing over 1,000,000 timesteps of data. The LDN forms a polynomial representation of a sliding window of length θ, allowing for a continuous representation of the time series. The second layer uses the Learned Legendre Predictor (LLP) to make predictions of how a subset of the input signal to this layer will evolve over a future window of time. In the case of model estimation, using the system states and control signal (at minimum), the LLP layer predicts how the system states will evolve over a continuous window into the future. The LLP uses a similar time series compression as the LDN, but of the representation of the layer prediction into the future. The weights for the LLP layer can be trained online or offline. The third layer of the network performs the transformation out of the Legendre domain into the units of the input signal to be predicted. Since the second layer outputs a polynomial representation of the state prediction, the state at any time in the prediction window can be extracted with a linear operation. Combined, the three layer network is referred to as the Learned Legendre Predictive State Estimator (LLPSE). The 2 layer version, without LDN context encoding, is tested online on a single link inverted pendulum and is able to predict the angle of the arm 30 timesteps into the future while learning the system dynamics online. The 3 layer LLPSE is trained offline to predict the future position of a simulated quadrotor over a continuous window of 1 second in length. The training, validation, and test data is generated in AirSim with Unreal Engine 4. The LLPSE is able to predict the future second of a simulated quadrotor's position with an average RMSE of 0.0067 on the network's normalized representation space of position (normalized from a

30x30x15 meter volume). Future work is discussed, with initial steps provided for using the LLPSE for model predictive control (MPC). A controller, the Learned Legendre Predictive Controller (LLPC), is designed and tested for state estimation across the control space. The design and future steps of the LLPC are discussed in the final chapter. A preliminary LLPC is designed and was integrated into the test suite, and is available along with all of the code for simulator interfacing, controllers, path planning, the LLP systems, and various utility functions at https://github.com/p3jawors/masters_thesis.

Acknowledgements

I would like to thank all the people who made this thesis possible. Thank you to Dr. Terry Stewart for his support and patience in discussing the Learned Legendre Predictor. Thank you to Dr. Michael Furlong for his support and teaching of the Learned Legendre Predictor and control theory. Thank you to Dr. Travis DeWolf for discussions in debugging control problems and general planning of experiments. A special thank you to Dr. Chris Eliasmith for his continual support and supervision throughout the entirety of the work covered in this thesis. Finally, thank you to my wife, Christina Jaworski, for her constant support and motivation through the passed 2 years. This thesis would not be possible without the dedication and help of these wonderful people.

Dedication

This is dedicated to my wife Christina, my daughter Olivia, and my soon to be born son.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The human brain is the most complex organ in the human body. It is the orchestrator behind everything that humans are able to do. This 3lb organ is capable of simultaneously interpreting the sensory data throughout the body, driving force based motion through control of muscle contraction, calculating multi-joint real-time path planning, and much more. In addition to the immense computational abilities of the human brain, it also acts as a storage for short and long term memories. In contrast, robots cannot compare to the capabilities of the human brain. There is significantly less sensory data available to robots, typically limited to sensors on actuators or cameras. Even with multiple high definition cameras, and dozens of sensors on every actuator, the amount of data available does not compare to the 10 million sensory neurons throughout the human body [19]. One of the key factors that limits the ability of robots is the amount of compute available. Simply modelling the amount of spiking neurons in the brain would take an estimated 10^{18} CPU FLOPS and 10^4 TB of memory [22]. As of June 2022, the Frontier super computer is the fastest super computer in the world and is capable of performing 1.12×10^{18} FLOPS [1]. However, Frontier does so by using a staggering 21 MW of power. In contrast, the human brain is capable of running off of 20W of power [4]. It quickly becomes evident that building large scale dymanical systems on conventional hardware does not begin to compare to the capabilities of the human brain.

Machine learning is the field of study of how systems can learn tasks by leveraging large amounts of data. Engineers use the methods of machine learning to build neural networks that loosely mimic how human brains function. Artificial neural networks (ANNs) are collections of interconnected nodes that are trained to learn a weighted association between a set of input data and expected system outputs. Numerical values are passed along connections to nodes, where they are summed and passed through the neuron's nonlinear function. By adjusting the weighting of the many connections, neural networks can learn large dimensional mappings between a defined input-output space. In recent years, neural networks have had great success in several fields, including model discovery of dynamical systems [5] and model predictive control [14]. Many methods use a heavily data driven, black-box approach, to try and model system dynamics. Although ANNs try to mimic the function mapping of the brain through a process of training, they are still designed to run on conventional hardware. Even if an ANN was designed to perfectly mimic the human brain, it would still be limited by the power budget of modern computers. Part of the reason brains are so efficient is that the billions of neurons performing computations function asynchronously. In contrast, central and graphical processing units (CPUs and GPUs) run on a set clock and use digital logic. To work towards brain level power efficiency and compute, foundational changes need to be made in both the hardware and software used for simulating neural networks.

The question then remains, how does one run programs, or even perform simple computations, using spiking, asynchronous, neural networks? The Neural Engineering Framework (NEF) [10] introduces methods for using spiking neurons to perform mathematical operations, and even model dynamical systems such as oscillators, integrators, and Kalman filters. The NEF is not limited to spiking neurons, as it can be extended to many neuron types, including the ones used in ANNs. It is also capable of constructing white-box systems that perform a predefined function, in addition to the black-box systems that are commonly built with machine learning. By designing systems with the underlying principles of the NEF, networks can be compiled not only onto conventional hardware (CPUs and GPUs), but onto neuromorphic hardware.

CPUs and GPUs use the von Neumann architecture where separate processing and memory units work in tandem, where the processing is aligned on a synchronous clock with a sequential instruction set. In contrast, neuromorphic hardware aims to function more like the human brain, with massively parallel, asynchronous processing where memory and processing are collocated [23]. Early hybrid digital-neuromorphic chips, such as the Loihi 2, have been shown to consume an order of magnitude less power than non-neuromorphic hardware. The Loihi 2 has been claimed to have a 16x improvement in power efficiency[21]. With continual development and the advent of memristor technology [7][17] where memory and compute are collocated in a single node, the power requirements will continue to improve past the order of magnitude already possible with early chips.

In this thesis I consider methods that allow NEF networks to be built and run on neuromorphic hardware. In the long term, theses methods should allow us to more closely approach the efficiencies of the biological brain. In particular, I focus on applying these techniques to the problems in motor control. Since the early 1950's researchers began thinking about how to build artificial brains to control aritificial bodies. Development in the field of robotics has never been held back by the creativity of humans, but by the limitations of the technology at the time. Robotics typically require efficient processing, as sensory feedback needs to be processed, and a control action decided upon, in real-time. As computational limits have increased, so have the abilities of robots over the past ten years. With limited hardware, simple linear controllers can be used to drive the measured states of a robot to a target reference by making control actions that aim to reduce the current error in the system. However, a robotic system would ideally be able to make predictions about the trajectory of its motion, as humans do. This would allow the system to adjust its trajectory to minimize an expected error, instead of making adjustments to control actions only after an error is observed. This would require a dynamical model of the system that would be able to make predictions of how the system states will evolve based on sensory input. Model Predictive Control (MPC) aims to do this, by using a dynamical model to predict the trajectory of its states, and optimizing its control action based on the predictions. However, dynamical models can become computationally taxing, and are also limited by the decisions the modeler makes.

As the world is infinitely complex, and the abilities of hardware are limited, decisions have to be made as to what forces are important to model to sufficiently capture the dynamics of a system. Ideally a system model would be updated online to account for changes in the system or environment. MPC typically begins with a known system model and does not change it while the system is running. In this thesis I develop recent methods in order to allow adaptaton of the system model in an online fashion. Specifically, I extend the Learned Legendre Predictor (LLP), which is a single layer neural network with a novel learning rule [12]. The LLP makes predictions in the Legendre domain, where the outputs are the coefficients of q Legendre polynomials. By using a polynomial representation, the output of the network represents the continuous values of a window of time of length θ. This encoding and decoding is a slight alteration of the Legendre Delay Network (LDN) [30]. The LDN is a mathematically optimal method for continuous time compression of discrete or continuous signals. It has been shown to outperform current methods like transformers and LSTMs by an order of magnitude [6]. The LDN takes any arbitrary signal s as input and creates a sliding window Legendre domain encoding that represents the past θ seconds of s. The LLP alters this encoding, instead learning to output a Legendre domain encoding that represents the *future* θ seconds of a signal. As the purpose of using a dynamical model is to make temporal predictions of a system's states, temporal information about the context provided to the model helps improve the predictions.

The purpose of this thesis is to use the methods of the NEF, LDN, and LLP to design a system capable of predicting the future states of a quadrotor. Beginning with the framework of the NEF allows for the designed system to have the ability of being compiled onto neuromorphic hardware. Building upon the LLP system, LDN encoding of context is added to provide temporal information, and the LLP output is decoded to provide predictions of state. The system proposed here, with the addition of LDN encoding and decoding, is referred to as the Learned Legendre Predictive State Estimator (LLPSE). Due to the added complexity of online learning, the LLP (no temporal encoding of context) is tested online on a simpler dynamical model, a single link inverted pendulum. Using the LLPSE, the dynamics of a simulated quadrotor were learned offline, using the methods of the NEF. Finally, multiple instantiations of the trained LLPSE were tested online on a simulated quadrotor, with a distribution of control context used among them. The multiple instantiations of the LLPSE with varying control context is referred to as the Learned Legendre Predictive Controller (LLPC). The LLPC contains an additional node that uses the predictions of the various LLPSEs and the planned reference to select the control action that was used as context for the predictor with the lowest expected error. Due to time constraints, the LLPC was not used in the task of selecting a control action, but rather only for making multiple predictions of state. This was done to test if the dynamical model learned was capable of reproducing the input-output mapping between rotor actions and the expected directions of motion.

In the remainder of this thesis, I describe the background, design, and results of these algorithms. The second chapter begins with an overview of the control methods covered in this thesis. An overview of how quadrotors move is given to provide an intuition for the mapping between control actions and direction of motion. A summary of the dynamical equations of motion and MPC are covered to show the complexity of modelling system dynamics in contrast to the proposed method. As neural networks require data to train, a summary of the linear controller used in simulation to generate the control actions used for training is covered. Chapter 3 describes the details of the neural methods used for the LLPSE, and an overview of neural networks. Chapter 4 introduces the simulation environment and the system designed for generating training data. Chapter 5 explains the details in the design of the LLP, LLPSE, and the alterations to the LDN encoding required for generating the ground truth for offline training. Results are shown in chapter 6, covering the online learning of the LLP on the inverted pendulum, a scan of the representational error of the various LDN encodings used, the results from the trained LLPSE, and the state estimations of the trained LLPC running online with a simulated quadrotor in the loop. Details of the LLPC and next steps are covered in chapter 7 on Future Work.

Chapter 2

Control Methods

With the goal of learning the underlying system dynamics of a quadrotor, how quadrotors move through space is covered first. Having an intuition of how a drone's rotors movement lead to movement through the world helps in understanding the underlying control mechanics. A simple feedback control method is discussed to show how a basic control loop can is designed by reacting to system error. This kind of feedback controller is used to generate training data for the LLPSE and LLPC in Section 4.2. To qualitatively compare to the complexity of current predictive control methods, an explanation of drone dynamics is also provided. The forces and torques that act upon the drone are discussed in relation to how they cause linear and angular translation. The Newton-Euler equations of motion are then used to describe how the system dynamics are derived. The chapter ends with an explanation of how Model Predictive Control (MPC) uses the dynamical model, in conjunction with an optimizer to select optimal control strategies.

2.1 Quadrotor Motion

When defining the motion of a system, the number of directions it can move in are referred to as the degrees of freedom (DOF). For a multirotor this includes three translational directions (x, y, z in cartesian coordinates), and three rotational directions (α, β, γ in euler angles). To control all 6 DOF individually, a drone would require 6 actuators, one for each DOF. Multirotors can have any number of actuators [34], but are most commonly found with 4. Quadrotors, having only 4 rotors, are under-actuated systems. They have more DOF of motion than actuators to control them. A way around this control limitation is to couple the motion between DOF. This way one actuator can be used to control the motion of multiple DOF. However, only one of the two coupled dimensions can be controlled at any one time. Additionally, without external forces, motion in one coupled DOF will always result in the motion in the other. Defined in the body frame, motion forward and back (x) is accomplished by pitching (β) the front of the drone up and down. Motion right and left (y) is accomplished by rolling (α) right and left. By coupling x and β, and y and α, quadcopters can still fly in all 6 DOF with 4 actuators, but can only maintain control of 4 at any one time. Adjustments to the controllable DOF (pitch, roll, yaw, and thrust) is accomplished by adjusting the rotor speeds.

The force required for motion is obtained through the resulting downward thrust from the spinning rotors. Similarly to how the shape of a wing creates lift in airplanes, the shape of a rotor creates a volume of high pressure below it as it spins. As the rotor speed increases, so does the amount of air mass being moved, resulting in more downward thrust. If all four rotors are rotated at the same rate and slowly increased, the drone will eventually take off when the downward thrust counteracts the force of gravity. To prevent the drone from spinning due to inertial effects, rotors are placed in opposing rotational directions as in figure 2.1.

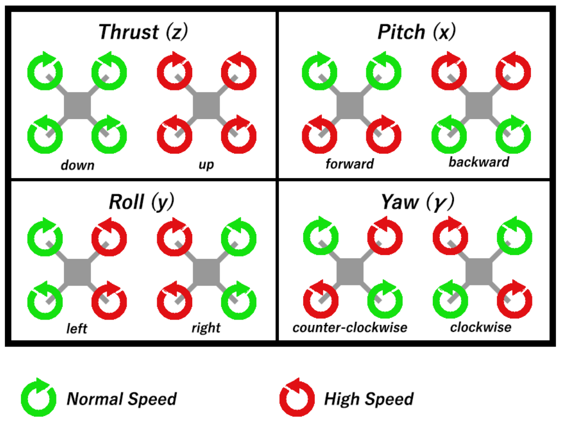


Figure 2.1: The rotor pairs and relative rotation speed used to achieve motion along x, y, z, and γ. The direction of rotation does not change, but the relative speed of rotor pairs does.

By varying the rotor speeds in pairs, thrust can be applied in different directions with respect to the world frame. With respect to the body frame, the thrust vector is always perpendicular to the quadrotor's body. When controlling aerospace systems, coordinates are typically given in the body frame, with the origin at the center of mass of the craft.

Called the north-east-down (NED) coordinate system, and often referred to as the body frame, this coordinate frame makes control more intuitive from the reference point of a pilot. Following the right-hand rule, it defines positive x as being in the aircraft's forward direction, positive y the right direction, and positive z pointing down towards the ground. Despite the peculiarity of defining $+z$ as down towards the ground, it allows for a more intuitive way to adjust the controllable DOF. By transforming from the world to the body coordinate system, increasing the roll dimension results in a clockwise rotation, increasing pitch brings the craft's nose up, and increasing yaw turn towards the right, all with respect to center of mass.

Although controlling pitch, roll, yaw, and thrust is more intuitive in the body frame, it becomes difficult to define target directions of motion. Figure 2.1 shows how pairing certain rotors can result in motion between the coupled DOF. This allows quadrotors to move forward/backward, right/left, up/down, and rotate clockwise/counterclockwise by adjusting the pitch, roll, yaw, and thrust. Using the rotor velocities required to hover as a baseline, a quadrotor's direction of motion can be shifted by speeding up and slowing down rotor pairs as in figure 2.1. The directions of rotation do not change, but the speed of rotor pairs does relative to an equilibrium point, usually the hover point. Rotors with a positive offset spin faster, resulting in the application of a larger force compared to rotors with a negative offset. This imbalanced force causes the thrust vector to have components along x and y, causing the drone to pitch and roll accordingly. Controlling the rotor velocities themselves to result in the desired roll, pitch, yaw, and thrust requires some form of feedback control. Comparing the measurable drone states ($\mathbf{x}(t)$) to the desired target reference ($\mathbf{r}(t)$), a feedback controller can modulate the rotor velocities to drive the system towards the reference.

2.2 Feedback Control

Through the use of feedback, a controller can be designed to bring the measured states towards a target reference by reacting to a calculated error. The error $\mathbf{e}(t)$ is calculated by comparing the measured states $\mathbf{x}(t)$ to the reference $\mathbf{r}(t)$, typically by taking the difference to preserve the sign of the error. The sign of $\mathbf{e}(t)$ signifies in which direction the error is. The larger the calculated error is, the larger the control signal response will be. With the goal of bringing $\mathbf{e}(t)$ to zero, a feedback controller can drive the system's states towards the reference without the need for a complicated dynamical model. The block diagram of a generic feedback controller (figure 2.2) shows the flow of information between the controller and the system.

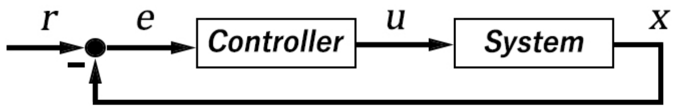


Figure 2.2: The block diagram of a feedback control loop. The control signal, $\boldsymbol{u}$, is shifted based on the error, $\boldsymbol{e}$, between system feedback, $\boldsymbol{x}$, and a target reference, $\boldsymbol{r}$.

Very commonly used, a Proportional-Integral-Derivative (PID) controller modulates the control signal $\mathbf{u}(t)$ by scaling various errors with gain terms. The gain terms add additional scaling to the control response. By varying the relative size of gains, errors in certain states can be weighed more or less heavily depending on their importance.

$$\mathbf{u}(t) = \mathbf{k_p}\mathbf{e}(t) + \mathbf{k_i}\int \mathbf{e}(t)\,dt \ + \mathbf{k_d}\frac{d\mathbf{e}(t)}{dt} \tag{2.1}$$

Equation 2.1 shows the generic form of a PID controller. $\mathbf{e}(t)$ is the error vector between the measured states and the target reference states, and $\mathbf{k_p}$, $\mathbf{k_i}$, and $\mathbf{k_d}$ are the proportional, integral, and derivative gain constants, respectively. To allow for a finer tuning of the controller, unique gain values can be used for each measured state. Determining the value of the gain terms takes careful tuning to create a feedback loop that will bring the system to its target state. For example, if the proportional gain is made too large then the system will overshoot the target. Pushed too far the system response will begin oscillating about the reference, possibly to the point of instability. If the gain is too low the system response will be slow, and unresponsive to disturbances.

Although written together in equation 2.1, any combination of P, I, and D can be used depending on the control scenario. P control aims to minimize the absolute error to the reference. The larger the absolute error the larger the system response will be. I control accumulates system error by taking the integral of $\mathbf{e}(t)$. This allows I control to account for steady-state errors. D control uses the derivative of $\mathbf{e}(t)$ to try to compensate for the system momentum. A Proportional controller alone is capable of bringing a system towards its target reference, but will have issues accounting for system momentum. Combining P and D terms to form a Proportional-Derivative (PD) controller will help to minimize overshooting the reference. PD control is very common and performs well when there are no external forces acting upon the system. However, if there are any steady state errors due to unmodelled disturbances, a PD controller would not be able reach its target state.

Gains are typically held constant while a system is running under any combination of P, I, or D control. The gains are tuned ahead of time and kept constant at run time. If an external force was introduced causing an unmodelled perturbation, such as an additional mass added to a quadrotor, it would start undershooting its target state along the direction of error. In the case of additional mass on a quadrotor, this would result in it undershooting its target along the direction of gravity. The PD gains could be adjusted to compensate for this new mass, but any change to the system dynamics would require a retuning of gains. Alternatively, an Integral controller can be added to account for these constant errors, making a PID controller. The integral term works by applying the gain term to a running sum of the state error. The P and D portions work to drive the system towards a steady state near the target. Given enough time, the error accumulating in the I term will drive the system the remainder of the way to the target state.

Through the use of feedback control, any combination of P, I, or D control can drive a system's states towards a reference in a reactive way. It does so based on the amount of error between the system states and the reference. The system response can be tuned by adjusting P, I, and D gains. In simpler scenarios where there are no external forces acting upon the system, or where a higher tolerance of error is acceptable, the additional complexity of tuning an Integral term can be omitted. Regardless of the combination of P, I, or D control, the controller designed will inherently be reactive as it only makes changes based on the current error. In an ideal case a controller would be able to make some form of prediction about its future states. In this way, a proactive controller could be designed with the aim of accounting for errors before they occur. For this control design, a dynamical model of how the system evolves over time would be required. Given some input of system states and a control signal, the system model could be used to predict how the system states would evolve over time. Comparing the predicted states to the planned future reference, the controller could begin making control changes proactively to mitigate the future predicted error.

2.3 Quadrotor System Dynamics

A model of system dynamics allows for a more robust form of control. A model allows for predictions to be made based on the system starting conditions and any additional external forces. Feedback control alone cannot adapt to unknown perturbations, or make predictions to proactively adjust control signals. In [14] a nonlinear model of drone dynamics was capable of adapting to unknown winds and payloads, while maintaining nimble flight of 70km/h. In [8] a nonlinear model of a robot arm was capable of quickly adapting to unmodelled short term perturbations like additional mass, as well as long term effects like years of simulated wear and tear. When discussing drone dynamics, it helps to use both the body and world frame, as used in section 2.1. However, a way to transform between the two coordinate systems becomes necessary.

$$\mathbf{R} = \begin{bmatrix} C(\gamma)C(\beta) & C(\gamma)S(\beta)S(\alpha) - S(\gamma)C(\alpha) & C(\gamma)S(\beta)C(\alpha) + S(\gamma)S(\alpha) \\ S(\gamma)C(\beta) & S(\gamma)S(\beta)S(\alpha) + C(\gamma)C(\alpha) & S(\gamma)S(\beta)C(\alpha) - C(\gamma)S(\alpha) \\ -S(\beta) & C(\beta)S(\alpha) & C(\beta)C(\alpha) \end{bmatrix} \tag{2.2}$$

$\mathbf{R}$ is the rotation matrix to shift from the body frame to the world frame, where C and S are the cosine and sine functions. To transform from the world frame to the body frame, the inverse of $\mathbf{R}$ is used. However, since $\mathbf{R}$ is an orthogonal matrix $\mathbf{R}^{-1} = \mathbf{R}^T$. The angles α, β, and γ are the rotation angles about the x, y, and z axes, respectively. In the body frame this is rotation about the body locked coordinate system, where rotation about x is referred to as roll, rotation about y pitch, and rotation about z yaw. When deriving the dynamics, shifting the body frame's forward direction to be aligned with one of the drone arms, as in figure 2.3, creates a symmetry that simplifies the equations. With the transform between reference frames, the forces of the system can be examined next.

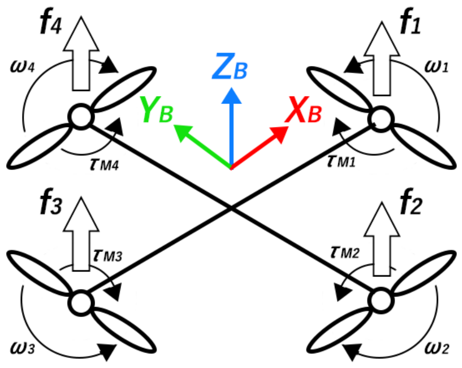


Figure 2.3: The body frame of a quadrotor. The body frame can be defined in many ways, but aligning the cardinal axes with the drone arms leads to a symmetry that simplifies the derivation of control dynamics.

Following a similar derivation of drone dynamics as shown in [16] [18] [20], the definition of the system begins from defining the net forces and torques acting on the system. Beginning with the controllable forces, as described in section 2.1, the spinning actuators of a drone apply a torque that spins the rotors. As they spin, the shape of the rotors causes a downward displacement of air, resulting in the reactive upward force acting on the body of the drone. Figure 2.3 shows the directions of the forces and torques relative to the body frame. Assuming a symmetrical quadrotor with equally spaced actuators, the force and torque at each rotor, i, can be defined as

$$f_i = k\omega_i^2 \tag{2.3}$$

$$\tau_{M_i} = b\omega_i^2 + \boldsymbol{I_M}\dot{\omega}_i \tag{2.4}$$

where f_i is the force at rotor i, τ_{M_i} is the motor torque applied to the rotor, and k and b are the lift and drag constants, respectively. The constants are determined from the drone's mass and properties of the rotors themselves. $\boldsymbol{\omega}$ is the angular velocity vector of the rotors and contains the controllable variables in the system, the rotor speeds. Usually the effects of $\dot{\boldsymbol{\omega}}$ are considered to be small and can be ignored [18]. Due to the symmetry of the system in the body frame, the moment of inertia tensor, $\boldsymbol{I_M}$, becomes diagonal and can be written as

$$\boldsymbol{I_M} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{2.5}$$

The moment of inertia tensor describes how the angular momentum of the system body changes about the center of mass. The diagonal components I_{xx}, I_{yy}, and I_{zz} are the moments of inertia about the x, y, and z axes, respectively. The off-axis components are the product of moments between axes. Due to the symmetry of the system, all-off axis components are equal to zero. The net force and torque on the system resultant from the spinning rotors can then be defined as

$$\boldsymbol{f_B} = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}; T = \sum_{i=1}^{4} f_i = k\sum_{i=1}^{4} w_i^2 \tag{2.6}$$

$$\boldsymbol{\tau_B} = \begin{bmatrix} \tau_\alpha \\ \tau_\beta \\ \tau_\gamma \end{bmatrix} = \begin{bmatrix} kl(-w_2^2 + w_4^2) \\ kl(-w_1^2 + w_3^2) \\ \sum_{i=1}^{4} \tau_{M_i} \end{bmatrix} \tag{2.7}$$

The sum of the four rotor forces is the total system thrust T. In the body frame the thrust is along the z axis only, so the force vector $\boldsymbol{f_B}$ has zero x and y components. The system torque in the body frame is defined by τ_α, τ_β, and τ_γ, which are the torques along the roll, pitch, and yaw directions. With the body frame defined as in 2.3, a torque in the roll direction is obtained by increasing ω_4 and decreasing ω_2. Similarly, torque about the pitch axis is obtained by increasing ω_3 and decreasing ω_1. Torque along roll and pitch are also governed by the moment arm, l, which is the length between the center of mass and any of the four evenly spaced rotors. The axis about which yaw rotation occurs is parallel to the rotation axes of the rotors. Torque about the yaw axis is therefor governed by the sum torque of the four rotors, τ_M. Since the drone body does not contain any flexible components, the Newton-Euler equations can be used to derive the system dynamical equations.

