Beyond Linear Horizons: NN and RL Algorithms Eclipse Traditional Models in High-Fidelity Simulation and Control for Omnidirectional Robot

Student Information

Names: Yuran Dai, Tianlang Zhang, Zekai Ma

School: Keystone Academy of Beijing

Province/State: Beijing

Country: China

Advisor Information

Advisor Name: Zhongyao Sun

Institution: Keystone Academy of Beijing

Submitted in September 2025

Beyond Linear Horizons: NN and RL Algorithms Eclipse Traditional Models in High-Fidelity Simulation and Control for Omnidirectional Robot

Yuran Dai, Tianlang Zhang, Zekai Ma

September 2025

Abstract

Omnidirectional swerve drivetrains in the FIRST Robotics Competition (FRC) demand high-fidelity simulations and optimized motion profiles for peak performance. This study compares linear models—DC motor feedforward and state-space representations—against nonlinear NN architectures, including FNN, LSTM and TCN models for motor motion prediction. It also evaluates traditional trapezoidal motion profiles versus RL-based profiles using PPO.

We prototyped a functional FRC swerve module, calibrated and deployed it to collect high-resolution motion data. The cleaned voltage and velocity data from each driving motor were accurately and efficiently predicted by traditional linear models, yet the mechanics of steering motors, which were more prone to friction and nonlinear forces, were not well captured by linear models. Nonetheless, our experimented nonlinear models showed great improvement from the linear benchmark.

In motion profiling, a PPO-RL framework with curriculum learning for a straight path yielded policies that reduced average completion time of the robot compared to that of a traditional trapezoidal profile.

This data-driven data collection, model training, and RL deployment might offer FRC teams accessible tools for optimization, with extensions to broader wheeled robotics applications.

Keywords

Swerve Drivetrain; Omnidirectional Robotics; FIRST Robotics Competition (FRC); System Identification (SysID); Neural Networks (NN); Feedforward Neural Networks (FNN); Long Short-Term Memory (LSTM); Temporal Convolutional Network (TCN); Reinforcement Learning (RL); Proximal Policy Optimization (PPO); Trapezoidal Motion Profile; Curriculum Learning; Data-Driven Simulation; Nonlinear Dynamics; Inverse Kinematics; DC Motor Feedforward; State-Space Models; Robot Chassis Modeling; Motion Profiling; Autonomous Navigation

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 The Big Picture

Wheeled robots are arguably the most mature, widespread, and rapidly evolving category of rigid-body robots. Peak performance in autonomous wheeled robots is based on two fundamental pillars: an accurate simulator and an optimal control strategy. Although simulating and controlling wheeled robots is often simpler than working with other types of robots such as drones and humanoids, the current treatment of complex variants of wheeled robots lacks a deeper and more comprehensive analysis.

When it comes to building simulations for wheeled robots, two main families of models are often applied: the linear approach via state space models and the nonlinear method of using neural networks (NN) and other deep learning (DL) models.

Similarly, for planning motion policies or state-action pair series of wheeled robots, typically for autonomous navigation tasks, there is a popular linear approach of using a trapezoidal motion profile. In recent years, its nonlinear counterpart, the machine learning (ML)-based approach of using reinforcement learning (RL) strategies, has also made groundbreaking achievements.

Both linear and nonlinear schools of thought have been quite successful in simulating and controlling many sorts of wheeled robot, where the latter nonlinear method is gaining popularity due to its versatility and proven breakthroughs in many fields.

1.2 Background Research & Motivation

State-space models, in the more general field of robotics refer to determining a set of matrices of parameters that reflect how a system evolves over time. Typically, an equation relates the instantaneous rate of change of the states to a linear combination of the previous state and the current input of the system. In other words, state-space models are oftentimes linear approximations of a system, but the real system could very well behave nonlinearly.

For the class of wheeled robots, wheels are often actuated by permanent magnet DC motors; there is a particular feedforward model in the form of an ordinary differential equation that relates the applied voltage of a motor, its velocity, and acceleration with a trio of coefficients to be determined through linear regression.

Neural network models for simulating robots are less mature than linear state-space models, as feedforward models do well in many simple systems and are easier to train and deploy compared to emerging NN models. However, for more complex systems, it is a good idea to experiment with these models as they can capture the nonlinear dynamics of the system, which intuitively can be more accurate than linear approximation modules like the DC motor feedforward. In the context of wheeled robots, such nonlinearities include static friction, motor saturation, backlash, and cross-system interactions.