The Newton-Euler equations are the combined translational and rotational dynamics for rigid bodied systems. They combine Euler's two Laws of motion:

1. A change in linear momentum of a rigid body is equal to the sum of all external forces acting on the body.

$$m\dot{\boldsymbol{v}} = \sum_{i=1}^{N} \boldsymbol{f_i} \tag{2.8}$$

   where the linear velocity vector $\boldsymbol{v} = [\dot{x}, \dot{y}, \dot{z}]$, m is the system mass, and $\boldsymbol{f_i}$ are the forces acting on the system.

2. A change in angular momentum about a fixed point in the world frame is the sum of all external torques acting on the body.

$$\boldsymbol{I_M}\dot{\boldsymbol{\omega}}_{\boldsymbol{B}} + \boldsymbol{\omega_B} \times (\boldsymbol{I_M}\boldsymbol{\omega_B}) = \boldsymbol{\tau_B} \tag{2.9}$$

   where $\boldsymbol{\omega_B}$ are the angular velocities in the body frame.

Using Euler's first law, the translational dynamics are obtained by substituting the rotor forces (equation 2.6) and any external forces acting upon the drone into equation 2.8. A trade-off can be made between model accuracy and complexity. Modelling more forces provides a more accurate description of the system, but at higher computational cost. Some forces can also be difficult to model, such as blade flapping and turbulent air flow. Limiting external forces to gravity and drag, as in [18], and dividing the system mass through yields

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix} = \frac{1}{m}\boldsymbol{f_B}\mathbf{R} - \frac{1}{m}\boldsymbol{D}\dot{\mathbf{x}} - \boldsymbol{g} \tag{2.10}$$

where $\boldsymbol{f_B}$ is the sum force of the rotors in the body frame. The transformation matrix $\boldsymbol{R}$ is used to transform this into the world frame. $\boldsymbol{D}$ is the diagonal drag matrix. The diagonal components d_x, d_y, and d_z are the drag force coefficients in the corresponding world frame directions. The values of the coefficients are determined empirically as in [18]. The drag coefficients tell how much the system resists motion given the linear velocity, $\dot{\boldsymbol{x}}$. $\boldsymbol{g}$ is the acceleration vector due to gravity and on earth is $[0, 0, 9.81] m/s^2$, in the world frame.

The rotational dynamics are defined using Euler's second law of motion. Substituting equation 2.7 into 2.9, rearranging for angular acceleration, and expanding the cross product yields

$$\begin{bmatrix}\ddot{\alpha}\\ \ddot{\beta}\\ \ddot{\gamma}\end{bmatrix} = \begin{bmatrix}\tau_\alpha/I_{xx}\\ \tau_\beta/I_{yy}\\ \tau_\gamma/I_{zz}\end{bmatrix} + \begin{bmatrix}(I_{yy}-I_{zz})\dot{\beta}\dot{\gamma}/I_{xx}\\ (I_{zz}-I_{xx})\dot{\alpha}\dot{\gamma}/I_{yy}\\ (I_{xx}-I_{yyt})\dot{\alpha}\dot{\beta}/I_{zz}\end{bmatrix} \tag{2.11}$$

Provided with an initial condition $[x, y, z, \alpha, \beta, \gamma]$, the linear and angular dynamical equations can be used to predict how the system's states will change. With the underlying knowledge of how the physical forces and torques act on the system, solutions can be found analytically. Leveraging the ability to estimate the evolution of a system's motion over time, a dynamical model can be used to develop a predictive control scheme.

2.4 Model Predictive Control

One problem with feedback control, including PD and PID control, is that it is inherently reactive. The control law moves the system towards a target state by reacting to the current level of error. An alternative to Feedback control, is Model Predictive Control (MPC). MPC covers an advanced set of control methods that base their control law on predictions of how the system will behave in the near future. Using the dynamical model of the system, a simulation is run making predictions of how the system's states will change. The predictions are then used to make a proactive controller where control decisions are made not just by the current error, but with the predicted future error. However, solving these higher order differential equations is computationally taxing.

For real-time robotics, it is not always feasible to solve for the complex system dynamics at a rate fast enough for reliable control. The purpose of the dynamical equations is to step through time to see how the system states evolve. As the step size between system measurements gets smaller, the changes in the system's response begin to look linear. With the assumption that measurements can be made at a fast rate relative to the system dynamics, a linear approximation can be formed. Through a process of variable substitutions the higher order differential equations can be repackaged into a set of first order differential equations. A full derivation of the linear system is out of scope with regards to the focus of this thesis, but an example can be found in [27]. In control theory this representation is called the state space representation of a system. Although state space equations can be nonlinear, the state space representation given in equation 2.12 is in the linear form:

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \tag{2.12}$$

where $\boldsymbol{x}$ is a vector of the system states and $\boldsymbol{u}$ is a vector of the control signals. The way the system's states change relative to one another is encompassed in the dynamics matrix $\boldsymbol{A}$. $\boldsymbol{A}$ is calculated by taking the derivatives of the system of first order equations with respect to the system states $\boldsymbol{x}$. Similarly, $\boldsymbol{B}$ is the input matrix that encompasses the derivatives of how the states change relative to the manipulable control variables $\mathbf{u}$. With a linearized system model, it becomes less computationally taxing to perform predictions for use in generating control actions.

Since equation 2.12 has the change in control $\boldsymbol{u}$ as an input, different system trajectories can be calculated depending on the change in $\boldsymbol{u}$. In an ideal scenario predictions would be made across the entire possible control space. With all the possible control actions simulated, an optimal one can be selected given some criteria of optimality. MPC does so by introducing an optimizer and model block into its control loop as shown in figure 2.4. The combination of predictions and optimizations is what separates MPC from other control methods, in contrast to methods where the control laws are pre-calculated, such as PD or PID control where gains are left constant at run-time.

There are many methods for implementing MPC. Model Predictive Path Integrals (MPPI) are based on the stochastic sampling of all possible trajectories. By transforming the optimization problem with the Feynman-Kac lemma, the optimizaton can be solved for across a control window [33]. For the simplicity of illustration, the greedy approach will be used in explaining the key principles of MPC. Using the feedback of system states $\boldsymbol{x}(t)$, the MPC model block makes a prediction of where $\boldsymbol{x}(t + \Delta t)$ will be. By comparing the range of possible trajectories against the $\boldsymbol{r}(t + \Delta t)$ the optimization block selects the best control action $\boldsymbol{u}(t + \Delta t)$. The selected control action and state predictions are then

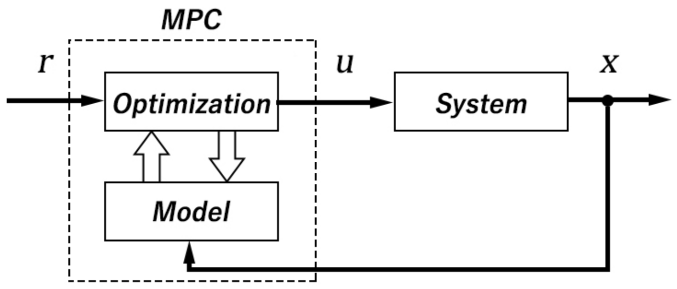


Figure 2.4: The MPC control loop. The MPC block contains the system model generating the predictions of the system dynamics, and an optimizer block that selects a control strategy given the constraints of the optimizer.

fed back into the model. This model simulation and optimization is performed iteratively over the prediction horizon, t_p, as shown in figure 2.5.

Once the predictions over t_p are complete, the MPC block only uses the first optimal control action $\boldsymbol{u}(t + \Delta t)$ to control the system. The prediction and optimization process is rerun over the entire prediction window for every control step. This is why MPC is also referred to as receding horizon control [24]. The main premise of MPC is that these short term optimizations will lead to long-term optimality. By introducing optimality into the controller, MPC also allows for the use of cost functions. The optimizer block of MPC works by trying to minimize a quadratic cost function, J, towards a predefined criterion. Consider the cost function:

$$J(t) = \sum_{j=1}^{N} \delta(j)(\boldsymbol{x}(t + j\Delta t) - \boldsymbol{r}(t + j\Delta t))^2 + \sum_{j=0}^{N-1} \lambda(j)\boldsymbol{u}(t + j\Delta t)^2 + \sum_{j=0}^{N-1} \psi(j)\Delta\boldsymbol{u}(t + j\Delta t)^2 \tag{2.13}$$

By introducing cost functions MPC can set hard constraints, which becomes useful for real systems with limitations in energy and system response. Where a feedback controller can output physically impossible control actions, MPC can set limitations on how quickly a control signal can change, or how large it can become. The cost function in equation 2.13 can be broken down into three terms. The first summation works to drive the system's states towards the reference. $\delta(j)$ is a scaling factor for weighing the importance of how quickly the system converges to its reference relative to the prediction step, j. The second summation works towards minimizing total energy expenditure. By minimizing the control

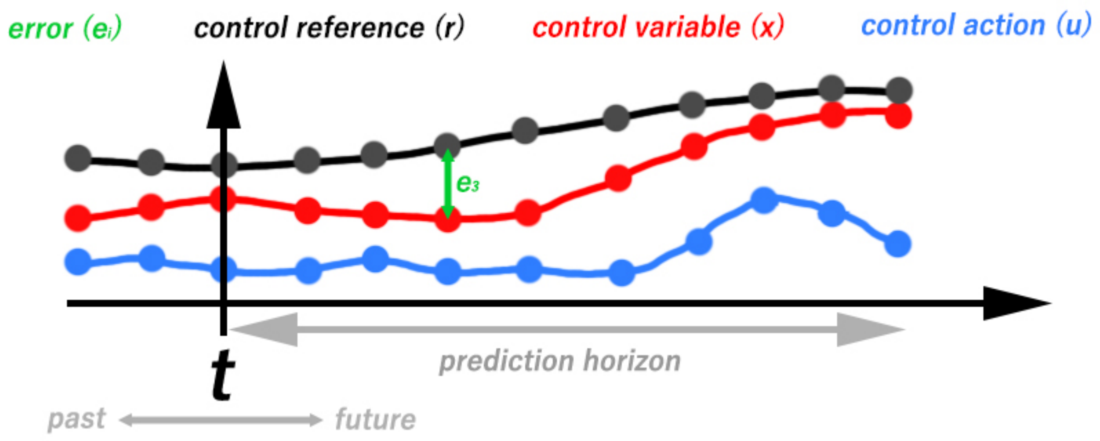


Figure 2.5: The prediction horizon of MPC. The past states and control actions, as well as the past and future reference trajectories are known. Using the known system parameters, the model runs iteratively making a prediction of where the system will move given various control actions. The control actions are typically chosen from a distribution, with the optimal strategy used for the next time step.

action, the amount of energy the system uses can be optimized. Similarly, $\lambda(j)$ is a scaling factor for the relative importance of this term. The final summation limits the step size of the control action, with ψ being the weighting factor. N is the number of discrete steps to make predictions for. The control sections in the cost function therefore go from $j = [0, N-1]$ as the last control action is not used for state prediction. The state section of the cost function goes from $j = [1, N]$ since the 0th state is provided through the system feedback.

With MPC, control actions can be made with some underlying measure of optimality. With the addition of constraints and state prediction, MPC separates itself from simple feedback control. However, due to the added complexity, fast simulations are vital as predictions over the entire predicition horizon are recalculated during every time step. Linearizing the dynamical equations helps in speeding up the model predictions. The process of defining the system dynamics can also be difficult and requires knowledge of the environment ahead of time. Additionally, the modeler has to decide between model complexity and resource requirements. If a force is omitted from the model, the control law would have no way to compensate for it in its prediction. Ideally, the system dynamics could be learned and then used for prediction and control. Additionally, the state prediction only uses the state and control for a single moment in time. Ideally, a history of context could be used to improve the prediction. Having some representation of state and control history would also inherently have some information of the respective derivatives due to the nature of being a signal over time. This would include important information such as system momentum and previous directions of motion and control. In this thesis, the use of the Neural Engineering Framework (NEF) for training, Legendre Delay Network (LDN) for temporal context encoding, and Learned Legendre Predictor (LLP) for prediction of state over a time horizon, are explored to address the limitations of traditional methods. In contrast to the methods covered, the NEF allows for low power models to be run on neuromorphic hardware, increasing the limits of what systems can be modelled. The LDN provides temporal context to be used as feedback with an optimally compressed representation. Finally, the LLP learns, and can adapt, system models online, removing the need for detailed models.

Chapter 3

Neural Methods

This chapter covers an overview of the neural methods used in designing and developing a state predictor. Starting with artificial neural networks (ANN) and how they are used to learn complex dynamical equations, the underlying princples of ANNs are covered. Neural networks have been used for system identfication [5] and control [3], but do so with training data generated from a baseline controller. Ideally these controllers would have their dynamics directly mapped onto neural networks, instead of the network learning a model of best fit to the training data. The subsequent chapter covers how the Neural Engineering Framework (NEF) can be used to map such models directly onto neural networks. With the goal of also embedding temporal context into the input data, the Legendre Memory Unit (LMU), is discussed. The LMU provides an optimal compression of a signal, and outperforms other memory units, such as long short term memory (LSTM) or gated recurrent units (GRU) by orders of magnitude [29]. Building off the principles of the NEF and LMU, the Learned Legendre Predictor (LLP) provides an online learning method for predicting the future window of a (sub)set of network input. With the training methods of the NEF, the optimal time compression of the LDN, and the LLP's ability to predict future context, the base elements of the LLPSE and LLPC designed in this thesis are covered.

3.1 Neural Networks

Researchers have had great success in recent years building neural network systems capable of beating some of the world's best human players in various games. IBM's Watson won in Jeopardy in 2011, beating two players who were considered the best at the time. The rate of progress only continued to increase as DeepMind began creating neural networks that could play various Atari games in 2013, beating the World's greatest Go player in 2017 with AlphaGo, leading to AlphaStar in 2019, which learned to play the real-time strategy game StarCraft, earning a ranking in the top 0.2% of human players. Despite the complexity of the tasks these networks can perform, the underlying computational unit is a fairly intuitive approximation of a neuron, as shown in figure 3.1.

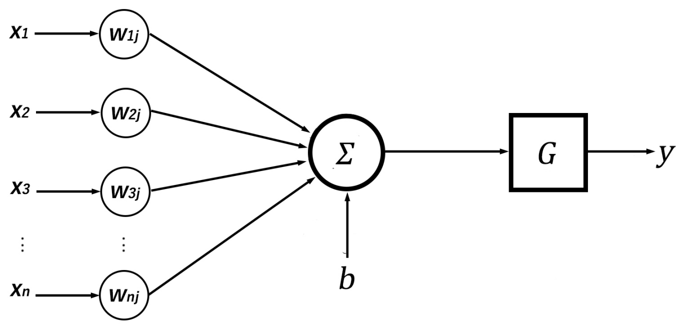


Figure 3.1: Input and output flow of data for a single neuron. Connected neurons pass their output value x_i, each of which gets scaled by their respective connection weight w_{ij}, where i is the neuron index in a layer, and j is the layer index. A bias b is added to the weighted summation of inputs before being sent into the neuron non-linear transfer function, G. This signal propagation continues down the network chain to subsequent neural layers.

Each neuron is treated as a nonlinear function that receives inputs from other neurons, similarly to how neurons are connected in the brain and send surges of voltage down their axons. In artificial neural networks (ANN) the input to each neuron is a numerical value on which the neuron performs a nonlinear operation. The particular nonlinear operation, often called the 'transfer function', can vary, and is chosen by the modeler. For neurons at the input layer of the network, the input values are provided by the modeler. For neurons not in the input layer, the input values are calculated by weighting and summing the outputs of all of the neurons connected to the given neuron, and then adding a bias, as given by

$$y = G(\boldsymbol{x} \cdot \boldsymbol{w} + b) \tag{3.1}$$

where G is the nonlinear transfer function of the neuron, $\boldsymbol{x}$ is the vector of input values, $\boldsymbol{w}$ is the vector of connection weights, and b the bias, typically randomly selected. y is the output of a neuron, and is the value that is passed to the neurons it is connected to.

By changing the values of the connection weights, networks can approximate any function given enough neurons [15].

Networks are typically organized with groups of neurons stacked in layers. If the number of layers exceeds 2 it is considered a deep network. The output of each neuron in a layer can be connected to subsequent layers sparsely (i.e., to only a few neurons), densely (i.e., to most or all of the neurons), or in a recurrent manner (i.e., to other neurons in the same layer, or earlier in the hierarchy). The majority of neural networks are trained through a process of error backpropogation through the network. With the connection weights initialized randomly, or to zeros, the network is provided with some task inputs, outputs an answer, and is then evaluated. Based on this evaluation, the error is then backpropagated through the network and used to adjust the connection weights. The weights are altered through gradient descent until a desired level of error is met. Because this kind of supervised training does not provide an understanding of what each individual layer is doing, the resulting neural network can be classified as a black-box system. This makes training the layers in a network and determining their various hyperparameters a laborious process, typically requiring a sweep across a parameter space using a search algorithm such as grid search, Tree-structured Parzen Estimator (TPE), etc.

The training of such a network is typically broken into a training, validation and test phase. The data collected is broken up between the three categories. Training data is used during the backpropagation phase where connection weights are adjusted. Validation data is used to evaluate how well the network is performing on new data that was not used for adjusting weights. Based on the validation results, the parameters of the training regime can be adjusted. During the test phase, it is important that the data used has never been shown to the network during training or validation. The performance of neural networks heavily depends on the quality of the data used for training. When training in a supervised manner, data consists of system inputs, and expected outputs, which are commonly referred to as the 'ground truth'. The data used for training should cover as much of the input space as possible. Backpropagation tries to obtain a best fit between input data and the corresponding ground truth. What the network ultimately learns is a mapping between its input and output space. Although networks can interpolate, if the data used to train the network omits an area of input space that varies from the space trained on, the network typically begins to quickly make errors as inputs become 'out of sample' compared to the training data, having never seen statistically similar examples of the mapping. Additionally, the training data should not be biased. For example, a network could be tasked with learning to identify noses. If all of the training data consisted of images with faces centered in the frame, the network may seem to perform well initially. However, once new data is used where faces are no longer centered, it may be realized that the network just learned to identify the center of images instead.

Through this method of training, control systems can be approximated by providing a network with large amounts of training data and an offline training regime. The final system then performs an inference task when running online. For a controller neural network, it would minimally take system states and a reference as input, and output a resulting control signal, akin to a feedback controller. When training a neural network to learn the dynamics of a system, some baseline system is usually required to create the training data. The data can be recorded from real systems, or simulated ones. In either case, this usually requires a controller to already be established to move the system through its input space while data is collected. Although other training regimes are also used [3][13], this thesis will focus on the use of supervised and online learning. Unlike supervised learning, online learning aims to minimize an error signal by adjusting network weights at run time. This is in contrast to supervised learning where the weights are adjusted with backpropagation during the training phase, and kept frozen during inference. Online learning usually works by using the Delta rule, and will be discussed section 3.4.

In many scenarios it is known how to develop a good controller. It may then seem strange to design a controller only to sample its inputs and outputs to train a neural network. The network is ultimately trained using a black box method to learn those dynamical equations. It may also underperform when compared to the original controller if the training data did not cover the complete state space. It would be much better to directly embed the original controller into a neural network. One set of methods for doing that is the Neural Engineering Framework (NEF).

3.2 Neural Engineering Framework

The Neural Engineering Framework (NEF) [10] provides a set of tools for developing cognitive and control systems that can be black-box, white-box, or a hybrid. Where typical deep networks have many stacked layers and use an iterative optimization process like gradient descent, the NEF allows for modelling specific functions in each of the layers. From a high level view, the NEF characterizes layers of neurons as a resource for representing numerical values with connections used to compute functions on their represented value. Functions can be mapped onto these connections through a least-squares regression using closed form functions, or using a provided set of evaluation and target points. However, the method is not limited to simple feed-forward connections, as it can also be extended to recurrent connections. Unlike typical ANNs, when developed with the NEF, connections between neurons model the signal propagation over time with a synaptic filter. The synaptic filter acts as a low pass filter to smooth out neural spikes, and additionally allows for the modelling of dynamical systems such as Kalman filters, oscillators, and integrators.

Inspired by the brain and developed by theoretical neuroscientists, the NEF takes many design cues from how the human brain works. Neurons in the brain fire at differing rates depending on the type of inputs they receive. Similarly, the NEF allows for a similarity measure between the inputs a neuron receives and the inputs that cause it to spike most. In contrast to equation 3.1 for ANNs, the NEF defines a neuron's computation as

$$a_i = G(\alpha_i \cdot \boldsymbol{e_i} \cdot \boldsymbol{x} + b_i) \tag{3.2}$$

Similarly to ANNs, G is the neural nonlinearity, $\boldsymbol{x}$ is the input vector, and b_i is the bias for neuron i. The encoding vector $\boldsymbol{e_i}$ defines the 'preferred' direction of the neuron. The more closely the input vector matches the preferred direction of the neuron, the faster it spikes. a_i is the resulting spike rate. Additionally, the NEF also has a gain term α_i that scales the similarity metric. Figure 3.2 shows the neural response, $\boldsymbol{a}$, of a group of leaky-integrate-and-fire (LIF) neurons over the 1D input space, x. Although x can be of arbitrary length, the single dimensional case is easier to visualize.

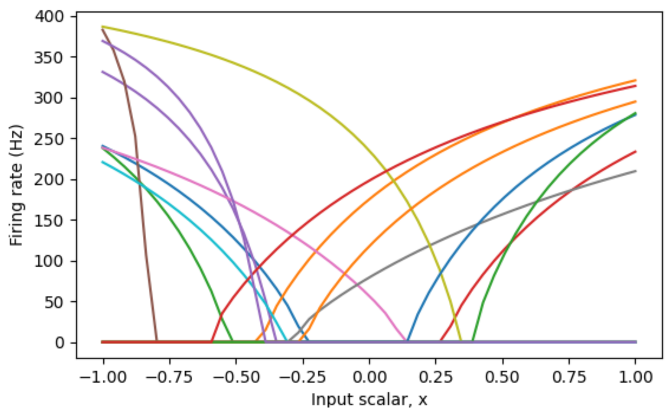


Figure 3.2: The response of various LIF neurons in the NEF to a range of input values.

It helps to visualize the encoding vector with a 3 dimensional example. Assuming a normalized input for simplicity, the input would be a vector that would point from the origin to somewhere on the surface of the unit sphere. The encoding vector can then be thought of as a preferred direction along the sphere. As the input vector approaches the direction of the encoding vector, the dot product between the two approaches 1. The dot product between the input and encoding vectors is then scaled by a gain α and shifted by a bias b, before being passed in to the neuron's function G. Adjusting the gains and biases of neurons changes the level of similarity that neurons begin to spike and the speed with which the firing rate changes. The encoders of neurons can also have positive or negative values, allowing for 'on' and 'off' neurons, whose output values rise or fall as the input signal rises, respectively. Gains and biases are usually randomly sampled to give a diverse response between neurons, although they can be specified. By selectively setting encoding vectors, the NEF allows for a whitebox tuning of neural responses to an input space. The encoding vectors can be uniformly sampled from a random distribution to cover the entire input space evenly, or more selectively tuned as needed. This allows for a way to encode incoming data into neuron spike trains, but these values need to be able to represent a numerical value that can be used for computation.

The NEF distinguishes between the neural activity in a group from the value the group represents. Unlike traditional neural networks where a single value is used in a connection between neurons, the NEF uses both input encoding weights, and output decoding weights, although these can be combined to form a standard weight matrix. The decoding weights are used to extract the represented value from the neuron group, or 'ensemble' in NEF terminology. Decoders can be solved for with a least squares minimization, given the input value to be represented and the neural activities. The represented value can then be extracted as

$$\boldsymbol{\hat{x}} = \Sigma_i^N \boldsymbol{a_i} \cdot \boldsymbol{d_i} \tag{3.3}$$

where $\boldsymbol{d_i}$ is a decoding vector for neuron i, $\boldsymbol{a_i}$ the respective activity, and $\boldsymbol{\hat{x}}$ the value the ensemble represented after summing over the N neurons in the ensemble. In practice, the differentiation between encoders and decoders is only made during the initial setup of the network, as the two weights get combined during the time of compilation as shown in figure 3.3.

At run time the encoders and decoders function similarly to a traditional ANN connection weight computationally, but with the benefits of being able to set neuron affinities to different input signals, while also decoding for a predefined function. On the left half of figure 3.3 the separated NEF encoders and decoders are shown with the ideal value of x in the center. For demonstration purposes, an artificial 'ideal' representation is shown in the center, as if the ideal value was decoded from layer A providing x, then encoded into layer B. Since the operation of encoding and decoding is linear, the weights can be combined to form the network on the right, but with the benefits of the NEF encoding and decoding process.

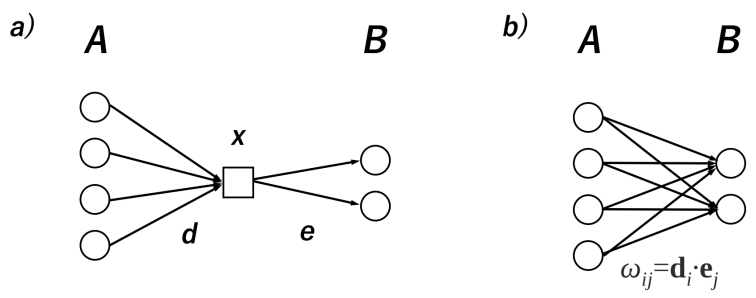


Figure 3.3: On the left, separated NEF encoders and decoders, and on the right the linearly combined weight matrix.

In the context of spiking neural networks (SNNs), working directly with spike trains can be difficult due to the discontinuous nature of the signal. Taking inspiration from biology again, the NEF smooths out the discontinuous spike trains by applying a synaptic filter over connections, as is done in the brain. The synaptic filters act as a low pass filter to smooth out the otherwise discontinuous spike raster output from neurons. This introduces a new set of dynamics into the network, with connections propogating signals over time, instead of instantly passing values between neurons as in ANNs. The previous method for solving decoders allowed for a group of neurons to simulate a function in the form of $y = f(x)$. However, this can be extended to linear dynamical systems. When used in recurrent connections, the synaptic dynamics can be used to model other dynamical systems. Treating the feedback connection like a pseudo memory of state x and incoming connections as control signals u, the NEF can directly solve for other dynamical systems that can be written in a linear state space equation as in 2.12, as well as for nonlinear dynamical systems. A and B are arbitrary matrices that describe the system dynamics and input. Although a full derivation is out of scope, it has been shown [10] that the system function can be computed by solving for connection weights the same way as in the feed-forward case, using least squares. The feedback connection weights are optimized to represent $\tau A(x) + x$, and the input connection weights are optimized to represent $\tau B(u)$, where τ is the time constant of the connection filter. These methods have been used for controlling real-time robotic systems [9], visual classification and localization models [8], natural language processing [6], and the world's largest functional brain model, Spaun[11], capable of complex tasks like reasoning, recall, motor control, and perception.

Using the methods of the NEF allows for a more direct approach to modelling dynamical systems with neural networks when compared to ANNs. This is done by treating neurons as a resource for representing numerical values with connections used to compute functions on their represented value. Through a least squares minimization, decoding weights can be solved for directly to allow for the modelling of linear or nonlinear functions in the form of $y = f(x)$. Extending this with recurrent connections then allows for the modelling of linear or nonlinear dynamical equations. Following the work of the NEF, and similarly inspired by biology, the LMU derives a novel memory unit that outperforms the current state-of-the-art, and opens the door for optimal temporal encoding to be used in neural networks.

3.3 Legendre Memory Unit

Many neural networks work by performing an inference on the instantaneous network input. This yields systems that can perform well for tasks that do not have heavily dynamical properties, such as object detection in images. As the input/output mapping that the network is tasked to learn becomes more dynamical, knowledge of how the input context varies over time becomes relevant. As the linear and angular motion of a system is inherently dynamic, predictions made with a single time step can lack the relevant information that describes the system motion. For example, knowing the instantaneous position of a system is not enough to know how it is moving. Given even the last two known positions, estimates can be made of the direction of motion and the size of the step. The longer the window of context history is, the more information is available to describe the past motion of a system. Neural networks often use recurrent connections to capture history, feeding outputs back into a group of neurons, to form a memory of inputs. Through backpropagation the network learns to form a representation of the time history of data over some window. These recurrent networks use many tricks to optimize how much data can be remembered, but are limited due to the vanishing gradient problem inherent with backpropagation over time.

The LMU is a novel recurrent network composed of a linear encoding layer, and a nonlinear layer for function learning, and greatly outperforms other commonly used recurrent and feedforward units [29]. Positional encoding, as in LSTMs and GRUs, is prone to saturation effects and unstable gradients when tasked with sequence lengths greater than 2000-5000 steps[29]. The linear portion of the LMU, the LDN, is mathematically proven to be an optimal compression for sequences, and can efficiently handle temporal dependencies spanning a million time steps [29]. LMUs have also been shown to have a 10x improvement in data efficiency on language modelling when compared to the previous state of the art transformer networks [6]. This means that, depending on the parameters set, a network with an LMU can be trained on 10x less data while achieving the same when compared to a transformer network.

The LDN approaches the problem of memory as one of delaying a signal by some length of time. This can be modelled with the linear transfer function for a continuous time delay, $F(s) = e^{-\theta s}$, where θ is the length of the delay. With the goal of implementing the algorithm with the NEF, the LDN approximates this by d coupled ordinary differential equations (ODEs).

$$\theta \dot{\boldsymbol{m}}(t) = \boldsymbol{A}\boldsymbol{m}(t) + \boldsymbol{B}u(t) \tag{3.4}$$

where $\boldsymbol{m}(t) \in \mathbb{R}^{q}$ is the q dimensional state vector, u is a 1D state, and $\boldsymbol{A}$ and $\boldsymbol{B}$ are the ideal state-space matrices. The rational polynomial representation of the ideal time delay leads to a system with infinite order. To overcome this [30] used Padé approximants to represent the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices as

$$\boldsymbol{A} = [a]_{ij} \in \mathbb{R}^{qxq}, a_{ij} = (2i+1)\begin{cases} -1, & i < j \\ (-1)^{i-j+1}, & i \geq j \end{cases} \tag{3.5}$$

$$\boldsymbol{B} = [b]_i \in \mathbb{R}^{qx1}, b_i = (2i+1)(-1)^i, \text{ where } i, j \in [0, q-1] \tag{3.6}$$

Padé approximants are similar to a Taylor series expansion, but extended to rational polynomials. The key property of the LDN is that $\boldsymbol{m}$ is a polynomial representation of the time history of a 1D state u. By representing the coefficients of a set of polynomials, an approximation can be made of a temporal signal. Specifically, the LDN uses shifted Legendre polynomials as its function basis, as shown in figure 3.4.

By projecting the input signal into the Legendre domain, the LDN orthogonalizes a sliding window of context history $u(t) \in \mathbb{R}$ of length θ onto q legendre polynomials. The LDN continuously updates the weighting of the q coefficients to maintain a sliding window representation of u. Using a polynomial representation also allows for a continuous representation of input data. As such, any time within the window, $0 \leq \theta' \leq \theta$, can be decoded with

$$u(t-\theta') \approx \sum_{i=1}^{q-1} P_i\Big(\frac{\theta'}{\theta}\Big) m_i(t), 0 \leq \theta' \leq \theta \tag{3.7}$$

$$P_i(r) = (-1)^i \sum_{j=0}^{i} \binom{i}{j} \binom{i+j}{j} (-r)^j \tag{3.8}$$

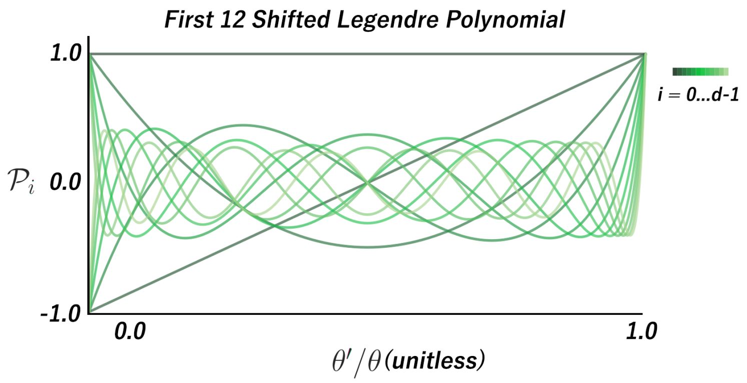


Figure 3.4: The first 12 shifted Legendre Polynomials with coefficients set to 1. The LDN uses linear combinations of the scale invariant Legendre Polynomials to represent the θ length window. Increasing the number of polynomials allows for the storage of higher frequency data.

where i is the index from 1 to q of the i^{th} shifted Legendre polynomial, P_i. r is the ratio of θ'/θ, where θ' is the number of seconds in the past relative to t to decode the value of u. Legendre polynomials can also be used to represent data regardless of time scale. As the frequency of input data increases relative to the time scale, increasing the number of Legendre Polynomials q improves the LDN's ability to represent the higher frequency data. Decoding is also a linear operation and since P_i is only defined by the constant r it can be calculated outside a control loop and held constant.

The LMU is then defined by feeding this optimally compressed memory signal u into the transfer function of a layer of neurons. The non-linearity can be used to learn functions of the time history of a signal. This introduces a highly efficient way to train neural networks with memory that has an underlying representation of time. Dynamical systems are inherently built upon the transition between states over time. In the same spirit, both the LMU and the NEF were developed for modelling neural networks as dynamical systems. The LDN allows for an optimal way to represent the *past* history of a context signal. Building upon the framework of the LDN, the Learned Legendre Predictor (LLP) uses online learning to learn a compressed representation of the *future* time window of a (sub)set of the input signal.

3.4 The Learned Legendre Predictor

The LLP, [12], is a system that uses LDN representations and a modified delta learning rule to make predictions with a single ensemble of neurons. By leveraging the ability of the LDN to represent time histories of data, the LLP uses the compressed time representations of neural activities and past network outputs to make predictions of the future trajectory of states $\boldsymbol{z}$. Notably, the prediction is made in the legendre domain, such that a continuous representation is formed. A diagram of the entire LLP system shows the flow of information in figure 3.5.

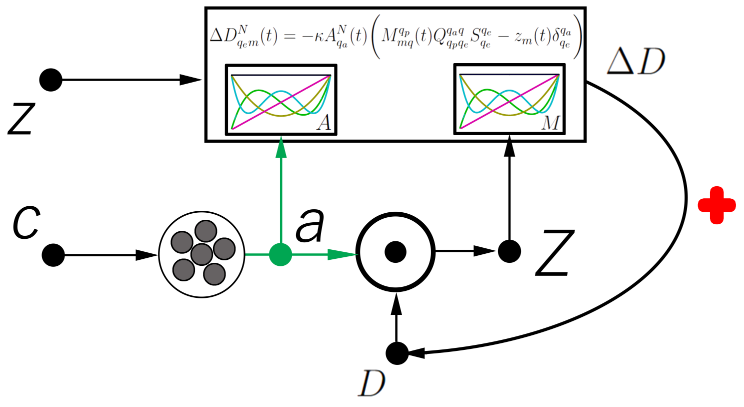

$\Delta D^N_{q_e m}(t) = -\kappa A^N_{q_a}(t)\left(M^{q_p}_{mq}(t) Q^{q_a q}_{q_p q_e} S^{q_e}_{q_e} - z_m(t)\delta^{q_a}_{q_e} \right)$
A
M
ΔD
z
c
a
Z
D

Figure 3.5: The network diagram of the LLP system. See text for more details.

The boxes with Legendre polynomials signify an LDN encoding. The context, $\boldsymbol{c}$, is passed into an ensemble of neurons. The neural activities, $\boldsymbol{a}$, are dotted with decoding weights, $\boldsymbol{D}$, to make a prediction, $\boldsymbol{Z}$. The prediction is an LDN representation of the future θ length time window of states $\boldsymbol{z}$. $\boldsymbol{z}$ can be the same as $\boldsymbol{c}$, a subset of it, or an entirely different value. $\boldsymbol{c}$ is the context signal that the modeler selects as relevant in predicting the evolution of states $\boldsymbol{z}$. By providing the network output, activities, and current value of $\mathbf{z}$, the learning rule outputs a change in decoding weights, $\Delta\mathbf{D}$, to improve the network's prediction. The box in 3.5 contains the learning rule, which is a modified delta rule. The delta rule is a specific case of gradient descent used for a single layer of neurons. It can be formulated as

$$\Delta\boldsymbol{D}(t) = -\kappa\boldsymbol{a}(t) \times (\hat{\boldsymbol{z}}(t) - \boldsymbol{z}(t)) \tag{3.9}$$

where $\Delta \boldsymbol{D}$ is the weight matrix update, κ is the learning rate, $\boldsymbol{a}(t)$ are the activities of neurons in the layer at time t, and $\hat{\boldsymbol{z}}(t)$ and $\boldsymbol{z}(t)$ are the layer predictions and ground truth at time t. By adjusting the learning rate for stability, similar to gain tuning in PID controller, the weights of a layer of neurons can be slowly adjusted to better approximate a function. The main differentiator of the LLP is that the network learns to output a polynomial representation of the *future* θ seconds length window of $\boldsymbol{z}$ through a similar encoding as in the Legendre Delay Network. Similarly to the Prescribed Error Sensitivity (PES) [31] learning rule derived with the methods of the NEF and built upon the delta rule, the LLP can be run offline or online, despite the ground truth not being available at the time of prediction.

This type of prediction of future states can be done without the LLP by training neural networks to directly predict the future state of a system, given some context. However, there are a few limitations with current approaches. A neural network trained with back-propagation could learn to make this type of prediction, but only at discrete times in the future. By predicting the output in the Legendre domain, the prediction can be decoded at any continuous time between t and $t + \theta$. Another issue arises in that the delta rule uses a known ground truth to update the decoders. Since the prediction is being made of the unknown, future state, a history of activities and predictions needs to be held for θ seconds to be able to make a comparison to the ground truth when it becomes available. With a conventional approach, this would require a secondary network [12]. One network would learn to output a prediction of the future states given the current context, while the secondary network would be given temporal histories and learn to make better predictions. Through weight sharing, the two networks could learn to make a prediction of the future state, but still in a discretized manner. The issue also remains of how to represent the temporal context that needs to be stored for θ seconds.

Through the use of the LDN to compress the time histories of neural activities and the network predictions, the LLP derives a learning rule that can make use of the LDN optimality to make continuous time and space predictions of future states online. Starting with the delta learning rule in equation 3.9, making a prediction of the future at a single time (not continuous) requires the error to be aligned in time with the activities of the neurons that made the prediction which led to the error. This can be accomplished by shifting the network outputs by the delay τ. Equation 3.11 shows equation 3.10 in Einstein notation [1], and represents the delta rule for a time shifted weight update.

[1]The remainder of the section is written in Einstein notation where the subscripts and superscripts define the dimensionality of the tensor, and as such bold font will be omitted as the tensor dimensionality will be noted.

$$\Delta \boldsymbol{D}(t) = -\kappa \boldsymbol{a}(t-\tau)(\hat{\boldsymbol{z}}(t-\tau) - \boldsymbol{z}(t)) \tag{3.10}$$

$$\Delta D(t)^N_m(t) = -\kappa a^N(t-\tau)(\hat{z}_m(t-\tau) - z_m(t)) \tag{3.11}$$

N defines the number of neurons in the layer, and m is the dimensionality of the network output. Since the goal of the network is to learn the LDN representation of the future window of z_m, the value of $\hat{z}_m$ is not directly output. Instead, the network outputs the prediction in the legendre domain as Z^q_m, where q is the number of legendre polynomials used to represent the temporal prediction. To make the comparison to previous network states and outputs as the delayed ground truth becomes available, LDNs are used to store the time histories of network predictions, Z^q_m, and neuron activities a^N. The time shifted values of predictions and activites can be decoded from their LDN representations $M^{q_p}_{mq}$ and $A^N_{q_a}$, respectively. The same linear operation as in equation 3.8 is used for decoding. Since Z^q_m is itself an LDN representation, $M^{q_p}_{mq}$ is an LDN representation of an LDN representation. $M^{q_p}_{mq}$ therefore needs to be decoded twice to extract the predicted value $\hat{z}_m(t-\tau)$. With this substitution equation 3.11 becomes

$$\Delta D(t)^N_{q_e m} = -\kappa A(t)^N_{q_a} P(\tau)^{q_a} \Big(P(\tau)_{q_p} M(t)^{q_p}_{mq} P(\tau)^q - z_m(t) \Big) P(\tau)_{q_e} S^{q_e}_{q_e} \tag{3.12}$$

where the network weights, $\Delta D(t)^N_{q_e m}$, are in the Legendre domain. Equation 3.12 can be broken up into three terms. The first term, $A(t)^N_{q_a} P(\tau)^{q_a}$, decodes the delayed neural activity that resulted in the prediction. $P(\tau)_{q_p} M(t)^{q_p}_{mq} P(\tau)^q$ is the value of the network prediction, $\hat{z}_m$, of what z_m will be τ seconds into the future, made τ seconds ago. This delay allows for a comparison to the delayed ground truth $z_m(t)$. Since the network weights are in the Legendre domain, the difference between $\hat{z}_m$ and z_m needs to be shifted back into the Legendre domain as well. This is done with $P(\tau)_{q_e}$ since the error being learned is in the Legendre domain. This introduces the index q_e, which will always equal q, and represents the LDN dimensionality of the resulting error term. However, LDN representations of the activity history and prediction history can have a different Legendre dimensionality, q_a and q_p, than used for the network prediction q. Lastly, $S^{q_e}_{q_e}$ is a scaling factor to normalize the Legendre polynomials, as they are othogonal, but not orthonormal. $S^{q_e}_{q_e}$ is a constant diagonal matrix with $S[i,i] = 2i+1$, and can therefore be precomputed.

Using equation 3.12 allows for predictions to be made for a particular value of τ. To get the value across the continuous window θ for every value of τ, the integral needs to be taken.

$$\Delta D(t)^N_{q_e m} = -\kappa A^N_{q_a}(t)\left(M(t)^{q_p}_{mq}\int_0^1 \mathcal{P}(\tau)^{q_a}\mathcal{P}(\tau)_{q_p}\mathcal{P}(\tau)^{q}\mathcal{P}(\tau)_{q_e}S^{q_e}_{q_e}d\tau - z_m(t)\int_0^1 \mathcal{P}(\tau)^{q_a}\mathcal{P}(\tau)_{q_e}S^{q_e}_{q_e}d\tau\right) \tag{3.13}$$

Since $A(t)^N_{q_a}$ and $M^{q_p}_{mq}$ are independent of τ they can be pulled out of the integral. τ is bound between $[0,1]$ since $\theta' <= \theta$. With some rearranging and simplification [12], the integrals can be simplified as

$$\Delta D(t)^N_{q_e m}(t) = -\kappa A^N_{q_a}(t)\left(M^{q_p}_{mq}(t)Q^{q_a q}_{q_p q_e}S^{q_e}_{q_e} - z_m(t)\delta^{q_a}_{q_e}\right) \tag{3.14}$$

The second integral of 3.13 simply becomes an identity matrix, $\delta^{q_a}_{q_e}$. The first integral becomes the 4D tensor $Q^{q_a q}_{q_p q_e}$. However, $Q^{q_a q}_{q_p q_e}$ is independant of any observations and can be precomputed. This results in the final LLP learning rule. Using the LLP system then allows for a neural network to be developed to make predictions of the future evolution of system states over a continuous window. Adding in the ability to represent inputs with LDNs extends the ability of the LLP by providing temporal context to learn system dynamics. The python code for calculating $S^{q_e}_{q_e}$, $\delta^{q_a}_{q_e}$, and $Q^{q_a q}_{q_p q_e}$ is shown in A.2.

Chapter 4

Experimental Methods

This chapter includes the methods used to develop the training system that was used to train and test the novel contributions covered in chapter 5. The training system encompassed the experimental setup and simulation environment. Simulations, as opposed to real-world experiments, allow for a faster turn around when running many experiments, and do so at a much lower cost. The training data for the LLPSE and LLPC was generated with a PD controlled quadrotor. Details on the state transformations and gains determined for the PD controller are covered in this chapter, in addition to the generation of the reference trajectory. A PD controller was chosen as no external perturbing forces were added to the simulation, so the added complexity of tuning an integral term was omitted. With the goal of capturing the underlying dynamics of the system, and not recreating the controller used to generate the training data, there was also a larger tolerance for error in the baseline controller used.

4.1 Experimental Setup

All the experiments in this thesis were run using a simulated quadrotor in Unreal Engine 4 (UE4). The flight physics were simulated using the AirSim [25] plugin. In addition to providing many tools for data collection, at its core the AirSim plugin provides a quadrotor model with a fast physics simulation of flight dynamics. Leveraging the high customizability of UE4, this allowed for near real-time simulation in photo-realistic environments. With the underlying Unreal Engine framework, the combination provides for a rich, and extendable simulation environment, as shown in figure 4.1. The specifications of the computer used for simulations are: Ubuntu 20.04.2 LTS Operating System, AMD Ryzen 9 3950x CPU, Nvidia GeForce RTX 2080 Ti GPU, and 64GB of memory.

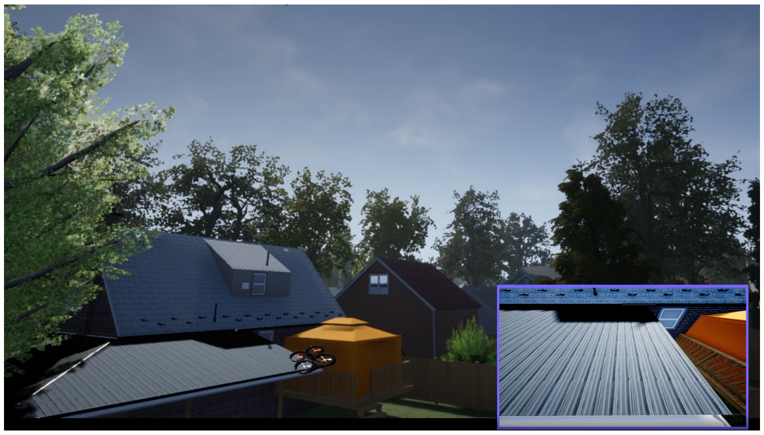
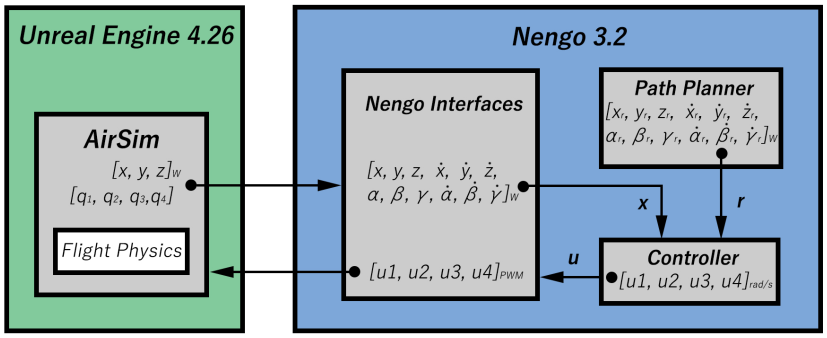


Figure 4.1: An example scenario with Unreal Engine and the AirSim quadrotor. AirSim also provides many methods for collecting data, including cameras such as RGB, Depth, Segmentation, Spiking, and more.

The AirSim plugin has a wide array of sensory feedback available for quadrotor simulations. Many visual sensors are available, as noted in figure 4.1. Although the LLPSE can include any of the sensors available in AirSim as part of its context, discussion in this thesis will be limited to using the drone position, orientation, and linear and angular velocities. The lowest level of control available through the AirSim API is through a pulse-width-modulated control signal, between 0 and 1, for the four rotors. The neural networks designed in this thesis were written in Python 3.8, using the Nengo software package.

Nengo is a neural network simulator and hardware compiler built with the NEF principles at its core. The core of Nengo is built from 5 objects: Networks, Ensembles, Nodes, Connections, and Simulators. Neural networks are contained in a Network object. The networks themselves are built up from Ensembles of neurons, Nodes, and Connections. A Node is a generic block in Nengo to contain arbitrary Python code to be interfaced with ensembles of neurons. The connection object passes information between ensembles and nodes, in addition to performing the synaptic filtering and applying network weights. Once compiled, the Network object is then simulated in a Simulator object. Additionally, Nengo has Probe objects that can be used to record values in the network during simulation. In addition to Nengo, multiple packages are available to extend the abilities of the core objects. Nengo GUI allows for visualization of networks with live plotting of Probe values. Nengo DL allows for the development of DNNs using traditional backpropogation.

Various backends are available for compiling networks to run on CPUs, GPUs, FPGAs, and Neuromorphic hardware. The Nengo Interfaces package allows for Nengo networks to be interfaced with various simulators such as Mujoco and CoppeliaSim. In this thesis it was used to interface between the Nengo networks and the AirSim simulator, running in Unreal Engine 4. The communication between interfaces is outlined in figure 4.2.



Unreal Engine 4.26
AirSim
$[x, y, z]_W$
$[q_1, q_2, q_3, q_4]$
Flight Physics
Nengo 3.2
Nengo Interfaces
$[x, y, z, \dot{x}, \dot{y}, \dot{z}, \alpha, \beta, \gamma, \dot{\alpha}, \dot{\beta}, \dot{\gamma}]_W$
$[u1, u2, u3, u4]_{PWM}$
Path Planner
$[x_r, y_r, z_r, \dot{x}_r, \dot{y}_r, \dot{z}_r, \alpha_r, \beta_r, \gamma_r, \dot{\alpha}_r, \dot{\beta}_r, \dot{\gamma}_r]_W$
x
r
u
Controller
$[u1, u2, u3, u4]_{rad/s}$

Figure 4.2: The communication between the various simulators and control modules.

The drone feedback from Airsim included translational states (position and velocity) in the world frame, in units of meters and meters per second, respectively. The rotational states were provided as a quaternion. The Nengo Interface's AirSim wrapper converts the quaternion feedback into Euler angles. The Euler angles were provided in the $z-x-y$ Tait-Bryan convention, commonly used for flight, as it aligns with the body frame as discussed in Section 2.1. The target angles from the path planner were calculated in the $x-y-z$ order and required an additional transform in the controller to align with the Tait-Bryan angles returned from Nengo Interfaces. The Python code for converting the Euler angle order can be found in Appendix A.1. Nengo Interfaces also converts control signals as rotor speeds in $\left(\frac{radians}{second}\right)^2$ to the equivalent pwm signal expected by AirSim, given the drone constants. The path planner was encapsulated in a Node and provided the reference, $\boldsymbol{r}$, to the Nengo networks and the PD controller. With the simulation environment in place, the first step was to generate the training data for the LLPSE to use to learn the system dynamics.