Thus, by their architectural properties, NN models have great potential to improve simulation accuracy compared to the linear models that are currently being used for wheeled robots.

When designing motion profiles to be executed over preplanned paths, the linear trapezoidal motion profile sets fundamental limitations to the dynamics of these robots by setting a strict accelerate – cruise – decelerate isosceles trapezoidal shape to the velocity curve. Although this approach is classic, simple, and widely used, it might be suboptimal due to its inherent limitations to the linear assumption.

In contrast, reinforcement learning (RL) strategies can learn more versatile, nonlinear motion profiles based on customized value functions and are not constrained to a specific shape, so there is a lot of opportunity for RL-based training to discover more optimal policies.

Therefore, like the task of simulating wheeled robots, an RL-driven approach might perform better than classical linear profiles for the task of finding optimal policies as well.

1.3 Contributions & Significance

To summarize the big picture, our research consists of two main components: 1) comparing the simulation accuracy of linear state-space models with nonlinear neural network models and 2) comparing the performance of policies planned by the linear trapezoidal profile versus nonlinear profiles from reinforcement learning.

We performed this duo of investigations in the context of a challenging and popular real-life problem that involves a complex member of the wheeled robot's family: an omnidirectional swerve drivetrain in the FIRST Robotics Competition (FRC). In the FRC community, high schoolers from all over the world build and program their robots to compete in human-sized arenas to earn points through completing certain tasks such as shooting targets and placing objects. The swerve happens to be the most complex wheeled robot built by the FRC community, making it a great mechanism to test with.

Many software tools and libraries have been created for modeling/simulation and motion profiling, but most of these simulators are based on linear assumptions such as the state space model and are calculated from a few low-dimension equations, which do not promise great adaptability and fidelity. Furthermore, in the autonomous period of the competition, many teams still use a built-in trapezoidal policy to plan the relative velocities of the robot chassis, which could be oversimplified or suboptimal, as previously discussed.

By completing this research, our contribution is twofold. The first contribution is designing a data-driven, ML-based procedure for the FRC community to create high-fidelity simulators for every team's robot, given a sufficient and varied dataset. The road map here is to test three kinds of NN architecture (FNN, LSTM, and TCN) with two sets of input and output vectors (motor velocity predictions or chassis velocity predictions) and compare them against error metrics of the linear model benchmarks of two scopes (motor relative or chassis relative), and finally to interpret the results. Our second contribution is to experiment with an RL-based training routine so that all FRC teams can plan more versatile and varied motion profiles to execute during the autonomous period of the competition. The trained policies are converted to voltages applied to each motor based on the models we created in the first section and finally deployed to the physical world to validate and compare speed and accuracy metrics.

The significance of this research is that this is the only formally documented attempt to test the performance of multiple NN models on simulating a complex system with so many inputs and outputs. It is also one of the few attempts to decompose a robot in two ways (i.e. motors or chassis-relative velocities). In terms of RL, we experimented with the new method of training a robot with minimal degrees of freedom, and finally, using motor prediction models, reverse the trained policy to input voltages for physical deployment.

2 Hardware & Data Preparation

2.1 Hardware Preparation

The swerve drivetrain is the most complex and expensive chassis used in the FRC arena. It is an omnidirectional robot, just like the more well-known Mecannum chassis, meaning that the swerve can translate in any direction and rotate simultaneously and independently of translation. To enable this unique feature, each of the four wheels is independently driven and steered, as you can see in the robot mechanical drawing. This means that there are eight motors controlling the drivetrain (four wheels, each wheel is steered by a motor and driven by a motor). Then, through forward kinematics of aggregating the velocity vector of each wheel, it will yield the chassis-relative translation and rotation velocities.