4.2 Dataset Generation

System identification typically falls into one of two categories, model-based or model-free [28]. A PD controller was selected for generating the training data to simplify the problem of identifying the system dynamics. Although a model-free approach could theoretically work, by starting with an underlying controller, the modeler has more control over the state space covered during data collection. As there were no external perturbing forces added to the simulation, a PD controller was chosen over PID to simplify the tuning of gains. The modified form of the PID equation (2.1) used to control the quadrotor included the gravity term, and removed the integral gains and errors, and is given as:

$$\mathbf{u}(t) = \mathbf{T} \cdot \mathbf{K} \cdot \mathbf{e}(t) + \mathbf{g} \tag{4.1}$$

where, $\mathbf{g}$ is the gravity vector containing the four rotor velocities required to hover in units of $\left(\frac{radians}{second}\right)^2$, and acted as a baseline for the control signal. The value was determined empirically to be approximately $[6800, 6800, 6800, 6800]^T \left(\frac{radians}{second}\right)^2$, or $[82.5, 82.5, 82.5, 82.5]^T$ in $\left(\frac{radians}{second}\right)$. The point was selected where the drone would just begin to take off. The remainder of equation 4.1 adds the offset about the hover point, given the current body frame error, $\boldsymbol{e}(t)$, to the reference trajectory, $\boldsymbol{r}(t)$, where

$$\boldsymbol{e}(t) = \begin{bmatrix} e_x & e_y & e_z & e_{\dot{x}} & e_{\dot{y}} & e_{\dot{z}} & e_\alpha & e_\beta & e_\gamma & e_{\dot{\alpha}} & e_{\dot{\beta}} & e_{\dot{\gamma}} \end{bmatrix} \tag{4.2}$$

$\mathbf{e}(t)$ is the 12 dimensional vector of the position, linear velocity, orientation, and angular velocity errors. The error was calculated by taking the difference between $\boldsymbol{x}$ and $\boldsymbol{r}$. The errors were then transformed into the body frame by rotating positions and linear velocities to align with the body frame of the drone. This was done by rotating the x and y components of position and velocity by the system yaw, γ. This positioned errors along x in the drone forward direction, as discussed in Section 2.1. The yaw error was rolled over to use the shortest angle between the state and target, yielding the final error term used. The error was dotted with $\mathbf{K}$, the gain matrix formed by stacking the individual gain vectors of the four controllable states: thrust, pitch, roll, and yaw, where pitch and roll were coupled with x and y motion, respectively, giving

$$\boldsymbol{K} = \begin{bmatrix} 0 & 0 & k_z & 0 & 0 & k_{\dot{z}} & 0 & 0 & 0 & 0 & 0 & 0 \\ k_{xy} & 0 & 0 & k_{\dot{x}\dot{y}} & 0 & 0 & 0 & -k_{\alpha\beta} & 0 & 0 & -k_{\dot{\alpha}\dot{\beta}} & 0 \\ 0 & k_{xy} & 0 & 0 & k_{\dot{x}\dot{y}} & 0 & k_{\alpha\beta} & 0 & 0 & k_{\dot{\alpha}\dot{\beta}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_\gamma & 0 & 0 & k_{\dot{\gamma}} \end{bmatrix} \tag{4.3}$$

The rows of $\boldsymbol{K}$ represent the gain for scaling the importance of error along z, *pitch*, *roll*, and *yaw* axes. The columns signify the error with respect to the 12 measurable drone states, as ordered in equation 4.2. This way the gain matrix also acts as a transform from the measureable states to the controllable states. Note that the x and y, and α and β gains are coupled to maintain a balanced control as there should not be a significant difference in motion along the xy plane, nor in the roll and pitch axes. The gains for linear and angular velocity were similarly coupled. Errors along z and $\dot{z}$ were used to adjust the thrust. Due to the coupling as a result of the under-actuated system, as discussed in section 2.1, pitch was adjusted based on errors along x, $\dot{x}$, β, and $\dot{\beta}$. Similarly, roll was adjusted based on errors along y, $\dot{y}$, α, and $\dot{\alpha}$. Lastly, yaw was adjusted based on errors along γ and $\dot{\gamma}$. With the errors scaled by their respective gains and transformed into z, *pitch*, *roll*, and *yaw* commands, the final step was to transform from the controllable states to the corresponding rotors to elicit the desired motion using

$$\boldsymbol{T} = \begin{bmatrix} -1 & -1 & -1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & -1 & -1 \end{bmatrix} \tag{4.4}$$

The matrix $\boldsymbol{T}$ transforms from the controllable states to rotor space. Here, rotor space is defined as rotor commands in $\left(\frac{radians}{second}\right)^2$, in the order front right, rear left, front left, and rear right (i.e., $\mathbf{u}(t) = [u_{FR}, u_{RL}, u_{FL}, u_{RR}]^T$). The rows of $\boldsymbol{T}$ represent the rotor space, with the first row controlling the front right rotor, the second row the rear left rotor, etc. The columns represent motion along the z, *pitch*, *roll*, and *yaw* axis. With $\boldsymbol{g}$ providing the baseline of the control signal, $\boldsymbol{T}$ then takes the errors along the controllable axes, and transforms them to the corresponding rotor direction offsets about the hover speed. $\boldsymbol{T}$ follows the same pairings as shown in figure 2.1. As an example, a positive error along the roll axis (column 2) signifies that the drone needs to increase rotor velocities for the rear left and front left rotors (rows 2 and 4), and decrease the velocities of the rear right and front right rotors (rows 1 and 3). This would increase the roll in the positive direction, resulting in a larger thrust component along y, lowering the error to reference y_r. In this manner, the transform would set the direction of the rotor offsets, and the gain and error terms would determine the magnitude of the offset with respect to the hover point.

The reference path, $\boldsymbol{r}$, is generated using the path planner avalailable in the Github open source repository, ABR Control. An example 12D path is show in figure 4.3. $\boldsymbol{r}$ is planned in the velocity space to create a path that maintains some physical constraints of motion. This includes avoiding large discrete steps in velocity, and unattainable speeds or accelerations. Given the timestep of the simulation to be run, a path is planned where a position, orientation, and their derivatives are updated each step, and they lay out the target trajectory for the drone to follow over time. For simplicity, the trajectories between targets are kept linear in position, with start and target velocities set to $0m/s$. However,

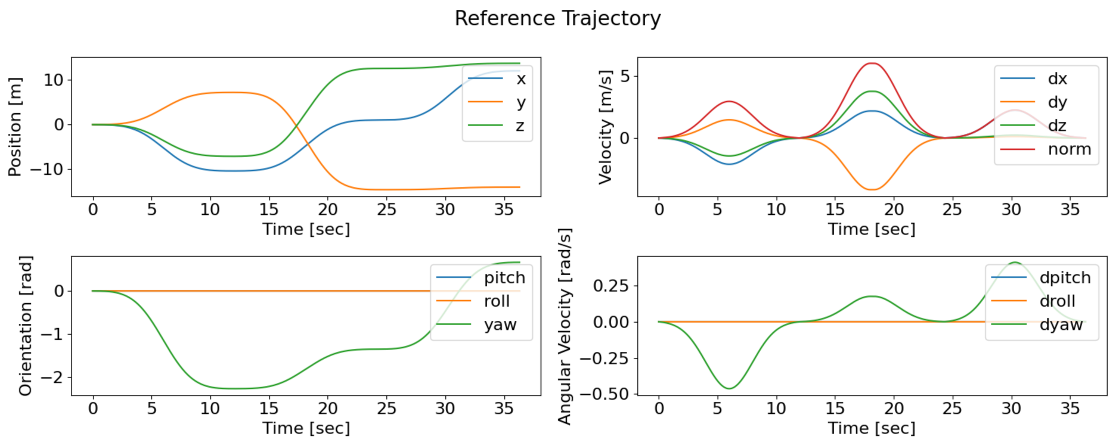


Figure 4.3: An example generated reference trajectory, with the same constraints used for data collection. The target positions are chosen from a uniform distribution between $[-15, 15]m$ for x and y, and $[-1. -30]m$ for z. The start and target velocities at each target position are $0m/s$, with a maximum acceleration of $1m/s^2$, and a maximum velocity of $6m/s$. The orientation path is planned using quaternion spherical linear interpolation (SLERP), and is set to have a matching velocity profile to the planned position path to maintain a synchronized convergence to translational and rotational targets.

the velocity profile follows a Gaussian curve up to a maximum velocity of $6m/s$, provided there is enough time given an acceleration of $1m/s^2$. The path accelerates to the maximum possible velocity, within the limit, before needing to decelerate to converge to the target position while moving at the target velocity. If the maximum velocity is reached before needing to slow down, the velocity is held constant at the maximum. The orientation is calculated using quaternion spherical linear interpolation (SLERP), and converted to euler angles. The quaternion SLERP algorithm takes in a normalized value, and returns the angle between the start and target that is that proportion along the way. A value of 0 is the start angle, 1 the target, and 0.5 the angle between the two. However, the step size used for quaternion SLERP is set to match the normalized step size in the position path. This allows for a synchronized convergence to targets between position and orientation paths. Target roll and pitch angles are held constant at 0 radians.

With a reference available, the controller gains can be tuned. The gain vector is

$$\boldsymbol{k} = [k_{xy}, k_z, k_{\alpha\beta}, k_\gamma, k_{\dot{x}\dot{y}}, k_{\dot{z}}, k_{\dot{\alpha}\dot{\beta}}, k_{\dot{\gamma}}] \tag{4.5}$$

The controller gains are tuned using Neural Network Intelligence (NNI) with the Tree-structured Parzen Estimator (TPE) to trace the trajectory outlined in $\boldsymbol{r}$. NNI is an open source resource for running optimizations over a search space given an error metric from a loss function. The loss function is a running sum of error $\boldsymbol{e}$, with some additional heuristics. Early stop metrics are used to stop simulations early that exhibit clear failure markers, such as extremely high angular velocities, or large distances in positional error. For simulations that are stopped early, the error at the cutoff is used for the remaining steps that are omitted. This allows for poorly performing gains to still be ranked in the optimization. Additional scaling is applied to orientation errors to improve convergence in target yaw values. The specific gain values found that are used in the experiments in this thesis are: $[9483, 4157, 2974, 11013, 14788, 9378, 334, 6736]$. Using the gains from the NNI optimization, shown in equation 4.5, an average 2norm error of 2cm over time is obtained. To use the training data for training neural networks, some preprocessing is required to remove outliers and normalize the data.

4.3 Preprocessing

The context signal given as network input data is a stacked combination of $\boldsymbol{x}$, $\boldsymbol{r}$, and $\boldsymbol{u}$. The training data generated in Section 4.2 is preprocessed before any training. A mean shifting and scaling is applied to ensure normalized input vectors. This is common practice in training neural networks as it helps during the training process. The scale between the various dimensions of the three groups varies by several orders of magnitude. To speed up the training process, and to start with a similar weighting between input dimensions, the data is commonly normalized. Due to the unconstrained nature of PD control, the output of the controller occasionally leads to physically impossible control actions. These result in large spikes in the output of the controller. The spikes in control actions can be observed in Appendix B.1. To increase the resolution of the lower amplitude control signal with physically feasible control actions, the control data is first clipped to a range of [6400-7400] $\left(\frac{radians}{second}\right)^2$, or [57.27-107.65] in $\frac{radians}{second}$. The default AirSim drone has a maximum rotor velocity 107.65 radians/sec and sets the upper limit. With the gravity compensation selected as the mean, the difference to the maximum speed is subtracted to get the lower limit. The signal is shifted by the mean (gravity compensation) and scaled by the clipped max. A portion of the preprocessed control signal data is show in figure 4.4.

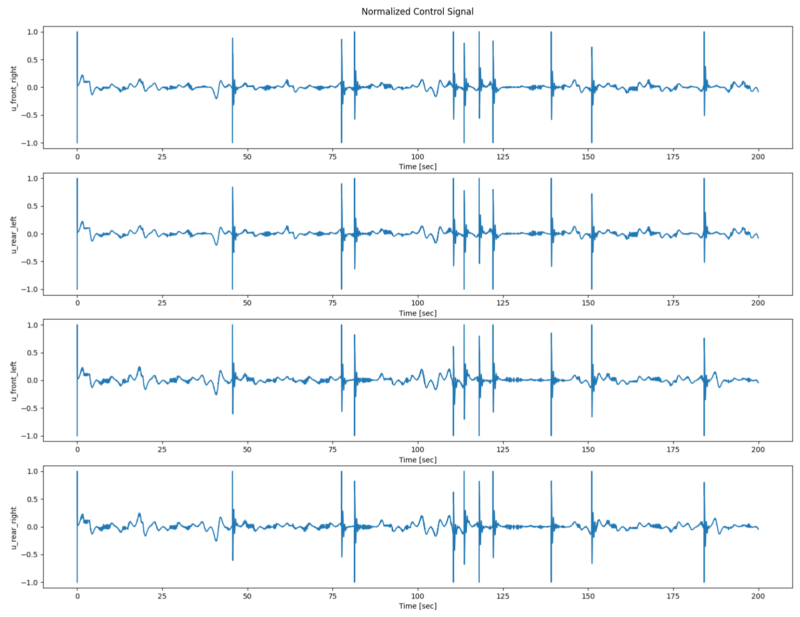


Figure 4.4: The normalized control signals used for training, validation, and testing. The spikes in the control action occur when a new target is set in the path planner. As the PD controller is reactive, it takes some time to overcome the system momentum, resulting in a delayed control response. This delayed response leads to an accumulation of error. The error feeds back into the control law, leading to a feedback loop that causes the spikes in the control action observed. As the control law begins to overcome the system momentum, the error begins to drop and the control action drops back into a feasible range for the drone.

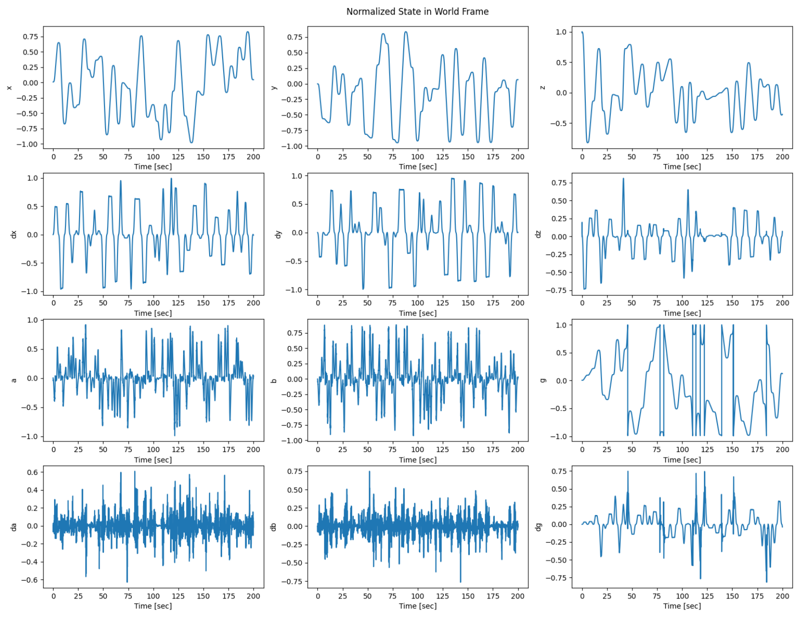


Figure 4.5: The normalized world frame states used for training, validation, and testing. The rows from top to bottom are: position, linear velocity, orientation, and angular velocity. The columns show the 3 dimensions of the sub-state in the row (i.e., x, y, and z, for row 1). The states shown are the resulting feedback from using a PD controller (equation 4.1) to control a simulated quadrotor in AirSim to follow the target path, $\boldsymbol{r}$.

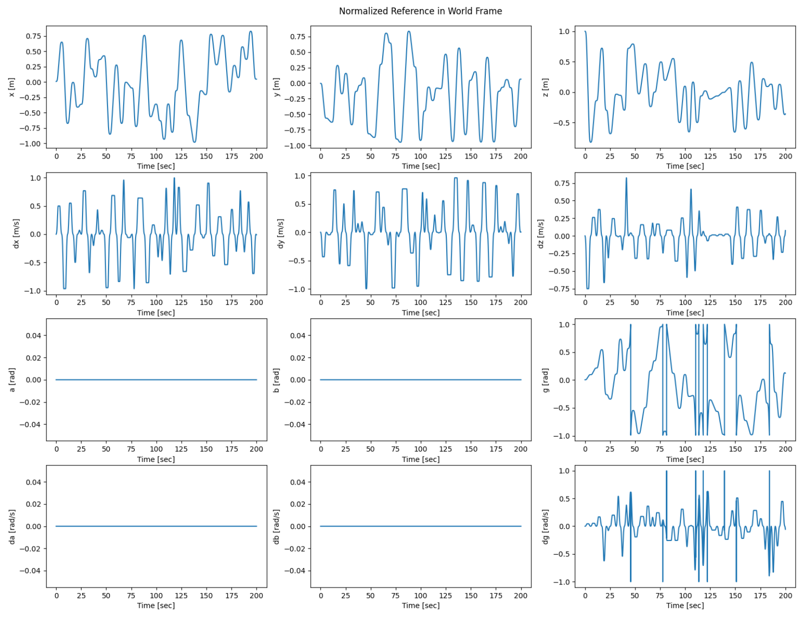


Figure 4.6: The normalized world frame reference used for training, validation, and testing. The target positions are chosen from a uniform distribution between $[-15, 15]m$ for x and y, and $[-1. - 30]m$ for z. The start and target velocities at each target position are $0m/s$, with a maximum acceleration of $1m/s^2$, and a maximum velocity of $6m/s$.

The state data, $\boldsymbol{x}$, and reference, $\boldsymbol{r}$, are similarly normalized. Each individual state dimension of the 12D signals are mean shifted and normalized by the absolute maximum value. $\boldsymbol{r}$ has large discrete jumps in angular velocity as the yaw rolls over from $-\pi$ to π. This is caused by the way the angular velocity is calculated through differentiating the yaw path in the path planner. This is corrected by clipping the angular velocity of $\boldsymbol{r}$ at $2\ \frac{radians}{second}$. This is the maximum observed rotational velocity of the drone in $\boldsymbol{x}$. A portion of the state and reference used as context for the LLPSE is shown in figures 4.5 and 4.6. The quadrotor dataset is used for offline training of the LLPSE.

To test the LLP online learning, a secondary and smaller dataset is generated. The smaller dataset contains the control signal and 2D state of an inverted pendulum dynamical system. The state contains the pendulum angle and angular velocity. The control signal contains the torque applied by a motor at the pivot of the pendulum. A similar preprocessing method is used, scaling the angle, velocity, and control signal by 1.57, 60, 100, respectively, to normalize the data. The inverted pendulum is controlled by a P controller (I and D gains of zero), with the goal of maintaining the upright position. Every 5 seconds a 5Nm force is applied to perturb the system states, with the simulation running for 30 seconds. The dynamical model, controller, and simulation were obtained from [2] and modified for saving data and adding the perturbing force. With the training data prepared, it is now possible to train the LLP.

Chapter 5

Design of the LLP System

This chapter covers the novel contributions of the thesis, including the implementation of the LLP learning algorithm and the LLPSE. All algorithms are implemented in Python using the Nengo software package. The LLP learning algorithm is developed to test the ability of the LLP to learn system dynamics online. The purpose of the state estimator is to encapsulate the LDN encoding of input context, and decode the network predictions out of the Legendre domain. The state estimator is also developed to allow for offline and online learning. The goal of the state estimator is to learn the drone dynamics by learning the input-output mapping from system context, $\boldsymbol{c}$, to the future trajectory of a subset of the context, $\boldsymbol{z}$. The context selected is the same information made available for MPC, including the system feedback, $\boldsymbol{x}$, the reference trajectory, $\boldsymbol{r}$, and the incoming control signal, $\boldsymbol{u}$. $\boldsymbol{z}$ is selected to be the world frame position, $[x, y, z]$. The goal of the LLPSE is to use multiple instantiations of the estimator with varying control context to make predictions across the control space, with the ultimate goal of using the predictions to select the control action for the simulated quadrotor.

5.1 Design of the LLP and LLPSE

The LLP system description in this section is written in Einstein notation to simplify the notation of tensor dimensionality. The subscript and superscript q will be used to define the LDN dimensionality, with an additional subscript used to differentiate between the input data being encoded. With system context available from $\boldsymbol{x}_{12}$, $\boldsymbol{r}_{12}$, and $\boldsymbol{u}_4$, the subscript and superscript n is used to identify the number of dimensions selected from the available context dimensions (12 for $\boldsymbol{x}$ and $\boldsymbol{r}$, and 4 for $\boldsymbol{u}$). An additional subscript x, r, and u is used to differentiate between the state, reference, and control.

The LLP system is written as a Python class and designed using the Nengo package. The Nengo network can be visualized in the Nengo GUI, as shown in figure 5.1. A single ensemble of N neurons is used with the learning rule encapsulated in a Node. A Node object is opted for over a Connection object as it simplifies the integration into the Nengo network. The raw ensemble activities, $\boldsymbol{a}_N$, are passed into a node, labelled 'ldn_activities' in figure 5.1, to perform the LDN encoding into $\boldsymbol{A}_N^{q_a}$. Similarly to the activity, the LDN encoding of $\boldsymbol{Z}_m^q$, $\boldsymbol{M}_{mq}^{q_p}$, is encapsulated in a Nengo Node, labelled 'ldn_Z'. $\boldsymbol{A}_N^{q_a}$, $\boldsymbol{M}_{mq}^{q_p}$, $\boldsymbol{a}_N$, and $\boldsymbol{z}_{n_z}$ are fed into the learning node. Given a non-zero learning rate upon instantiation of the LLP class, the learning node calculates $\boldsymbol{\Delta D}_m^{Nq_e}$ using equation 3.14. The numpy *einsum* function simplifies keeping track of which dimensions are being summed across in the various dot products of the learning rule. The learning node also performs the dot product between $\boldsymbol{a}_N$ and $\boldsymbol{D}_m^{Nq_e} + \boldsymbol{\Delta D}_m^{Nq_e}$, yielding the LDN representation of the predicted future window of $\boldsymbol{z}_{n_z}$, $\boldsymbol{Z}_m^q$. The values of $\boldsymbol{\delta}_{q_e}^{q_a}$, $\boldsymbol{S}_{q_e}^{q_e}$, and $\boldsymbol{Q}_{q_p q_e}^{q_a q}$ are precalculated using the Python code in Appendix A.2.

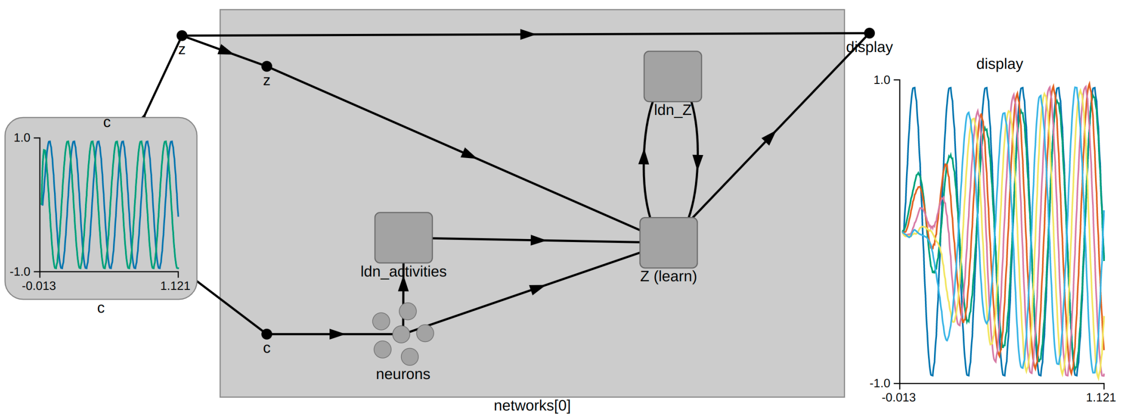


Figure 5.1: A screenshot from the Nengo GUI. The LLP system implemented in Nengo, learning to predict a sine wave online. Within 1sec it is able to begin predicting the future window of the signal, given the current value and its derivative.

Figure 5.1 shows a view from the Nengo GUI of the LLP learning to predict the future second of a sine wave. With the task of predicting drone position, the available context space grows significantly. At the core there are 28 dimensions of context from the drone's translational and rotational states, $\boldsymbol{x}$, in the world frame, the corresponding 12 dimensional trajectory reference, $\boldsymbol{r}$, and most importantly for the context of predictive control, the 4 dimensional control signal, $\boldsymbol{u}$. With the additional option of LDN encoding parts or all of $\boldsymbol{c}$, the full context signal's dimensionality scales based on the number of Legendre polynomials used, q_x, q_r, and q_u, to represent $\boldsymbol{x}$, $\boldsymbol{r}$, and $\boldsymbol{u}$, respectively. A diagram of the LLPSE with context encoding and decoding is shown in figure 5.2.

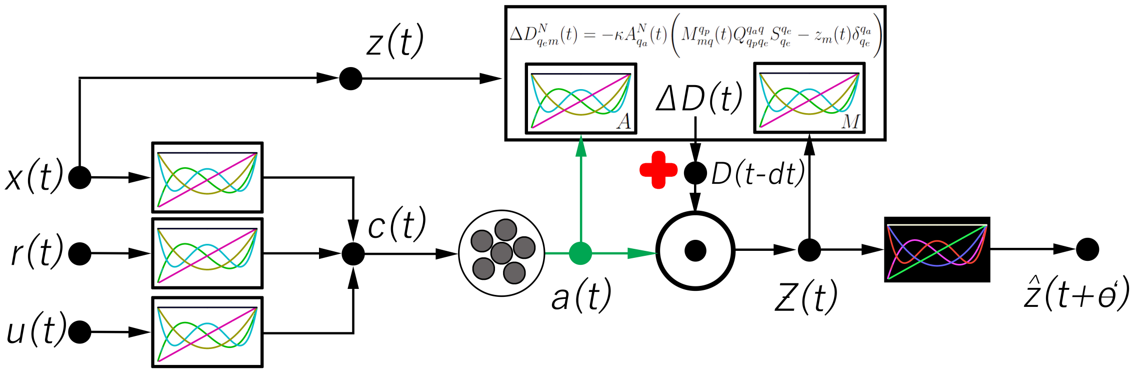

$\Delta D^N_{q_e m}(t) = -\kappa A^N_{q_a}(t)\left(M^{q_p}_{mq}(t)Q^{q_a q}_{q_p q_e}S^{q_e}_{q_e} - z_m(t)\delta^{q_a}_{q_e}\right)$
z(t)
A
ΔD(t)
M
x(t)
r(t)
u(t)
c(t)
D(t-dt)
a(t)
Z(t)
ẑ(t+θ')

Figure 5.2: A single LLP state estimator module with additional LDN encoding of context and decoding of network output. The boxes with polynomials represent an LDN encoding, with the inverted (black box) version signifying an LDN decoding.

In figure 5.2, the six dark circles encompassed by a single larger circle represent an ensemble of neurons. White boxes with polynomials represent an LDN encoding, and black boxes with the inverse colours represent an LDN decoding process. The LLP context, $\boldsymbol{c}$, is formed by LDN encoding $\boldsymbol{x}_{n_x}$, $\boldsymbol{r}_{n_r}$, and $\boldsymbol{u}_{n_u}$ into $\boldsymbol{X}^{q_x}_{n_x}$, $\boldsymbol{R}^{q_x}_{n_r}$, and $\boldsymbol{U}^{q_u}_{n_u}$, and stacking them into a $1 \times (n_x \times q_x \times n_r \times q_r \times n_u \times q_u)$ vector. The quadrotor's position in world coordinates, $[x, y, z]$, is selected as the subcontext to predict, $\boldsymbol{z}_3$ (with $m = 3$). The LLP module generates the prediction in both the Legendre domain, $\boldsymbol{Z}^{q_p}_3$ (with $m = 3$), as well as the decoded value, $\hat{\boldsymbol{z}}_3$. Multiple predictions can be decoded along the continuous prediction horizon, for any θ' value in the domain of $[0, \theta]$. Using the NEF and Nengo, the LLP module can be trained to make these predictions in both an online, or offline manner.

5.2 LLP State Estimator Offline Training

Despite being able to learn online, the LLPSE was trained offline because it simplifies the hyperparameter tuning. This is done in Nengo, using the NEF methods outlined in

Section 3.2. The NEF allows for testing the feasibility of the network's ability to learn the system dynamics. In addition to the vast number of combinations of possible dimensions of context available, there are also many more hyperparameters, including: the length of time (θ) to encode each portion of LDN context, the number of points (θ') to evaluate the prediction, the number of Legendre polynomials to use for the LDN encodings, the learning rate, the number of neurons in the ensemble, and various neuron parameters. Where the LLP receives input data sequentially, the NEF method for weight solving can run a least squares optimization on the entire training dataset. With NEF weight solving, it can be determined whether a set of weights can be solved for to provide the input-output mapping from $\boldsymbol{c}$ to $\boldsymbol{Z}$. This removes certain parameters from the search space, such as learning rate, q_a, and q_p, in addition to speeding up the process of testing various context combinations.

As mentioned in Chapter 3.2, the NEF least-squares method requires a set of evaluation points and target points. During offline training the future θ seconds of subcontext $\boldsymbol{z}$ is available, so a ground truth LDN encoding can be generated. The goal of the system is to predict the Legendre coefficients that represent the future θ length window of time. A slightly modified LDN method is used for the encoding. First, the ground truth $\boldsymbol{z}_m$ is encoded into an LDN representation, with the first θ seconds truncated from the recorded data. This aligns the ground truth so that at time t, the ground truth is the encoded state from time $t + \theta$. This LDN encoded state from θ seconds in the future represents $\boldsymbol{z}_m$ from time t to $t + \theta$. Values can be decoded using equation 3.7.

However, with a naive implementation of the LDN encoding, decoding the value at $\frac{\theta'}{\theta} = 1$ on the time shifted data aligns with the current state, not the future one. The coefficients of the LDN encoding process are set such that decoding with $\frac{\theta'}{\theta}$ from $0 \rightarrow 1$ decodes values in reverse chronological order. This results in $\frac{\theta'}{\theta} = 0$ aligning with the current time, t, and $\frac{\theta'}{\theta} = 1$ decoding from time $t - \theta$. To keep a similar symmetry where $\frac{\theta'}{\theta} = 0$ aligns with 'now', the LLP defines decoding at $\frac{\theta'}{\theta} = 1$ to decode at time $t + \theta$. However, with the naive reimplementation the decoding remains in reverse chronological order. Since the Legendre basis is symmetric, this is circumvented by flipping the signs of the odd coefficients with a simple transform. Note that the equation in 5.1 counts from base zero, resulting in even counter numbers, i, being flipped. The transformation can be written as

$$T_{P_{LDN} \rightarrow P_{LLP}} = [T]_{ii} \in \mathbb{R}^{qxq} \begin{cases} -1, & i\%2 = 0 \\ 1, & i\%2! = 0 \end{cases} \tag{5.1}$$

Using the controller from Chapter 4.2, a simulation was run in Unreal Engine 4 with the AirSim plugin, flying the drone to 10, 000 different targets. The targets are randomly generated from a normal distribution, ranging from -15m to +15m about the drone start in x and y, and a range of -1m to -31m in z. The reference path outlining the ideal trajectory is generated with start and target velocities of $0m/s$, a maximum velocity of $6m/s$, and an acceleration of $1m/s^2$. The flight data is recorded for the 10,000 target flights at a frequency of 200Hz. With a 5ms time step this equates to a total of over 5 million timesteps of data available for training, validating, and testing. This provided more data than was ultimately needed, with the final system using 80,000 steps for training, and 20,000 for validating.

Chapter 6

Experimental Results

The LDN temporal encoding is an integral part in the LLP learning rule and context encoding. This chapter begins with results of the accuracy in the LDN encoding of the state, reference, control signal, neural activities, and Legendre domain network prediction. The results are used to guide the search space in testing the LLP learning algorithm and state estimator. The learning rule is tested online on an inverted pendulum dynamical system. Due to the increased complexity of the quadrotor dynamics, the LLPSE is trained offline on the quadrotor dataset using the NEF weight solving method. The learned weights are used to test the LLPSE online with the simulated quadrotor. Multiple parallel instantiations of the trained estimator are tested with a distribution of control signals. This is done to explore the input-output mapping the estimator learned with respect to the control space. The error used to evaluate performance in this chapter is the RMSE between the normalized decoded prediction $\hat{\boldsymbol{z}}$, and normalized ground truth, $\boldsymbol{z}$, shifted by θ' to align with the prediction in time. The ground truth is normalized using the same preprocessing discussed in section 4.3. As the network is using the normalized value of $\boldsymbol{z}$ as context, the decoded output, $\hat{\boldsymbol{z}}$, is also in the same normalized range. As such, all references to RMSE are of either an LDN or LLP decoded prediction and the normalized, world frame, time shifted, ground truth.

6.1 LDN Representation Error

Due to the size of the input space of the quadrotor dataset, some preliminary tests are run to minimize the search space of the hyper-parameter sweep with the open source Neural Network Intelligence (NNI) software package. The ability of the LDN to represent the preprocessed values of $\boldsymbol{x}$, $\boldsymbol{r}$, and $\boldsymbol{u}$ is tested by calculating the RMSE between the decoded value and the time shifted ground truth. This narrows the search space of the various q values. Due to the similarity in magnitude and frequency between $\boldsymbol{x}$ and $\boldsymbol{r}$, the results of the parameter sweep for $\boldsymbol{r}$ are omitted.

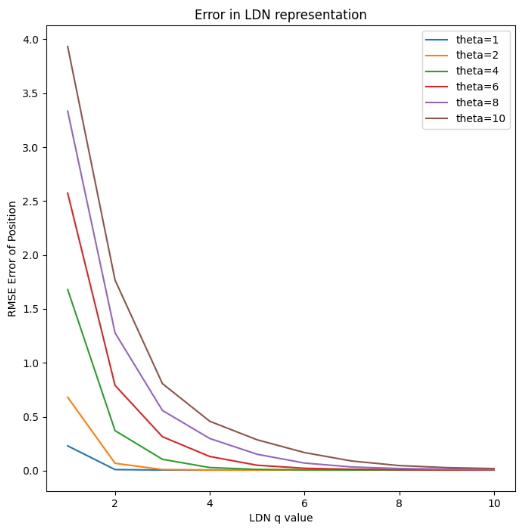


Figure 6.1: RMSE in LDN representation of state position, which is the first three dimensions of $\boldsymbol{x}$. RMSE is taken between $\hat{\boldsymbol{z}}$ decoded at θ, and the time shifted ground truth.

As the length of θ increases, so does the representation error. With larger windows to represent, the LDN requires more Legendre polynomials to maintain a similar level of error compared to shorter length windows with the same q value. This is a result of the increase in frequency of the represented data with respect to the length of the window. With a window length of 1-2 seconds, the LDN has an RMSE less than 0.1 with only 2 Legendre polynomials. The error from the LDN decoding in figure 6.1 is used to guide the search space for q, the dimensionality of the LDN encoded output of the state estimator. Since the network is predicting the system position, it was assumed that it is be of similar frequency. The quadrotor position contains mostly low frequency data due to the linear paths taken between targets and the smoothly changing reference. This allows for the LDN encoding to obtain low errors with few Legendre polynomials. A similar encoding is observed with $\boldsymbol{r}$ due to the similarity in signals. The RMSE of the LDN representation of $\boldsymbol{u}$ is shown in figure 6.2.

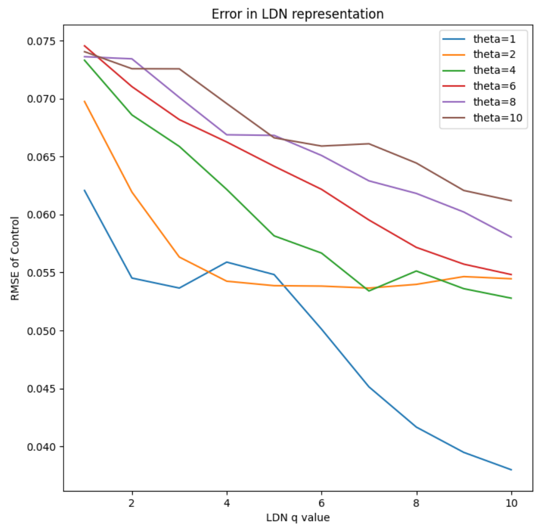


Figure 6.2: RMSE error in LDN representation of the control signal, $\boldsymbol{u}$. RMSE is taken between $\hat{\boldsymbol{u}}$ decoded at θ, and the time shifted ground truth.

A similar pattern is observed in the control signal encoding with respect to the length of the prediction window and the q value. However, a lower error is obtained with a similar LDN when compared to the encoding of position. Most of the control signal consists of small variations about the hover point. The scaling used for normalization increases the range of the control space to the maximum capabilities of the simulated drone. This scales down the amplitude of the variations about the hover relative to the normalized range, lowering the amplitude of the higher frequency terms of the signal. This lower amplitude in the high frequency terms results in less error compared to the state encoding, 0.055 RMSE vs 0.1 RMSE when $q = 2$ for $\boldsymbol{u}$ and $\boldsymbol{x}$, respectively..

In addition to the encoding of input context, the encodings within the learning rule ($\boldsymbol{a}$ and $\boldsymbol{Z}$) are also explored in the same way. To determine a reasonable range for q_p, the encoding error of the ground truth $\boldsymbol{Z}$ is tested. Figure 6.3 shows the RMSE error for the representation of $\boldsymbol{Z}$, encoded with $q_p = 8$. The value for q_p is selected to be larger than required, as figure 6.1 shows, to obtain an accurate representation (less than 0.01 RMSE for θ=1-6). The higher frequency polynomial terms move toward zero if there is not enough high frequency variation in $\boldsymbol{z}$. This allows for the LDN encoding of any higher frequency terms present in the state data.

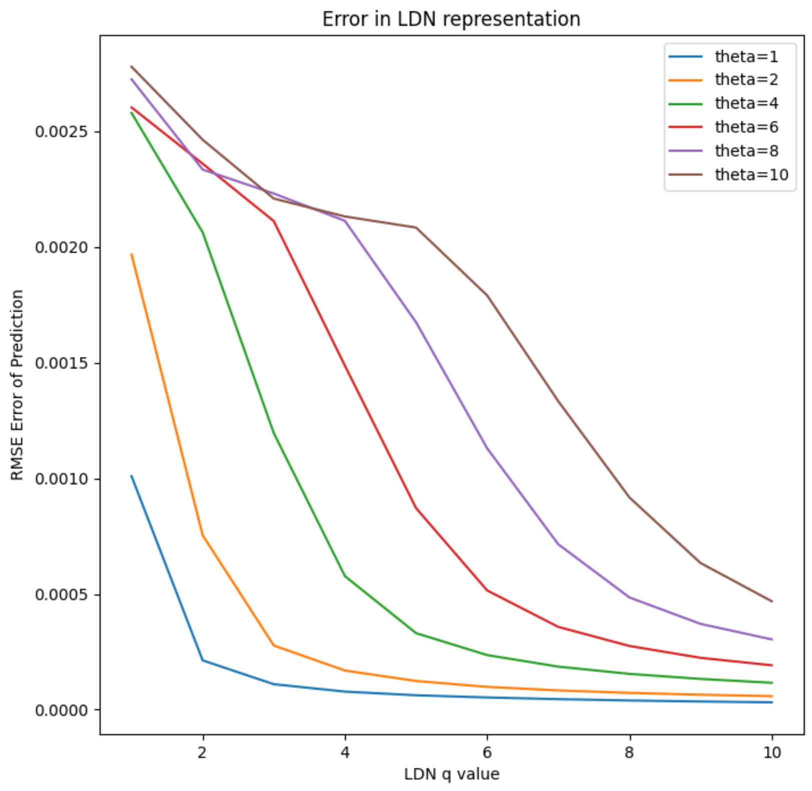
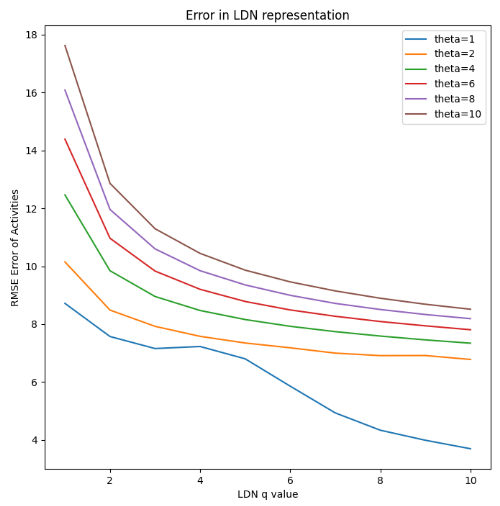


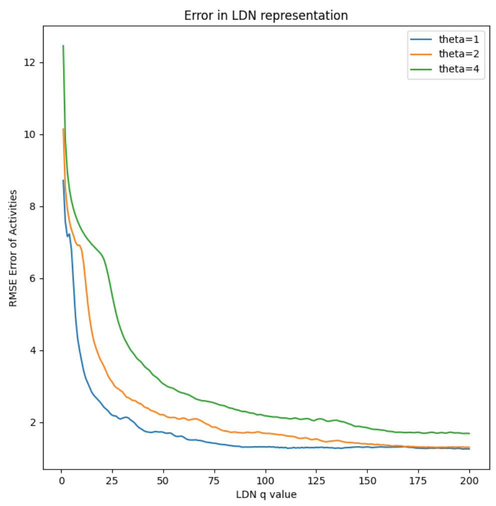
Figure 6.3: RMSE error in LDN representation of the network prediction, $\boldsymbol{Z}$. RMSE is taken between $\hat{\boldsymbol{Z}}$ decoded at θ, and the time shifted ground truth.

The errors of the LDN encoding of $\boldsymbol{Z}$ are an order of magnitude lower than the errors in the context encoding. The small errors in all θ values tested signify that the Legendre polynomials of $\boldsymbol{Z}$ change slowly. As $\boldsymbol{Z}$ is an LDN encoding and uses a polynomial representation of its data, it does not change as quickly as the discrete, stepwise, state would. This allows for a low dimensionality LDN to be used to encode the θ length memory of $\boldsymbol{Z}$.

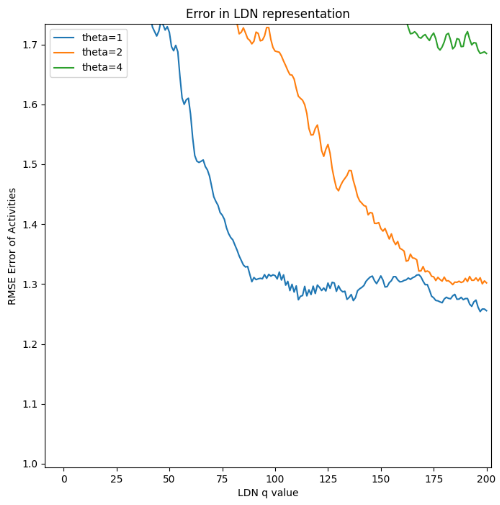
The last encoding tested is the encoding of the activity of 10 randomly selected LIF neurons. As the signal contains much higher frequency data, a larger value is required for q_a to get comparable errors to the previous LDN representations. Figure 6.4a shows the prediction across the same q and θ range as previous figures. The search of q_a values is expanded to 200 (figures 6.4b-c), where the number of Legendre polynomials equals the number of discrete steps being represented in the network prediction when $\theta = 1sec$ and $dt = 0.005sec$. The high frequency data in the neural activity leads to much larger error in representation when compared to the previous LDN encodings with the same q dimensionality. These results determine which parameters are explored for the LLP online and offline learning tasks.



(a)



(b)



(c)

Figure 6.4: RMSE error in the LDN representation of neural activities of ten randomly selected LIF neurons. Subfigure a) shows the same θ and q range as previous tests. Subfgure b) shows the results of expanding the range of q to equal to number of discrete points being represented for $\theta = 1$ ($dt = 5ms$). Subfigure c) is of the same data as b), but zooms into the higher order Legendre representation errors for the expanded q search.

6.2 LLP Online Learning

To demonstrate that the basic implementation is working as expected, the online LLP learning rule is tested with a simpler dynamical system, a single link pendulum. The pendulum is controlled with a P controller (no I or D gains) and given a constant reference of the inverted position. Every 5 seconds a 5Nm perturbing force is added. The LLP is tasked with predicting the angle of the pendulum, without LDN encoding of context. The current pendulum angle and angular velocity, along with the 1D control signal torque, are stacked to form $\boldsymbol{c}$. The Nengo simulation is run with a 1ms timestep. Figures 6.5, 6.6, and 6.7 show the online predictions for θ values of 10ms, 20ms, and 30ms, respectively, with a simulation step of 1ms.

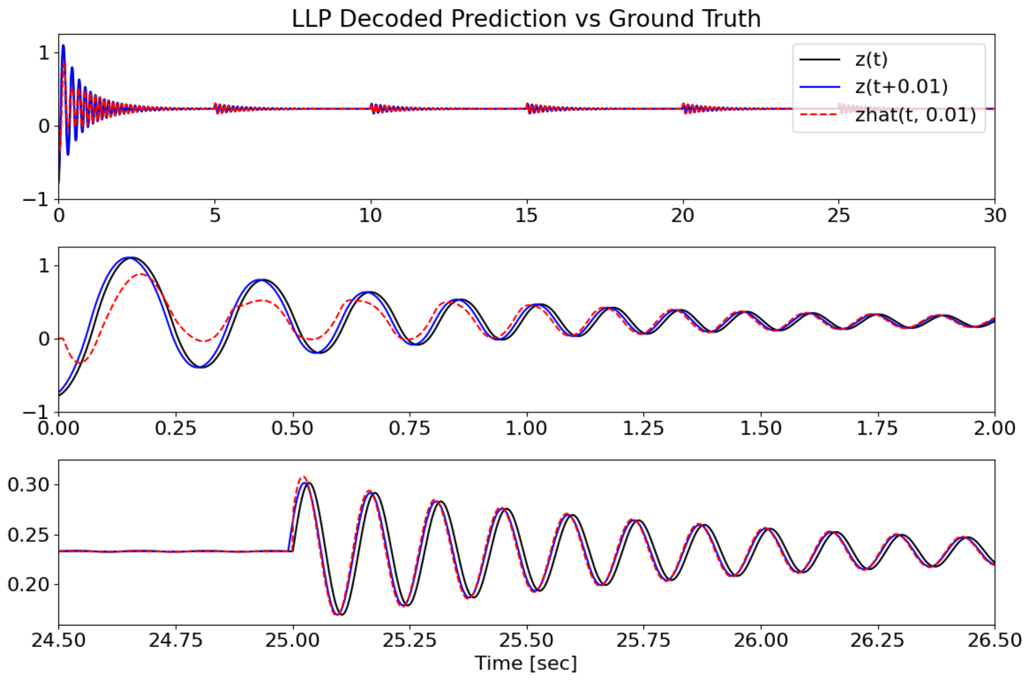


Figure 6.5: The LLP prediction (red) of what the pendulum angle (black) will be θ seconds in the future (blue). From top to bottom: the predictions over the 30 second simulation, the first 2 seconds of predictions, and the predictions during the last application of the perturbing force. The prediction is made for 10 timesteps into the future.

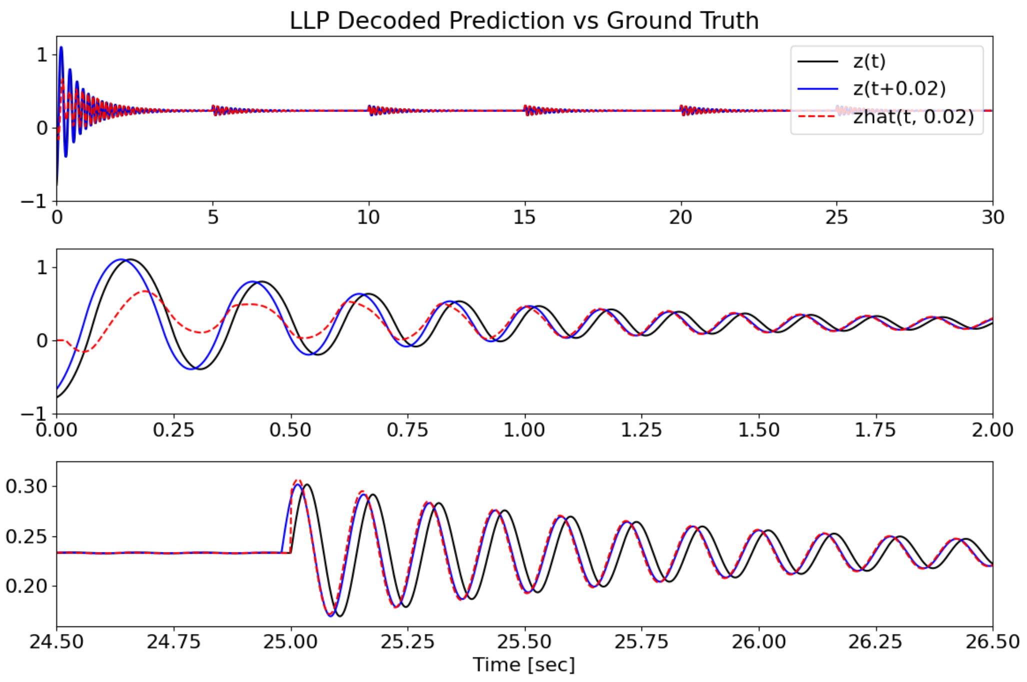


Figure 6.6: The LLP prediction (red) of what the pendulum angle (black) will be θ seconds in the future (blue). From top to bottom: the predictions over the 30 second simulation, the first 2 seconds of predictions, and the prediction during the last application of the perturbing force. The prediction is made for 20 timesteps into the future.

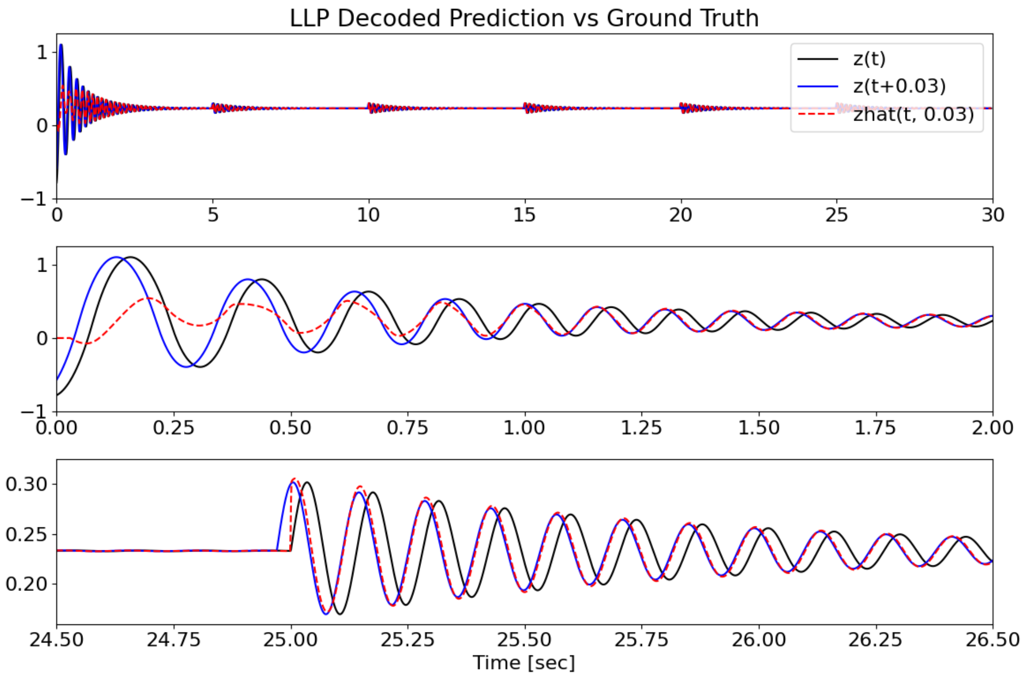


Figure 6.7: The LLP prediction (red) of what the pendulum angle (black) will be θ seconds in the future (blue). From top to bottom: the predictions over the 30 second simulation, the first 2 seconds of predictions, and the prediction during the last application of the perturbing force. The prediction is made for 30 timesteps into the future.