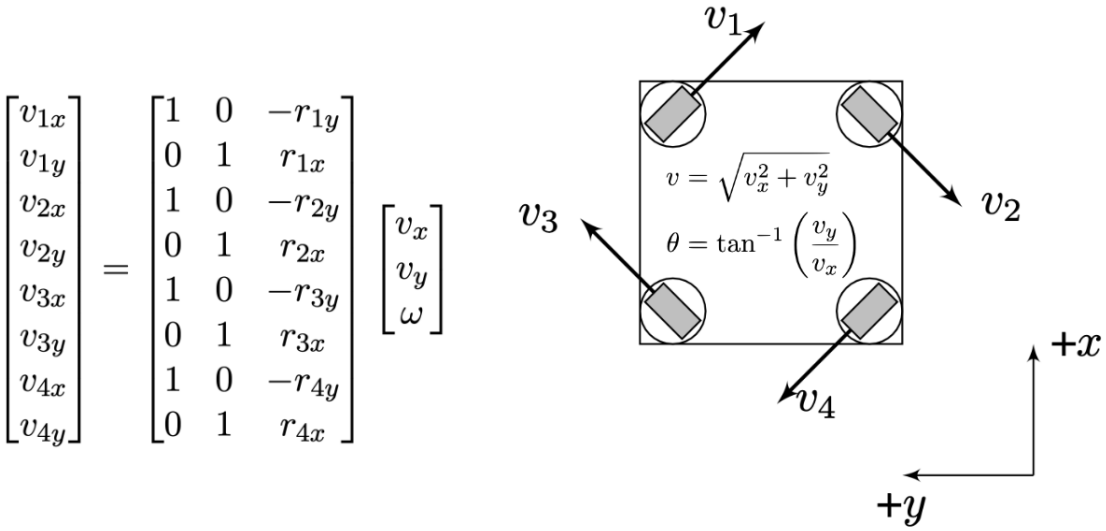

Figure 1: Inverse kinematics formula for swerve drivetrain

In Figure 1, the propagation of the inverse kinematics from the relative speeds of the chassis (v_x, v_y and ω) to the velocity setpoints for each of the four modules is shown. Note that v_{ix} and v_{iy} represent the x and y decompositions of each wheel's velocity vector, and r_{iy}, r_{ix} refers to the relative position from each wheel to the center of the swerve.

It is essential that we have access to a real, functional, swerve drive to collect physical data and to deploy trained policies. As members of a 2-year-old FRC team, we have all the necessary tools to build a swerve chassis in a short period of time. We used an online computer-aided design tool, OnShape, to design the mechanism and wiring; then we assembled all the components that came from a manufacturer. The mechanical drawing of our robot is shown in Figure 2, where you can see the four motor modules in the corners. Note that we used a square frame for simpler setup in the robot code.