The LLP is set with parameters $q = q_a = q_p = 10$, and 1000 neurons. The learning rates between the 10ms, 20ms and 30ms windows are 1e-3, 5e-4, and 3e-4, respectively. In all cases the LLP stabilizes its prediction about the ground truth within the first second of simulation. The LLP is also able to quickly adapt after the unexpected perturbing forces. Due to the higher frequency of the state data with respect to the timestep compared to the drone state, a higher q_p value is required to maintain a good prediction than that noted in section 6.1. Conversely, a significantly lower value is required for q_a, signifying that the exact activity is not required for learning. The network is able to learn the system dynamics without the need for the higher frequency information in the neural activity.

6.3 LLPSE Offline Training and Testing

Due to the increased complexity of the quadrotor dynamics compared to the 1D inverted pendulum, the ability of an ensemble to learn the input-output mapping is tested with offline training for the quadrotor dataset. The State Estimator is trained with 80,000 consecutive steps of the quadrotor dataset run at a 5ms timestep. The first four figures in this section show the results from running inference on the 20,000 steps following the 80,000 steps that are used for training. Similarly to section 6.1, a preliminary scan across the contextual search space is performed manually. This is used to guide the decision making process of the search space to use for the NNI sweep. Section 6.1 looked into the number of Legendre polynomials required to represent the different parts of the context. The purpose of the preliminary scan in this section is to uncover, approximately, what state pairings of $\boldsymbol{x}$ and $\boldsymbol{r}$ (figures 4.5 and 4.6) provide the lowest RMSE in the prediction of position. The effect of changing θ is also explored to see if the underlying system dynamics have different temporal dependencies between $\boldsymbol{x}$, $\boldsymbol{r}$, and $\boldsymbol{u}$. Finally, the effects of the size of the neural ensemble on the RMSE are tested to explore the trade off between model size and performance.

The following figures list the parameters that were kept constant between tests to the right of the figure, with varying parameters and their corresponding values listed in the legend. Only combinations of $\boldsymbol{x}$ and $\boldsymbol{r}$ are tested in the state estimation task. All dimensions of $\boldsymbol{u}$ are used to allow for prediction across the control space of all four rotors. The state and reference dimensions listed in the legends follow the order $[x, y, z, \dot{x}, \dot{y}, \dot{z}, \alpha, \beta, \gamma, \dot{\alpha}, \dot{\beta}, \dot{\gamma}]$. The RMSE is calculated across ten evenly spaced θ'/θ values in the range of [0.1, 1], where values of 1 correspond to the decoding of the prediction of the world frame position θ seconds in the future. The RMSE is calculated on the normalized state and ground truth data.

Figure 6.8 shows the error that results while only varying the dimensions of $\boldsymbol{x}$ used as context. To search across a single parameter at a time, the LLP State Estimator is set to use the position from $\boldsymbol{r}$, the full control signal, $\boldsymbol{u}$, and encode $\boldsymbol{x}$, $\boldsymbol{r}$, and $\boldsymbol{u}$ with a $\theta_x = \theta_r = \theta_u = 1$ and $q_x = q_r = q_u = 6$. The prediction is made with $q = 6$ and 3000 neurons in the ensemble. Using the position dimensions of $\boldsymbol{x}$ provides the lowest error at 1 second, with the position and linear velocity performing slightly worse. At first glance it seems odd that the addition of velocity context results in a worse prediction. Knowledge of the velocity signifies the direction of motion, and the magnitude of the next step. However, as the LDN encoding stores the temporal context of its input, it innately has some underlying representation of how the signal changes over time. At least in the quadrotor dataset with a 1 second prediction horizon, the addition of velocity

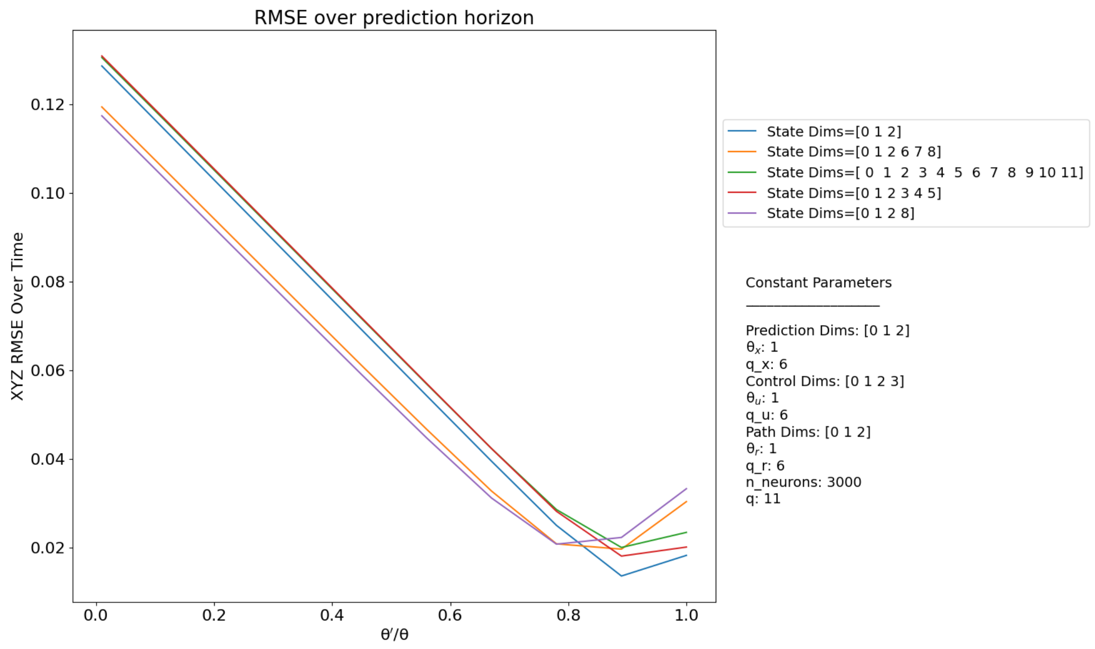


Figure 6.8: RMSE averaged over time for 10 decoded values along the prediction horizon. The dimensions of the state context, $\boldsymbol{x}$, vary between tests. The x-axis shows how far into the prediction horizon an output is decoded, and is defined by θ'/θ. A value of 0 is a prediction of the current time. A value of 1 is a prediction θ seconds into the future.

context leads to a slightly higher prediction error (0.019 vs 0.021 RMSE). One notable point for every curve in figure 6.8 is how the prediction accuracy improves further along the prediction horizon. It is expected that smaller θ' values yield more accurate predictions as the uncertainty in the prediction increases over time. By providing the planned reference θ seconds in the future, the network may use that as an anchoring point to base its prediction around. As the PD controller follows the reference rather closely, this could lead to better predictions when they are made closer in time to the provided reference.

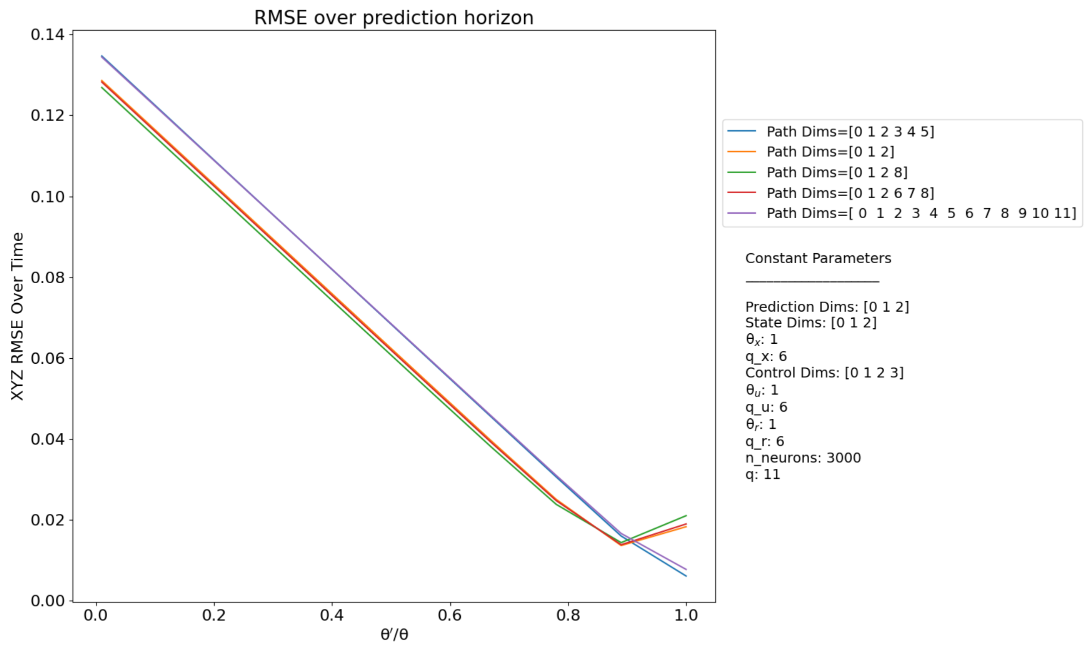
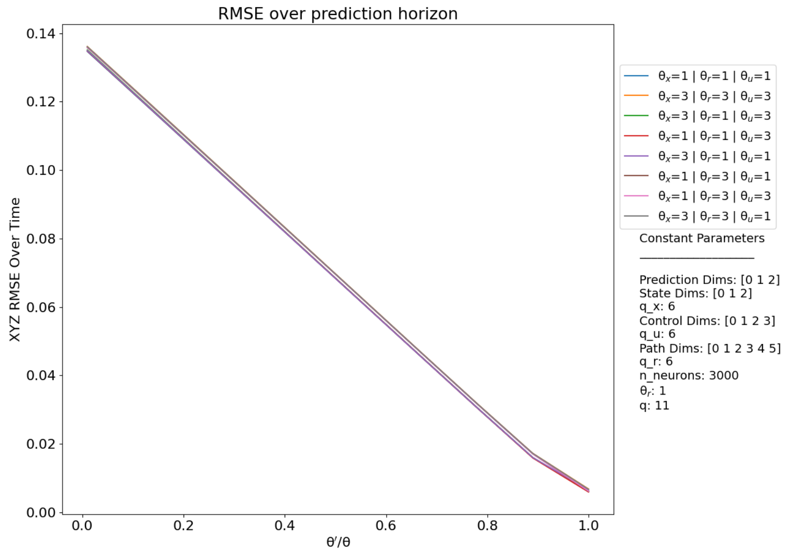
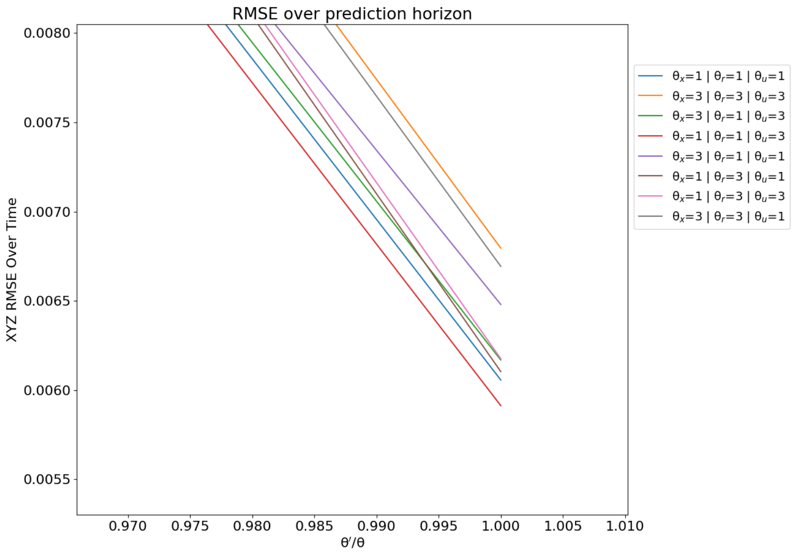


Figure 6.9: RMSE averaged over time for 10 decoded values along the prediction horizon. The dimensions of the reference context, $\boldsymbol{r}$, vary between tests. The x-axis shows how far into the prediction horizon an output is decoded, and is defined by θ'/θ. A value of 0 is a prediction of the current time. A value of 1 is a prediction θ seconds into the future.

Using the position from $\boldsymbol{x}$, the dimensions of $\boldsymbol{r}$ are explored in figure 6.9. Maintaining the same constant values as in figure 6.8, the prediction with the lowest error is the one that has position and linear velocity context of $\boldsymbol{r}$. Conversely to the state context, the addition of linear velocity context from $\boldsymbol{r}$ improves the RMSE over position alone from 0.019 to 0.007. Using the best performing parameters, a small scan of θ values is performed, with results shown in figure 6.10a. Figure 6.10b shows the last section of the prediction window to better differentiate between θ values.



(a)



(b)

Figure 6.10: RMSE averaged over time for 10 decoded values along the prediction horizon. The values of θ_x, θ_r, and θ_u vary between tests. The prediction towards the end of the θ window is shown in b) to better differentiate between results.

As can be seen from figure 6.10, the best performing LLPSE used a $\theta_x = \theta_r = 1$ and a $\theta_u = 3$. This may signify an increased importance in control context history in the state prediction. However, the difference between errors is quite small (0.0058 and 0.0061 RMSE between $\theta_u = 3$ and θ_u=1) so requires further analysis.

Although the LLPSE predictions may continue to improve in performance by increasing the number of neurons, the LDN encoding dimensionality, or the amount of context provided, the cost of running more complex simulations has to be taken into account. With the ultimate goal of running many predictions across the control space in real-time, low cost simulations become increasingly important. To explore the tradeoff in accuracy versus neuron count, the size of the ensemble used is explored next. For consistency, a constant θ of 1 second between the three types of context is used to explore the effects of ensemble size on the prediction accuracy. The number of neurons is also explored to see how small the network can be made while maintaining accurate predictions.

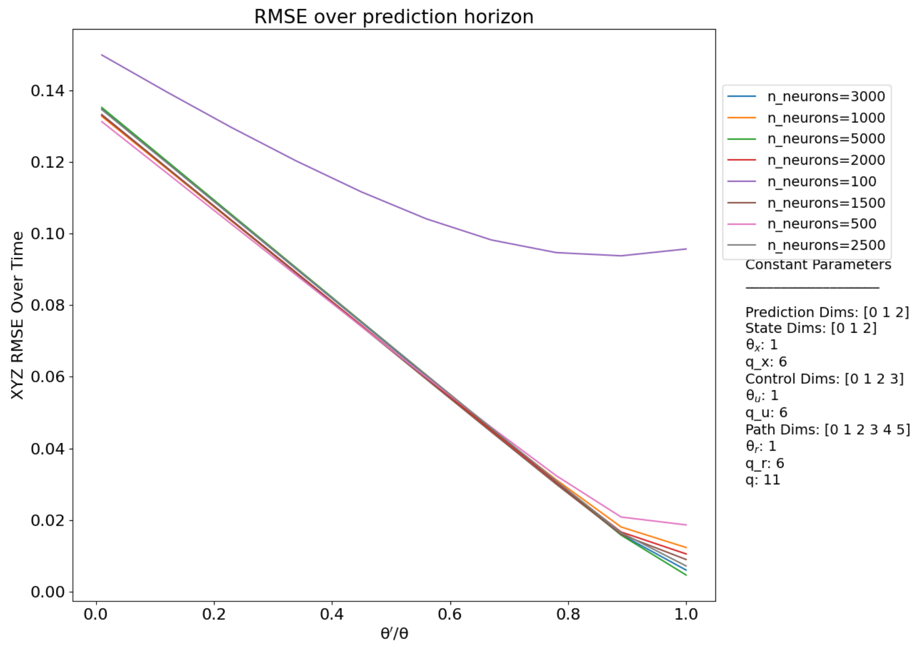
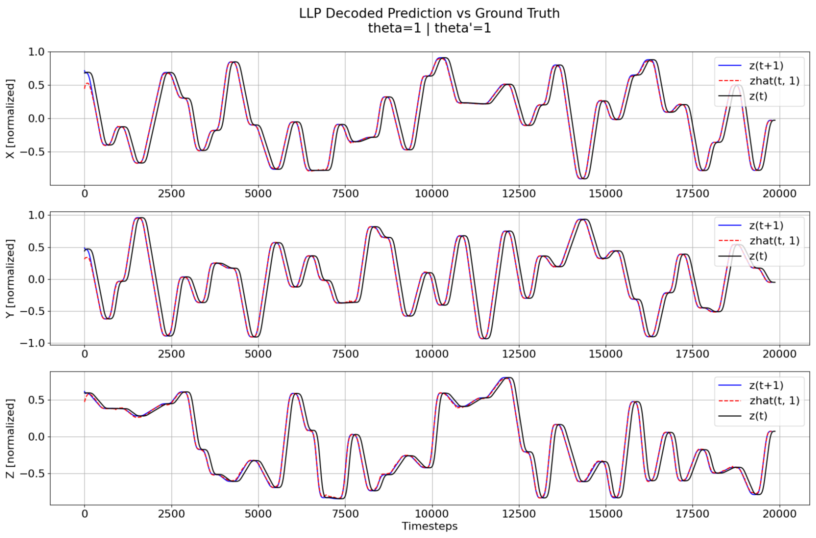


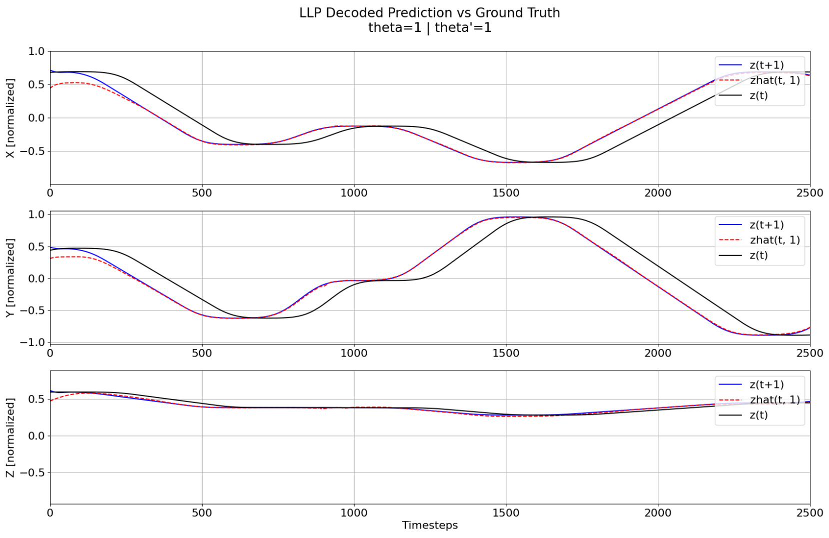
Figure 6.11: RMSE averaged over time for 10 decoded values along the prediction horizon. The number of neurons between tests varies, as noted in the legend. The x-axis shows how far into the prediction horizon an output is decoded, and is defined by θ'/θ. A value of 0 is a prediction of the current time. A value of 1 is a prediction θ seconds into the future.

As shown in figure 6.11, the network is able to maintain predictions with an RMSE less than 0.01 using 1500 neurons. A network with 500 neurons is still capable of making predictions with an RMSE of approximately 0.02. This provides a single parameter that can be used to adjust the network accuracy at the cost of computation. This allows the predictor to be adapted to different control needs and hardware limitations.

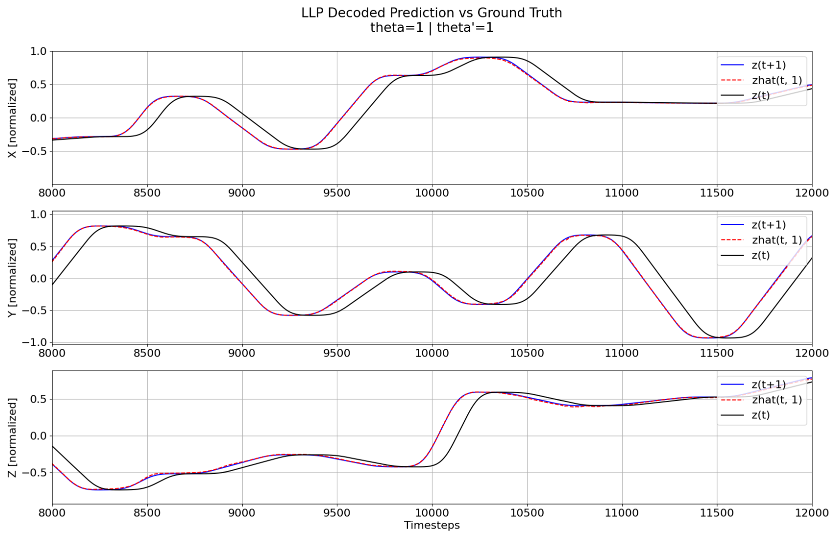
Using the results from figures 6.8 to 6.11, and from section 6.1, a parameter sweep is performed using NNI. In all tests, the LLPSE predicts the normalized, world state position. The same RMSE error is used to evaluate the performance of the predictor, taking the error between the normalized, decoded state prediction, and the normalized, time shifted ground truth. The RMSE is summed across the x, y, and z dimensions, and across time for the simulation. No additional weighting is added, with each dimension having an equal weight. The summed RMSE is used as the cost in the NNI optimization (using TPE as in the PD gain sweep). A θ value of 1 second is selected for the prediction horizon. The scan is performed across the search space listed in table 6.1. The final selected parameters are listed in the 'Selected Parameters' column. The NNI parameter sweep resulted in a q value of 11, but it provided a minor improvement compared to using $q_c = q = 5$. To speed up simulations and lower the number of parameters, the value of q is set to 5. The prediction of the LLP State Estimator is shown in figure 6.12, trained on the same 80,000 steps of the quadrotor dataset, and validated on timesteps 80,000-100,000. The RMSE of predictions along x, y, and z, along with the average across dimensions, are shown in figure 6.13. The RMSE across the prediction horizon with a 95% confidence interval is shown in figure 6.14. In this figure, the weights from training on steps 0-80,000 are used in 19 tests. The tests are run on different sections of the dataset, covering the entire 865,896 points. Each test has an RMSE calculated for each decoded prediction, averaged over time. The confidence interval is calculated over the 19 tests and is shown in the translucent filled area of figure 6.14.


LLP Decoded Prediction vs Ground Truth
theta=1 | theta'=1
X [normalized]
Y [normalized]
Z [normalized]
z(t+1)
zhat(t, 1)
z(t)
Timesteps

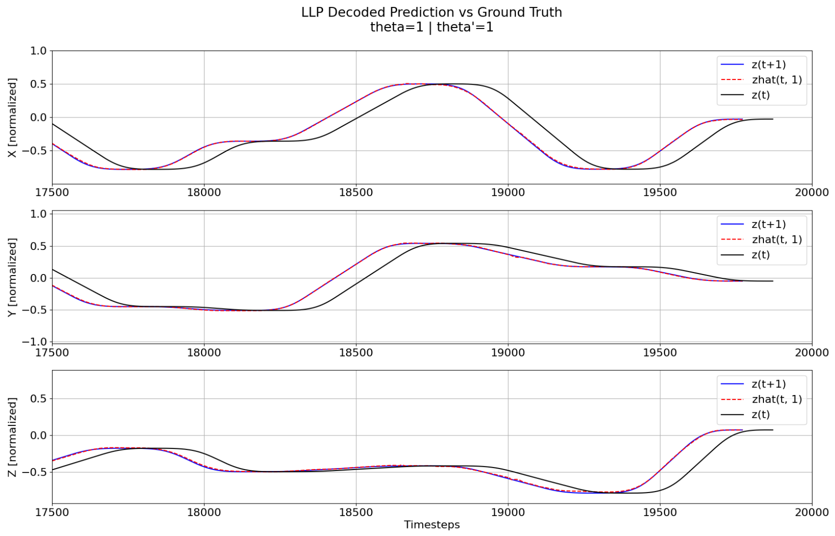
(a)


LLP Decoded Prediction vs Ground Truth
theta=1 | theta'=1
X [normalized]
Y [normalized]
Z [normalized]
z(t+1)
zhat(t, 1)
z(t)
Timesteps

(b)



(c)



(d)

Figure 6.12: The LLP prediction (red) of what the x, y, and z normalized world reference frame position (black) will be 1 second in the future (blue). Subfigure a) shows the predictions over the entire 20,000 steps. Subfigures b-d show different slices to better differentiate between the ground truth and predictions.

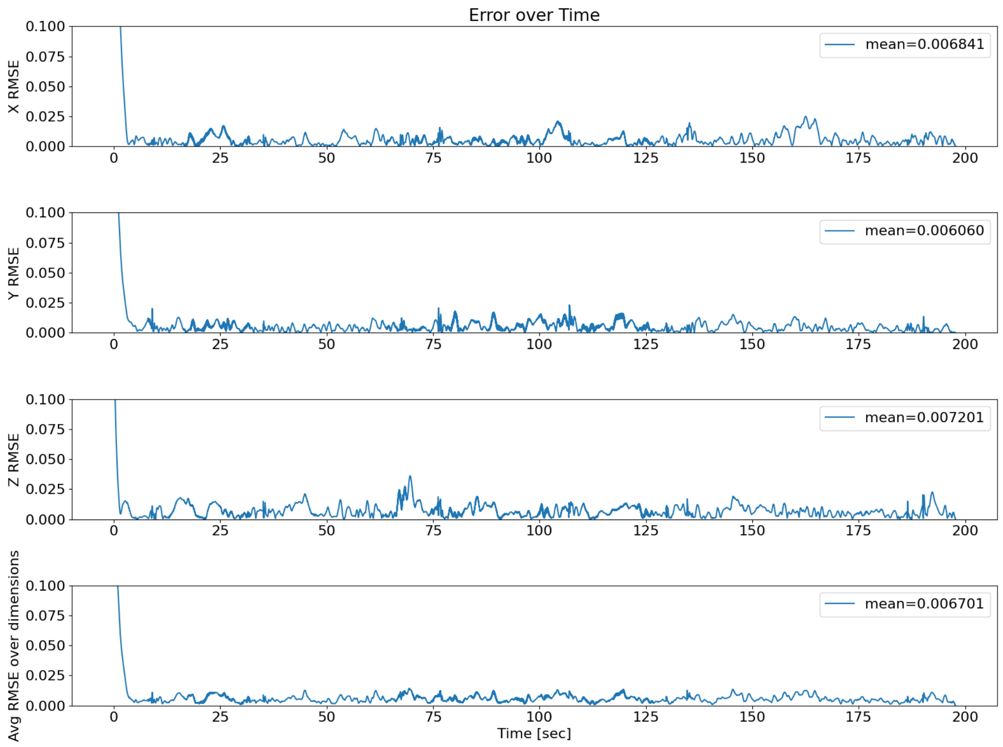


Figure 6.13: The error between the normalized, decoded prediction and ground truth. From top to bottom, the RMSE of the prediction along x, y, z, and the average RMSE of the 3D position $[x, y, z]$. The mean of each error over time is listed in the legend.

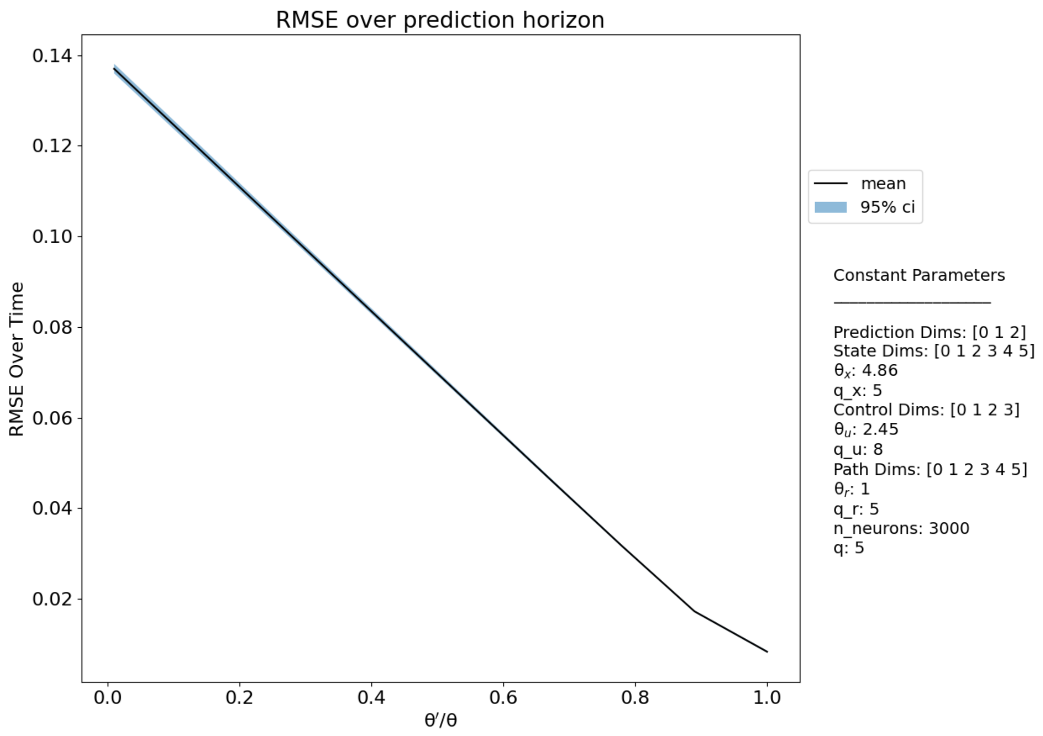


Figure 6.14: The error between the normalized, decoded prediction and ground truth. The filled in translucent sections signify the 95% confidence interval calculated over 19 tests. The 19 tests span the drone dataset from time steps 100,000 to 865,896, and use weights from training on steps 0 to 80,000. The x-axis shows how far into the prediction horizon an output is decoded, and is defined by θ'/θ. A value of 0 is a prediction of the current time. A value of 1 is a prediction θ seconds into the future.

Table 6.1: Search space for LLP State Predictor offline training.

Parameter	**Distribution**	**Value**	**Selected Parameters**
N	Random Int	[1000, 3000]	3000
$\boldsymbol{x}_{dims}$	Choice	[[position], [position + orientation], [position + yaw], [position + velocity], [position + velocity + orientation]]	[position + velocity]
$\boldsymbol{r}_{dims}$	Choice	[[position], [position + orientation], [position + yaw], [position + velocity], [position + velocity + orientation]]	[position + velocity]
q	Random Int	[1, 16]	5
q_x	Random Int	[1, 16]	5
q_r	Random Int	[1, 16]	5
q_u	Random Int	[1, 16]	8
θ_x	Random Uniform	[0.1, 10]	4.86
θ_r	Random Uniform	[0.1, 10]	1.28
θ_u	Random Uniform	[0.1, 10]	2.45

The final LLP State Estimator has an average RMSE over the position dimensions of 0.0067. The context of $\boldsymbol{x}$ has the longest θ value of 4.86 seconds. Additionally, despite the results of the initial manual scan, a higher q is required for the control signal compared to state. This suggests that the higher frequency terms in the control signal are important in the state prediction. When optimizing across the search space of table 6.1, the position and velocity context for both $\boldsymbol{x}$ and $\boldsymbol{r}$ lead to the best performing predictor. The 19 tests run over the span of the dataset show that the LLPSE generalizes well, as is evident in the tight bounds of the 95% confidence interval in figure 6.14. The weights from offline learning are used to test the state estimator with the quadrotor and AirSim in the loop in the next section.

6.4 Online State Estimation of the Trained LLPSE

With the goal of using the LLPSE for control, the estimator is tested with a live AirSim simulation of quadrotor flight. The same PD controller is used to fly the drone, while the LLPSE makes predictions of the drone position. To be used for model predictive control, the state estimator would have to make predictions across the control space. Multiple versions of the LLPSE are used with weights trained from the previous section. While a full controller was not developed (see chapter 7.2 on future work), the various state estimators were connected as shown in figure 7.1 in order to test their estimation abilities. The PD controller serves as the baseline control for the context of every estimator. A visualization from AirSim is shown in figure 6.15 of ten decoded predictions within the prediction horizon from a single LLPSE.

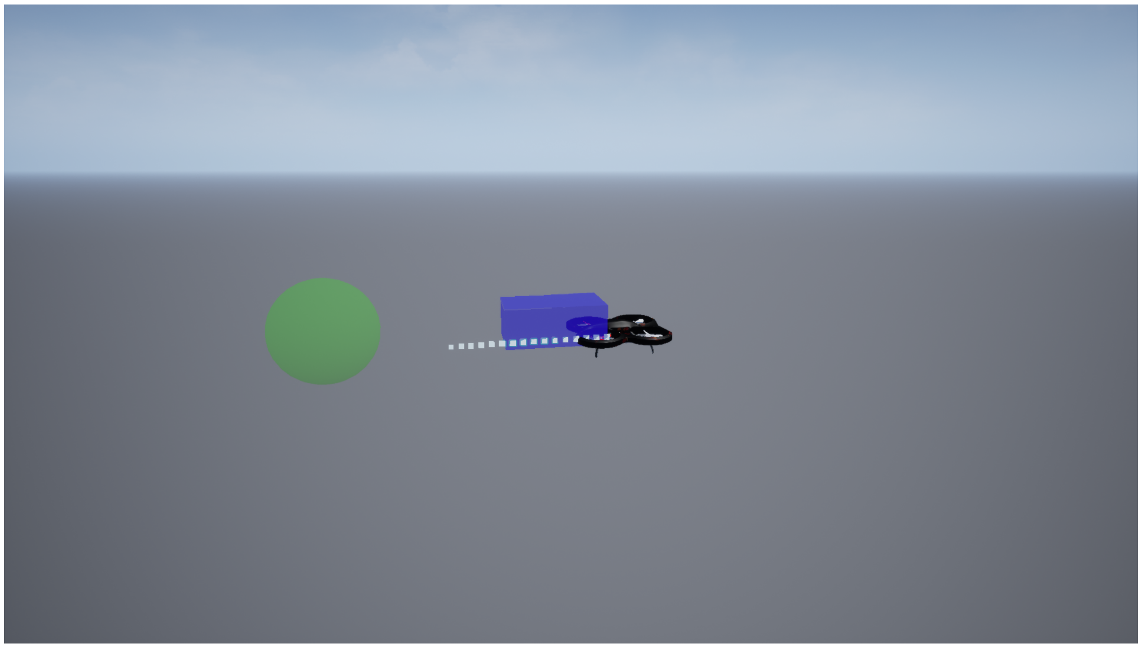


Figure 6.15: The predictions of quadrotor position decoded at 10 points in the prediction horizon. The white dots show the predictions decoded with θ' values from 0.1 - 1 second at 0.1 intervals. The blue box shows the current reference, and the green sphere shows the location of the final target point the reference is converging to.

The various state estimators differ in the control signal used for context. All begin with the same underlying normalized 4D control signal from the PD controller, but differ in the bias applied to it. Here, bias, is defined as a constant vector added to the underlying control signal. For every bias value there are 8 instantiations of the LLPSE. An instantiation refers to an ensemble of neurons making predictions, $\hat{\boldsymbol{z}}$. The 8 instantiations cover the 8 directions of motions: forward, backward, left, right, up, down, clockwise, and counter-clockwise. Each instantiation has its bias arranged in accordance with the rotor pairs that align with its direction of motion (as in figure 2.1). With the exception of one unbiased estimator, a bias is added to the control signal before LDN encoding for the other estimators. The distribution of biases is selected by looking at the range of $\frac{\delta u}{\delta t}$ from the PD controller.

The hypothesis is that given the appropriate bias offset, the estimator can predict the corresponding motion. As an example, if a positive bias was added to all four rotors, the state estimator should predict the position to be higher than in the unbiased estimator. By comparing the errors in the predicted state to the planned reference, the control context of the estimator with the lowest error could potentially be used in selecting the control action for the next time step. For the following figures, the state estimators are used exclusively to predict state, with no consequence on the resulting control action.

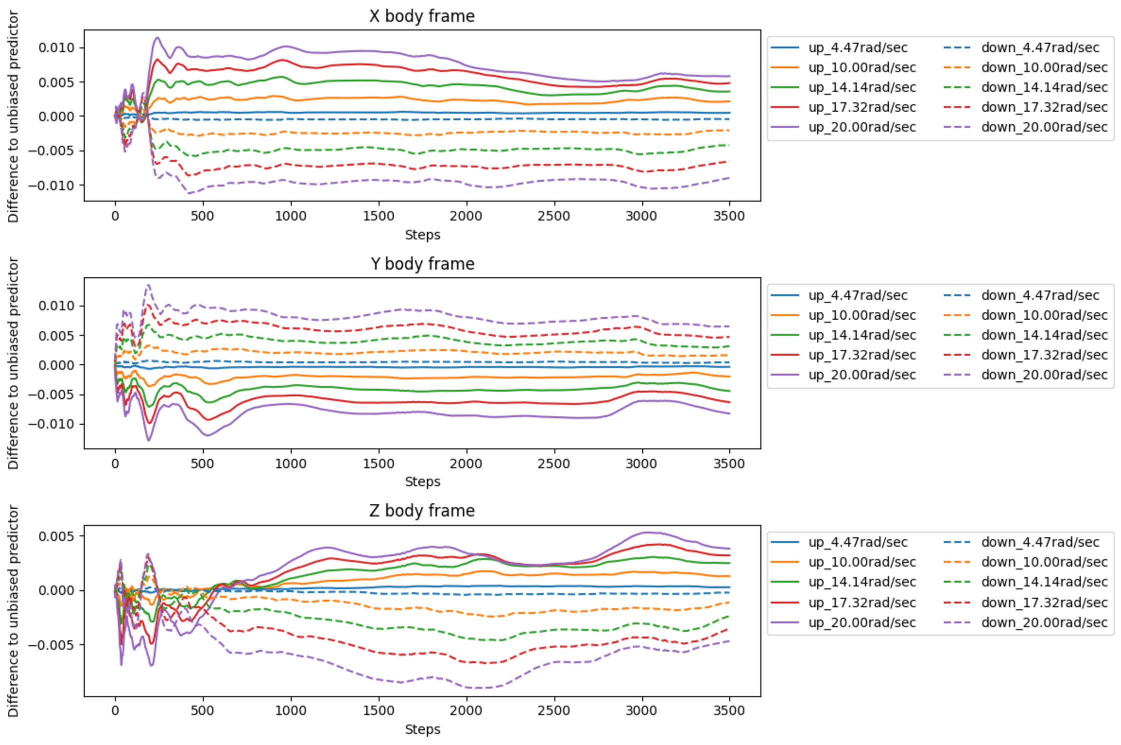


Figure 6.16: The difference in decoded outputs of biased state estimators compared to the unbiased estimator, comparing the biased estimators with up and down rotor pairings. The absolute value of the bias is listed in the legend. Although the actual bias added is normalized, the legend shows the equivalent offset converted to radians/second.

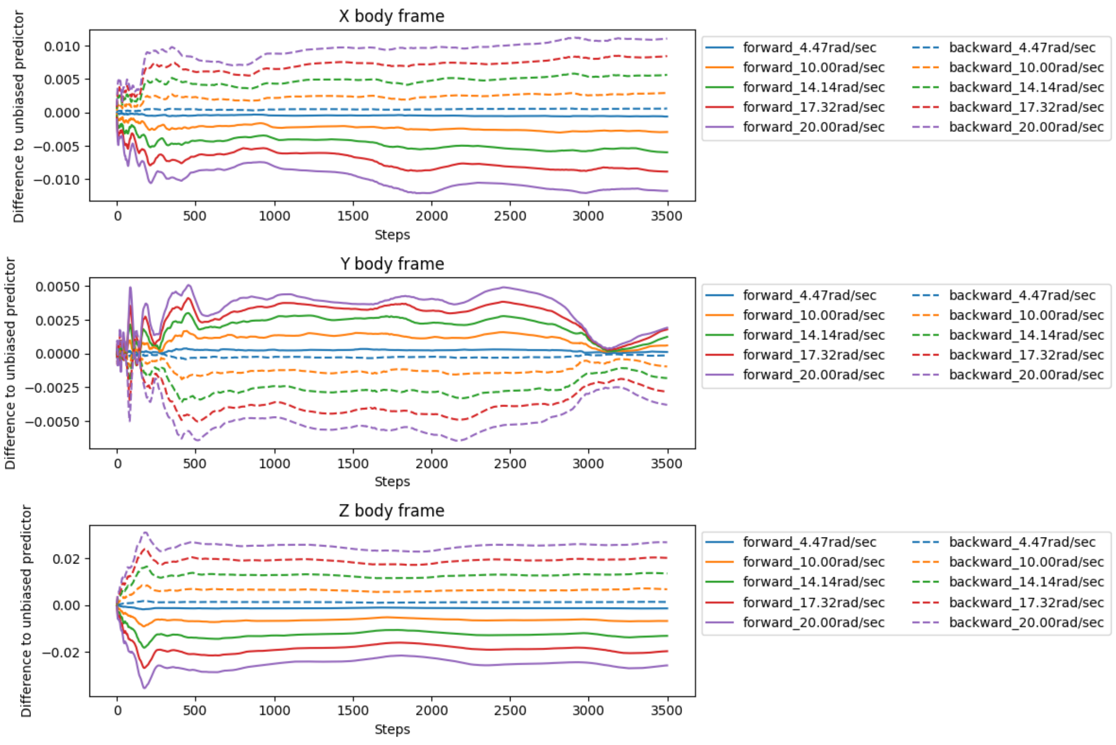


Figure 6.17: The difference in decoded outputs of biased state estimators compared to the unbiased estimator, comparing the biased estimators with forward and backward rotor pairings. The absolute value of the bias is listed in the legend. Although the actual bias added is normalized, the legend shows the equivalent offset converted to radians/second.

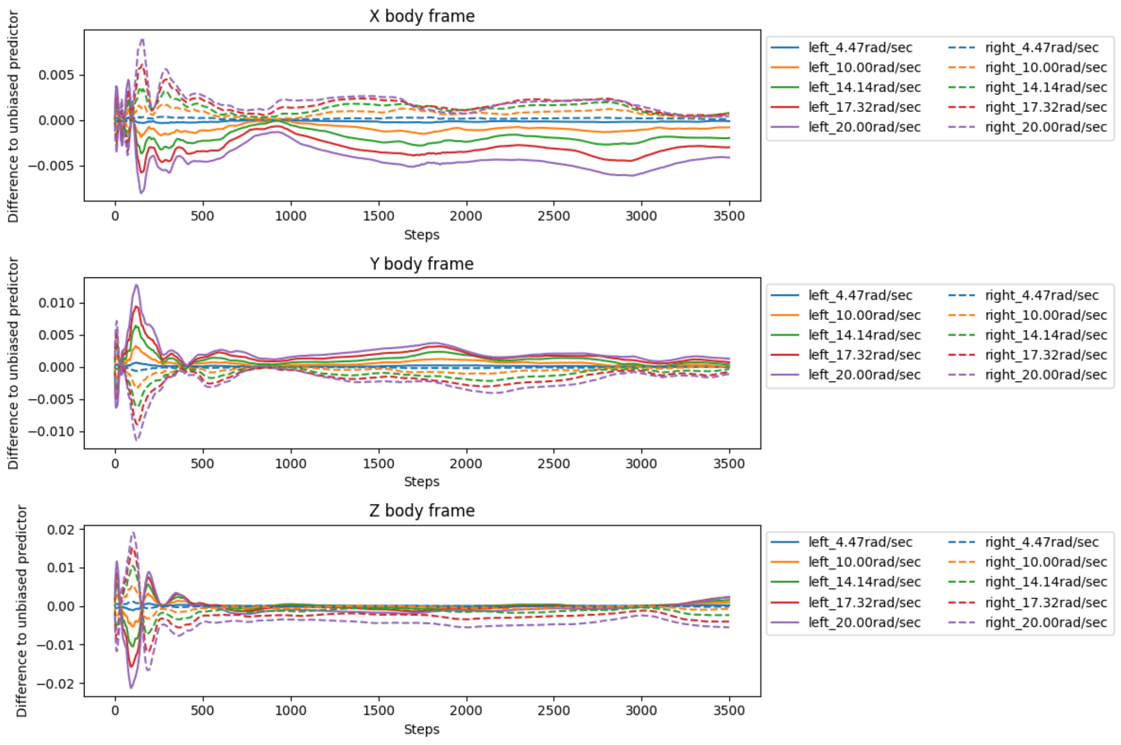


Figure 6.18: The difference in decoded outputs of biased state estimators compared to the unbiased estimator, comparing the biased estimators with right and left rotor pairings. The absolute value of the bias is listed in the legend. Although the actual bias added is normalized, the legend shows the equivalent offset converted to radians/second.

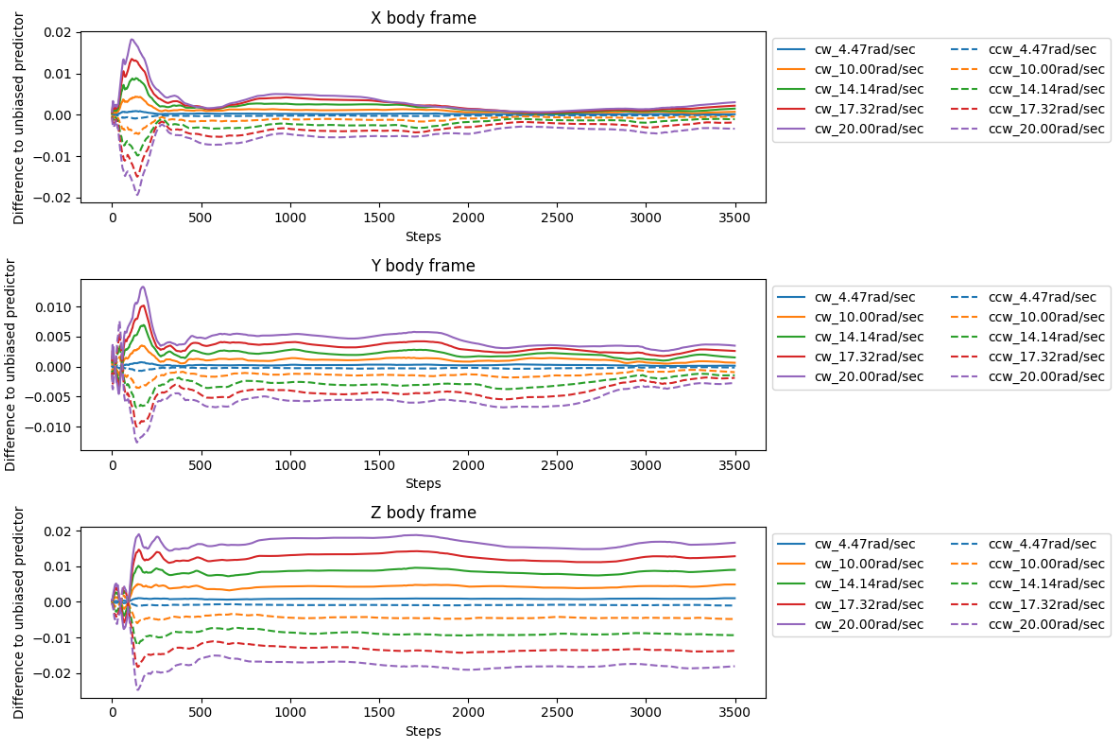


Figure 6.19: The difference in decoded outputs of biased state estimators compared to the unbiased estimator, comparing the biased estimators with clockwise and counterclockwise rotor pairings. The absolute value of the bias is listed in the legend. Although the actual bias added is normalized, the legend shows the equivalent offset converted to radians/second.

Figures 6.16 to 6.19 show the difference between the unbiased and biased state estimator decoded outputs. The drone reference is set to move to 3 targets, moving up 5m, forward 5m, and finally 5m to the right. The yaw is kept constant at 0 rad. The simple reference is chosen to try and isolate the motion along a single controllable DOF at a time. Predictions with like bias types, but larger magnitudes have a larger offset with respect to the unbiased predictor, as expected. The position is transformed into the body frame as the rotor pairings define motion along the body frame. The LLP state estimator is able to learn a mapping between the positive and negative directions of the bias pairing. Predictions with a positive bias (solid lines) are opposite of predictions with an equal, but opposite signed bias (dotted lines). This shows some underlying differentiation between up-down, right-left, forward-backward, and clockwise-counterclockwise. However, it appears that some of the dynamics are not learned successfully. Similar predictions are made along all 3 dimensions, instead of being isolated to the expected dimensions of motion, given the bias rotor pairing. A clearer, uncoupled, mapping between rotor pairings and predicted direction of motion is necessary to use the predictor for selecting the control action.

Chapter 7

Conclusion and Future Work

7.1 Discussion

The LDN provides highly compressed temporal representations for the LLP and LLPSE. With the exception of neural activity, the majority of context and internal learning rule states are well represented with q values less than 6. Interestingly, the LLP is able to predict the inverted pendulum dynamics without a highly accurate neural representation. This signifies that the learning rule can sometimes function successfully with only a running average of the neural activity, as the high frequency data is lost in the LDN compression. The LDN is able to maintain errors below an RMSE of 0.1 for $\boldsymbol{x}$, $\boldsymbol{r}$, $\boldsymbol{u}$, and $\boldsymbol{Z}$ for a prediction window of 1 second, with only 2 Legendre polynomials. With a timestep of 5ms, the 200 discrete data points are compressed to 2 Legendre polynomial coefficients. Even with higher order LDNs, the LDN provides a significant decrease in parameter count compared to memorizing the time history of data.

With the simpler dynamical system, the LLP is able to learn to predict the single link pendulum's state online. The predictions are accurate up to a prediction horizon of 30x the timestep. With the added complexity of learning online versus offline, and due to time constraints, further optimization and longer prediction horizons are not tested, as the accuracy begins dropping with manual parameter tuning of larger θ values.

The offline training on the quadrotor dataset yields highly accurate predictions of positions over a prediction horizon of 1 second as well. The first notable point from figures 6.8 to 6.11 is how the prediction accuracy improves further along the prediction horizon. It is expected that smaller θ' values will yield more accurate predictions. The results may be a side effect of how the network is evaluated during testing. $\theta' = \theta$ was used for calculating the error, and may have pushed the optimization to find parameter sets that better represent the future window than the near window. The NNI optimization selected larger θ_x, θ_r, and θ_u values, which help with predictions made further along the prediction horizon. Alternatively, the network may have learned to more heavily weigh the $\boldsymbol{r}$ subcontext. As the drone position closely follows the reference, using the future reference provides a good baseline to make predictions about.