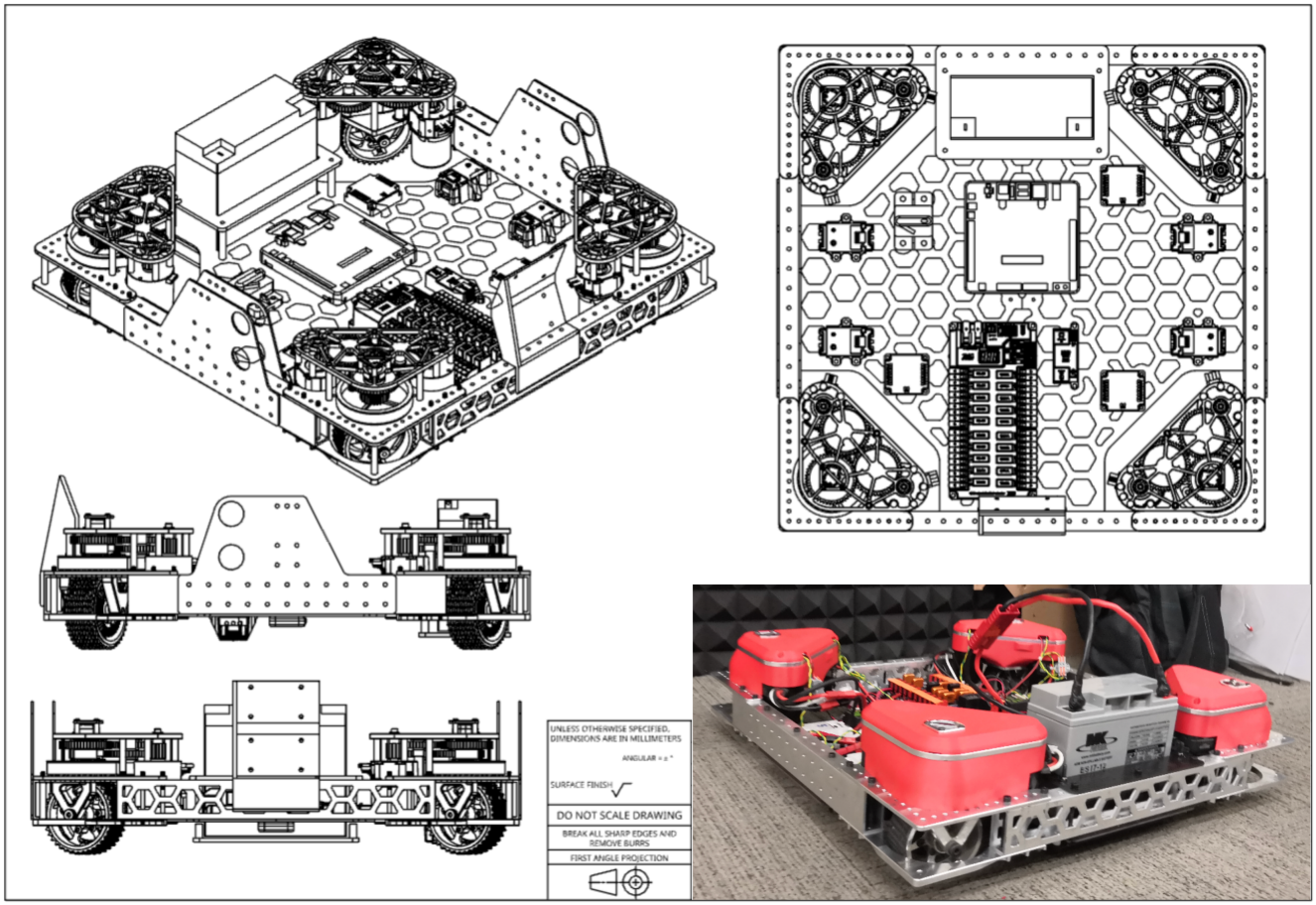

Figure 2: Mechanical drawing of the swerve

After assembly, we used the template code from our previous seasons to quickly activate and fine-tune the robot. Fine-tuning the feedback controller is especially important because we need to make sure that the robot we collect data from can navigate smoothly, just like what we would expect from a competition robot. After fine-tuning the parameters, they will not be changed throughout the data collection process, as they would be encrypted in the voltage-velocity response function.

2.2 Data Collection, Curation, Cleaning

In the data collection stage, our team has been operating the swerve drive in a fixed arena to ensure that the robot-relative velocities recorded can represent the expected distribution as in an actual FRC game. Ensuring the diversity of velocities is vital to our experiment, as the models we train need to be able to make accurate predictions for any possible maneuver that the swerve can perform.

During data collection, we made sure to lubricate the gears frequently, charge the battery to a fixed energy level baseline, and double check the quality of the wiring. This is to ensure that the decay caused by running the machine is minimized. During the summer, we only changed our venue once, after which we recollected all data points to ensure uniform friction properties from the floor over all samples ran.

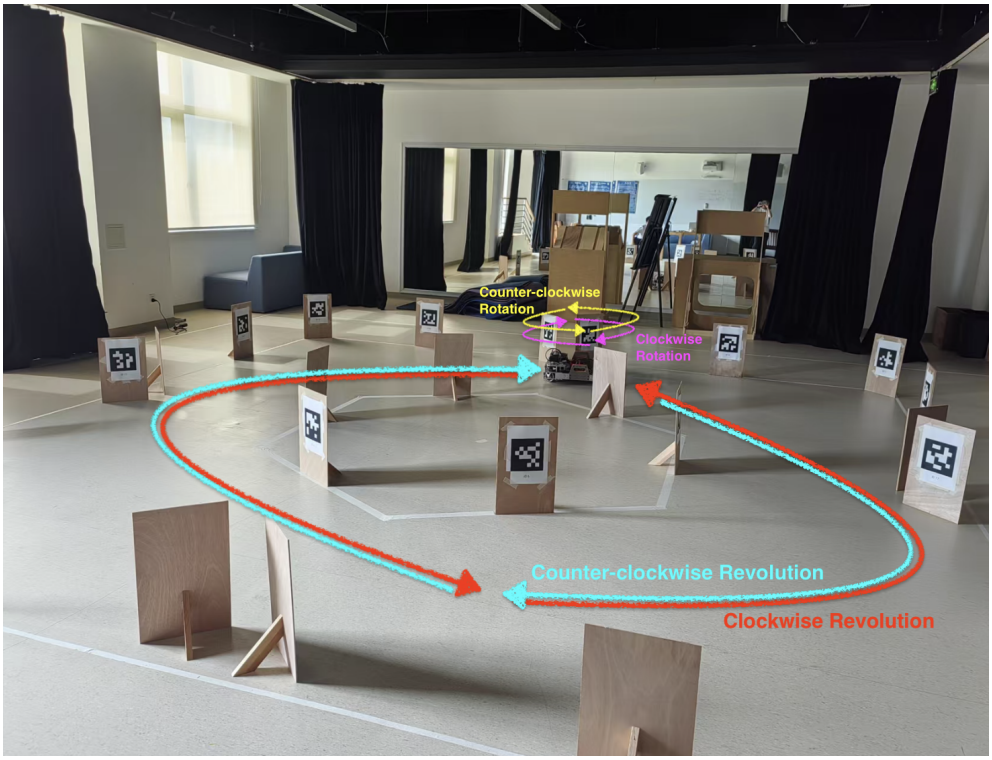

Figure 3: Layout of data collection arena

As marked in Figure 3, the design of our arena consists of a cyclic path. This guarantees that the robot will translate in every direction as it is driven along the path. Along the four sides of the outer square edge, the driver can also focus on speeding or accelerating at a larger magnitude. By combining different velocities of the robot's self-rotation with its revolution along the circular path, we can also collect a lot of data where the swerve is translating and rotating simultaneously. You can easily see in the distribution plots of the robot-relative linear and angular velocities that our routine design worked well, producing three large range, balanced Gaussian shapes.

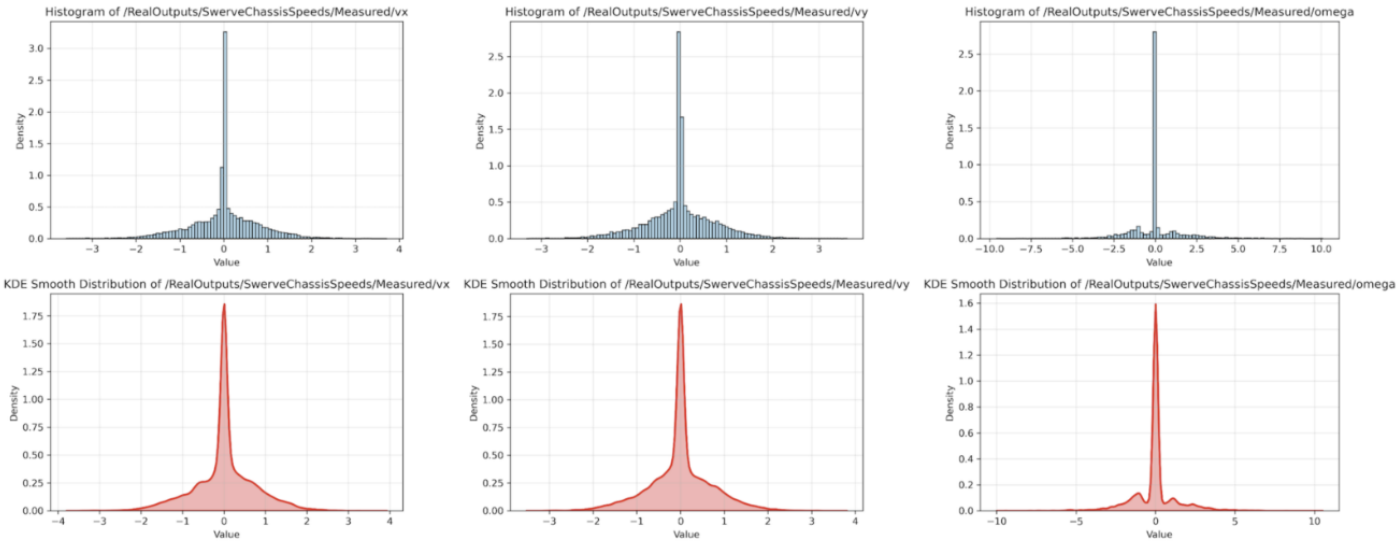

Figure 4: Histograms & KDE smooth distributions of v_x, v_y and ω

Notice how the x and y components of the linear velocity of the robot, measured from high-precision magnetic encoders in each motor, reach 2 meters per second, meaning that many of the magnitudes of the full linear velocity can reach 3 meters per second. The angular velocity ω, measured from a low

drift IMU (inertia measurement unit), can reach up to 10 radians per second, which is more than 1.5 rotations per second, shows the wide range of effective data collected from our procedure.