Despite the highly accurate state predictions made during offline testing, the control biased estimators are not able to make the expected state predictions. The PD controller used to generate the training data, and to drive the system during the online prediction, uses the rotor pairings defined in figure 2.1. With the underlying rotor pairings driving the control actions, it was hypothesized that the LLP state estimator can learn the input-output mapping between the control signal with respect to the pairing between rotors, and the direction along one of the four controllable DOF. This is partially observed with opposite signed biases making predictions in opposite directions with respect to the unbiased predictor. Larger biases also make larger steps in their predictions. However, there is a coupling between predictions along the x, y, and z dimensions. Some coupling is expected due to the underactuated nature of the system. With a non-zero pitch, an increase in control along the 'up' bias type will lead to an increase along both z and x, due to the tilt of the drone causing a component of the thrust to point along x. However, it is expected that the direction of motion with the largest predicted change will align with the corresponding bias type. This is not observed in figures 6.16 to 6.19. Although coupling is expected between x and z, or y and z, similar magnitude predictions are made along both x and y. This signifies that some the underlying control dynamics are not learned during the weight solving process, given the current network parameters.

7.2 Future Work

The LLP State Estimator is able to very accurately predict states when used in tandem with the underlying PD controller driving the system control actions. With an accurate state estimator, a predictive controller can be designed to select control actions based on state estimates. An LLP Controller (LLPC) was designed around the LLPSE, with some preliminary tests run on the state predictions shown in section 6.4. As the estimator, shown in figure 5.2, is designed to use a control signal as part of its context, $\boldsymbol{c}$, when making predictions, multiple state estimators with varying control context can be used to determine which control action will lead to the lowest predicted error. This allows for state predictions across a range of the control space. The LLPC design is shown in figure 7.1. The LLPC was tested with weights determined from offline training, although it is capable of updating network weights online with the LLP learning rule. Due to the poor mapping between rotor pairing control actions and the direction along the controllable DOF, the LLPC was unsuccessful in driving the system towards its reference. Due to time constraints, the LLPC was not further tested or optimized. This section outlines the design of the LLPC, and the flow of information.

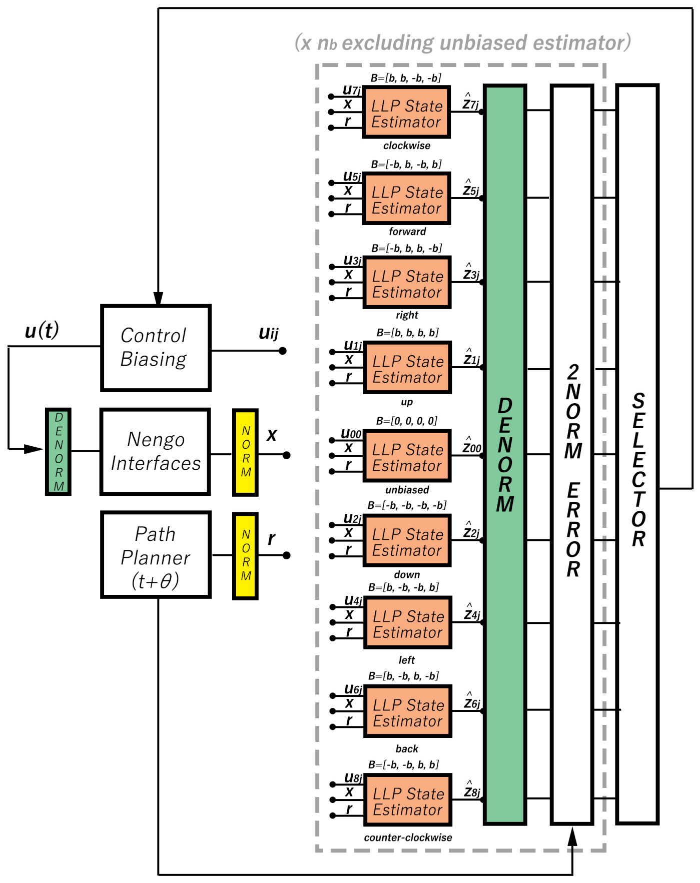


Figure 7.1: Network connections of the LLPC. $\boldsymbol{x}$ and $\boldsymbol{r}$ are passed to every state estimator, and every $\boldsymbol{u_{ij}}$ is passed to the corresponding estimator. The 8 biased estimators are repeated for every bias value. See text for details.

Using the system states, planned path from θ seconds in the future, and a biased control signal, the various LLP State Estimators produce the decoded prediction $\hat{\boldsymbol{z}}_{ij}$, where i notes the index of the bias value in the list of n_b biases, $\boldsymbol{b}$, and j is the type of bias offset. The various state estimators differ in the control signal used for context. All begin with the same underlying control signal, but differ in the bias applied to it. Here bias is referred to as a constant vector added to the underlying control signal. For every bias value there are 8 instantiations of the LLPSE. An instantiation refers to an ensemble of neurons making predictions, $\hat{\boldsymbol{z}}$. The 8 instantiations cover the 8 directions of motions: forward, backward, left, right, up, down, clockwise, and counter-clockwise. Each instantiation has its bias arranged in accordance with the rotor pairs that align with its direction of motion. With the exception of one unbiased estimator, a bias, b_i, is applied to the four rotors following the rotor pairings described in figure 2.1, where rotors labelled 'high speed' receive a positive offset, and rotors labelled 'normal speed' receive a negative offset. The control signal biasing is contained in a Nengo Node, labelled 'Control Biasing' in figure 7.1.

The control biasing node contains a stacked vector of the $1 + 8 \times n_b$ LLPSE normalized control signal inputs, before the LDN encoding. The baseline control that the biases are added to for the various state estimators is the control signal that is used to control the quadrotor on the previous timestep. The predictions, $\hat{\boldsymbol{z}}_{ij}$, are compared to the planned trajectory, $\boldsymbol{r}$, to get the 2-norm error of position. The errors from the various state estimators are passed on to a selector node. The selector node outputs the index of the state estimator that has the lowest predicted error θ seconds into the future. Depending on the index of the LLP state estimator with the lowest predicted error, the control biasing node outputs the corresponding biased control signal, $\boldsymbol{u}(t)$. The control signal is 'denormalized' by applying a scaling and a mean shift, before being passed on to the AirSim wrapper code contained in Nengo Interfaces. The normalized output, $\boldsymbol{u}(t)$, is passed with the appropriate biasing for the ij LLP state estimators to be used as part of the input context in making the state prediction for the next time step.

The biases are added to the control signal that is last sent to control the quadrotor. However, the control biasing node needs to be seeded with some starting control signal. Instead of seeding the biasing node to begin with a base control signal of all zeros, the mean shifted and scaled control signal for maintaining a hover is used. For the first θ seconds of simulation time, the control biasing node outputs the hover control signal as the baseline to make predictions on. This allows the state estimators to stabilize in their predictions. As the input context is encoded with LDNs, it requries approximately θ seconds to begin making stable predictions θ seconds into the future. After the first θ seconds of simulation time, the control is governed by the control signal determined by the selector node. Further work can be done by exploring the mapping between the control biasing and the corresponding directions of motion. A deeper statistical analysis with correlation measures would provide insight towards the relative importance between context dimensions. With improved accuracy in the mapping of the control space to the expected motion, the LLPC can make better decisions in selecting the control action.

7.3 Contributions

The main contribution of this thesis is a state predictor for dynamical systems using the NEF, LLP, and LDN. This is a proof-of-concept that an ensemble of neurons can make predictions in the Legendre domain about the future trajectory of a dynamical system. The state predictor also does so by using temporal representations of its context, encoded by LDNs. The second contribution is the Python implementation of the LLP learning algorithm. This is done using the Numpy *einsum* function due to the high dimensionality of the tensors involved in the learning rule. The LLP network is developed in Nengo for easy integration into larger networks. This allows for simple integration and expansion of the predictor. The third contribution is the design and implementation of a model predictive controller using the LLP state estimator. This lays the ground work for how a predictive controller can be designed to make continuous, temporal predictions, using a similarly encoded context as input. Similar to the many branches of MPC, various alternative MPC methods can be used to take advantage of the temporal context, and LLP predictions. The fourth contribution is a test suite developed to integrate the predictor and controller with Unreal Engine 4 and AirSim. Additionally, a dataset logger was developed to store, search, and compare across parameter sets and results. Combined, this system allows for fast and simple testing of different context and alternative control networks. Developing in AirSim allows for a simple expansion into a much broader range of sensory input, including various cameras, that can be used to expand the context available to train the LLPSE and LLPC.

7.4 Conclusion

The implementation of the LLP is capable of learning the dynamics of simple systems online. The angle of an inverted pendulum driven by a PD controller is accurately predicted online for prediction horizons up to 30x the timestep. Due to the extra complexity and size of context space for the quadrotor dynamics, the LLP state estimator is trained offline. With offline training the state estimator is capable of making predictions of where the quadrotor's position in world space will be 1 second into the future, with an average RMSE of 0.0067. The RMSE is calculated on the normalized ground truth and predictions, and thus provides an error relative to the range of the ground truth. The LLPSE bases the predictions off temporal encodings of the drone state, reference, and control context signals. A preliminary LLPC is designed for future work, and is integrated into the test suite. All of the code for simulator interfacing, controllers, path planning, the LLP systems, and various utility functions are available at https://github.com/p3jawors/masters_thesis.

References

[1] Frontier supercomputer debuts as world's fastest, breaking exascale barrier. https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier. Accessed: 2022-08-02.

[2] Inverted pendulum. https://drive.google.com/file/d/1Jpje3UWRr-ZvVKR1SlcwoM-KI1OGBNYA/view?usp=sharing. Accessed: 2022-07-10.

[3] Charles W Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

[4] Vijay Balasubramanian. Brain power. *Proceedings of the National Academy of Sciences*, 118(32):e2107022118, 2021.

[5] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[6] Narsimha Chilkuri, Eric Hunsberger, Aaron Voelker, Gurshaant Malik, and Chris Eliasmith. Language modeling using lmus: 10x better data efficiency or improved scaling compared to transformers. *arXiv preprint arXiv:2110.02402*, 2021.

[7] Sanghyeon Choi, Seonggil Ham, and Gunuk Wang. Memristor synapses for neuromorphic computing. *Memristors-Circuits and Applications of Memristor Devices*, pages 1–14, 2019.

[8] Travis DeWolf, Pawel Jaworski, and Chris Eliasmith. Nengo and low-power ai hardware for robust, embedded neurorobotics. *Frontiers in Neurorobotics*, 14:568359, 2020.

[9] Travis DeWolf, Terrence C Stewart, Jean-Jacques Slotine, and Chris Eliasmith. A spiking neural model of adaptive arm control. *Proceedings of the Royal Society B: Biological Sciences*, 283(1843):20162134, 2016.

[10] Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT press, 2003.

[11] Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, 2012.

[12] P. Michael Furlong, Andreas Stöckel, Terry Stewart, and Chris Eliasmith. Learned legendre predictor: Learning with compressed representaitons for efficient online multistep prediction. Technical report, Centre for Theoretical Neuroscience, 08/2022 2022.

[13] Hongliang Gao, Xiaoling Li, Chao Gao, and Jie Wu. Neural network supervision control strategy for inverted pendulum tracking control. *Discrete Dynamics in Nature and Society*, 2021, 2021.

[14] Drew Hanover, Philipp Foehn, Sihao Sun, Elia Kaufmann, and Davide Scaramuzza. Performance, precision, and payloads: Adaptive nonlinear mpc for quadrotors. *IEEE Robotics and Automation Letters*, 7(2):690–697, 2021.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[16] Brent Komer. Biologically inspired adaptive control of quadcopter flight. Master's thesis, University of Waterloo, 2015.

[17] Mario Lanza, Abu Sebastian, Wei D Lu, Manuel Le Gallo, Meng-Fan Chang, Deji Akinwande, Francesco M Puglisi, Husam N Alshareef, Ming Liu, and Juan B Roldan. Memristive technologies for data storage, computation, encryption, and radiofrequency communication. *Science*, 376(6597):eabj9979, 2022.

[18] Teppo Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 22:22, 2011.

[19] Jahangir Moini, Nicholas Avgeropoulos, and Mohtashem Samsam. *Epidemiology of Brain and Spinal Tumors*. Academic Press, 2021.

[20] Caitlin Powers, Daniel Mellinger, and Vijay Kumar. Quadrotor kinematics and dynamics. *Handbook of Unmanned Aerial Vehicles*, pages 307–328, 2015.

[21] Arjun Rao, Philipp Plank, Andreas Wild, and Wolfgang Maass. A long short-term memory for ai applications in spike-based neuromorphic hardware. *Nature Machine Intelligence*, 4(5):467–479, 2022.

[22] Anders Sandberg. Feasibility of whole brain emulation. In *Philosophy and Theory of Artificial Intelligence*, pages 251–264. Springer, 2013.

[23] Catherine D Schuman, Shruti R Kulkarni, Maryam Parsa, J Parker Mitchell, Bill Kay, et al. Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1):10–19, 2022.

[24] Max Schwenzer, Muzaffer Ay, Thomas Bergs, and Dirk Abel. Review on model predictive control: An engineering perspective. *The International Journal of Advanced Manufacturing Technology*, 117(5):1327–1349, 2021.

[25] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, pages 621–635. Springer, 2018.

[26] Terrence C Stewart. A technical overview of the neural engineering framework. *University of Waterloo*, 110, 2012.

[27] Zaid Tahir, Waleed Tahir, and Saad Ali Liaqat. State space system modelling of a quad copter uav. *arXiv preprint arXiv:1908.07401*, 2019.

[28] Arun Venkatraman, Roberto Capobianco, Lerrel Pinto, Martial Hebert, Daniele Nardi, and J Andrew Bagnell. Improved learning of dynamics models for control. In *International Symposium on Experimental Robotics*, pages 703–713. Springer, 2016.

[29] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre memory units: Continuous-time representation in recurrent neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[30] Aaron R Voelker and Chris Eliasmith. Improving spiking dynamical networks: Accurate delays, higher-order synapses, and time cells. *Neural Computation*, 30(3):569–609, 2018.

[31] Aaron Russell Voelker. A solution to the dynamics of the prescribed error sensitivity learning rule. *Centre for Theoretical Neuroscience, Waterloo*, 2015.

[32] Liuping Wang. *Model predictive control system design and implementation using MATLAB®*. Springer Science & Business Media, 2009.

[33] Grady Williams, Andrew Aldrich, and Evangelos A Theodorou. Model predictive path integral control: From theory to parallel computation. *Journal of Guidance, Control, and Dynamics*, 40(2):344–357, 2017.

[34] Svetoslav Zabunov, Garo Mardirossian, and Katia Strelnitski. Emerald–a 16-rotor multicopter for stereo imaging. *Aircraft Engineering and Aerospace Technology*, 2021.

APPENDICES

Appendix A

Python functions

A.1 Converting Euler angles into Tait-Bryan form

From [16].

```
def convert_angles(ang):
    """Converts Euler angles from x-y-z to z-x-y convention"""

    def b(num):
        """forces magnitude to be 1 or less"""
        if abs(num) > 1.0:
            return math.copysign(1.0, num)
        else:
            return num

    s1 = math.sin(ang[0])
    s2 = math.sin(ang[1])
    s3 = math.sin(ang[2])
    c1 = math.cos(ang[0])
    c3 = math.cos(ang[2])

    pitch = math.asin(b(c1 * c3 * s2 - s1 * s3))
    cp = math.cos(pitch)
    # just in case
    if cp == 0:
        cp = 0.000001

    yaw = math.asin(b((c1 * s3 + c3 * s1 * s2) / cp))  # flipped
    # Fix for getting the quadrants right
    if c3 < 0 and yaw > 0:
        yaw = math.pi - yaw
    elif c3 < 0 and yaw < 0:
        yaw = -math.pi - yaw
```

A.2 Functions for calculating LLP Constants

Function for calculating $Q^{q_a q}_{q_p q_e}$, $\delta^{q_a}_{q_e}$, and $S^{q_e}_{q_e}$.

```
    def generate_quad_integrals(self, q_a, q, q_p, q_r):
        def quad(i, j, m, n):
            li, lj, lm, ln = (Legendre([0] * k + [1]) for k in (i, j, m,
            ↪  n))
            L = (li * lj * lm * ln).integ()
            # the desired result is (L(1) - L(-1)) / 2 (as this is for
            ↪  non-shifted Legendre)
            #  but since L(1) == -L(-1), this is just L(1)
            return L(1)
        qs = [q, q_a, q_p, q_r]
        w = np.zeros((qs[0], qs[1], qs[2], qs[3]))
        for i in range(qs[0]):
            for j in range(i, qs[1]):
                for m in range(j, qs[2]):
                    for n in range(m, qs[3]):
                        # skip indices guranteed to be 0
                        if (i+j+m-n >= 0) and ((i+j+m-n) % 2 == 0):
                            v = quad(i, j, m, n)
                            for index in itertools.permutations([i, j, m,
                            ↪  n]):
                                # TODO catch these properly with something
                                ↪  like filter()
                                try:
                                    w[index] = v
                                except Exception as e:
                                    pass
        return w

    def generate_delta_identity(self, q_a, q_p):
        i = min(q_a, q_p)
        d = np.zeros((q_a, q_p))
        for ii in range(0, i):
            d[ii, ii] = 1
        return d

    def generate_scaling_diagonal(self, q):
        S = np.zeros((q, q))
        for i in range(0, q):
            S[i, i] = 2*i + 1
        return S
```

Appendix B

Additional Figures

B.1 Training Data Control Signal

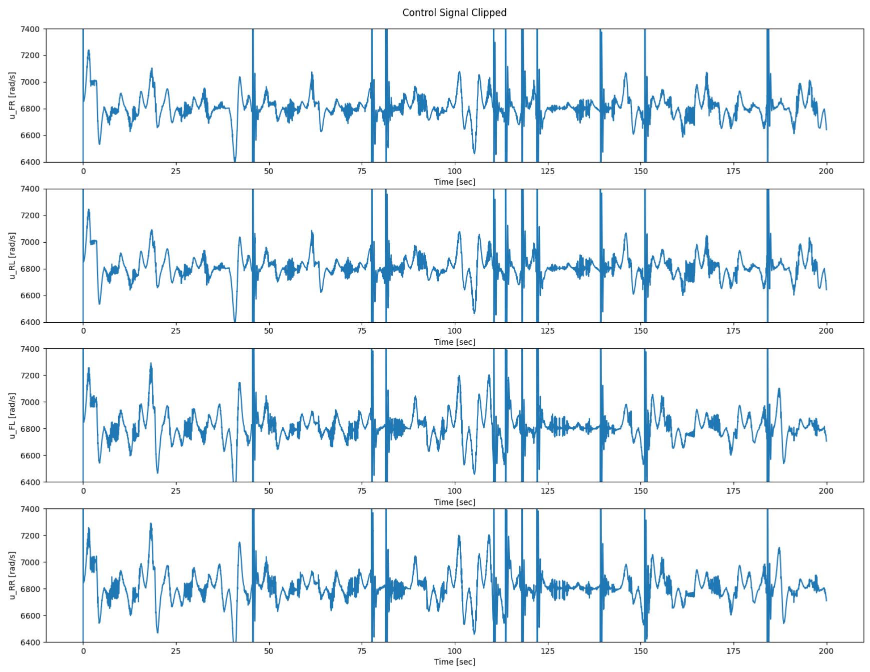


Figure B.1: A portion of the clipped control signal from the PD controller, before normalizing

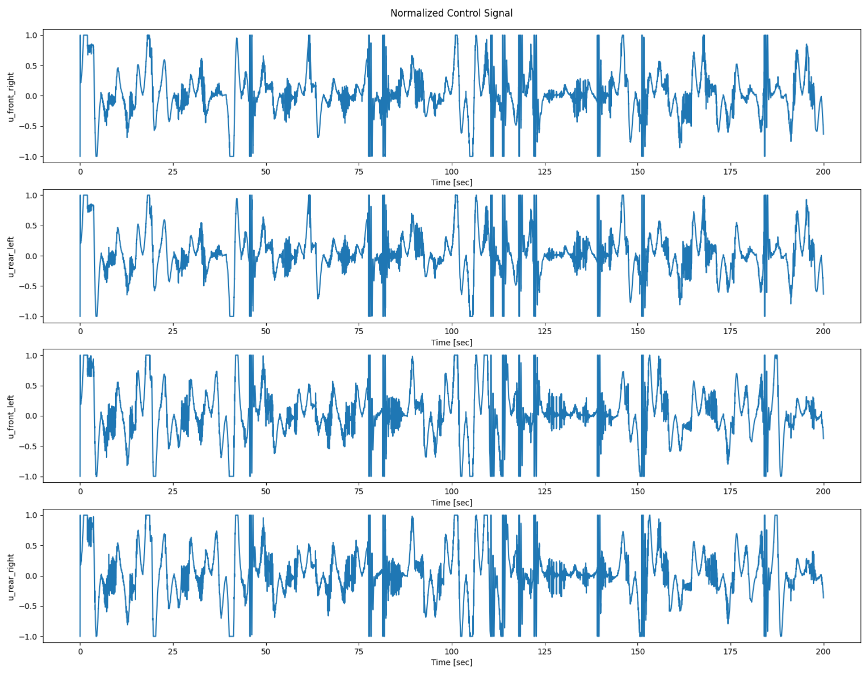


Figure B.2: A portion of the normalized PD control signal used for training data.

B.2 Training Data State

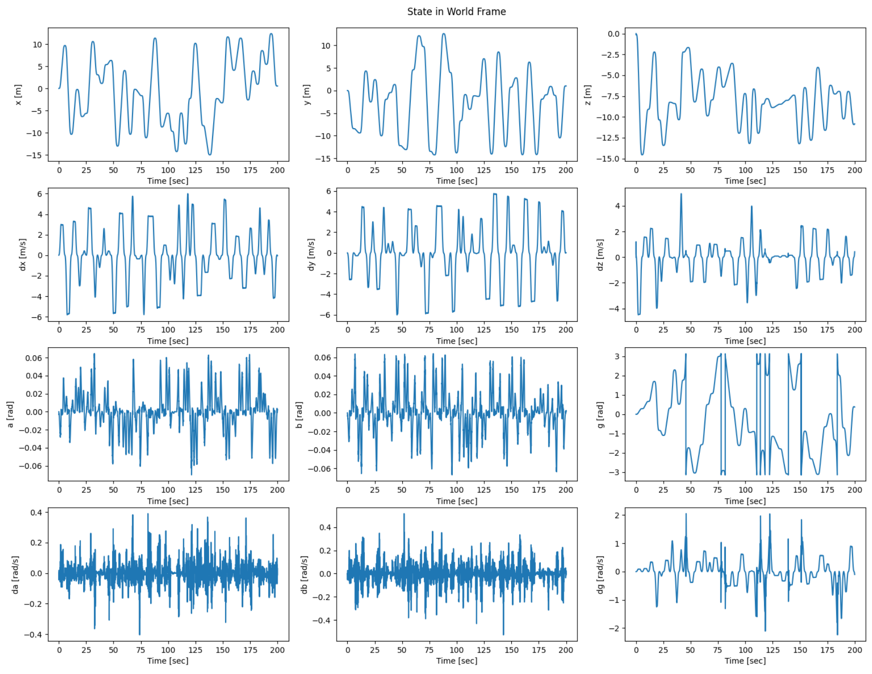


Figure B.3: The quadrotor state data before normalization. From top to bottom, the top row shows the 3 dimensions of position in meters, the second row the velocities in m/s, the third orientations in radians, and the fourth angular velocities in rad/sec.

B.3 Training Data Reference

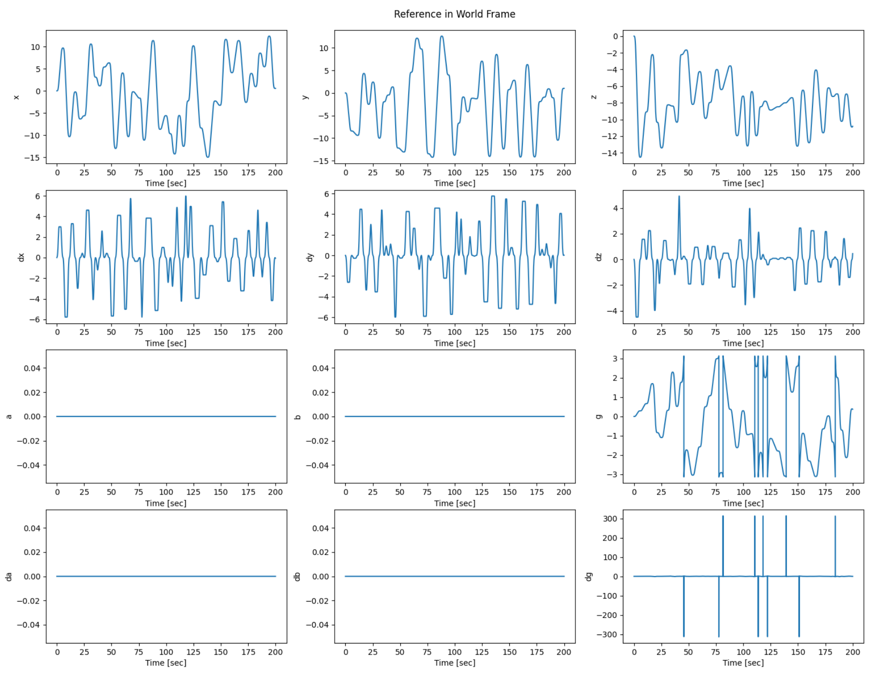


Figure B.4: The reference data before clipping and normalization. From top to bottom, the top row shows the 3 dimensions of position in meters, the second row the velocities in m/s, the third orientations in radians, and the fourth angular velocities in rad/sec.