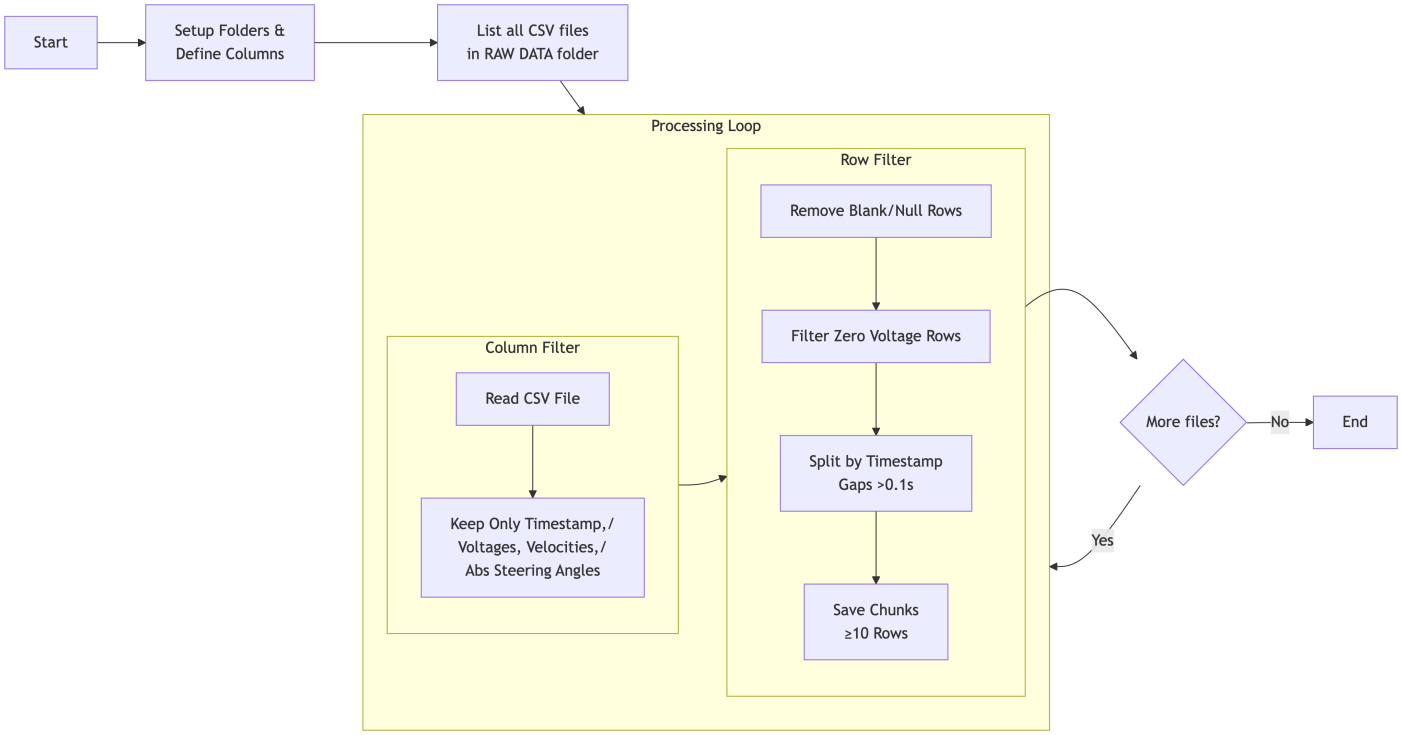

Figure 5: Flowchart of the data cleaning process

Next, to ensure that the data that enter the training stage are effective, we must filter out the timestamps where the driver was taking a break, meaning that no voltage is applied to any of the motors. We also had to make sure that the csv files we saved were not too short, because there could be an impulse of numbers logged that had a duration of less than 0.1 seconds, and it would not be very credible to put into training. Furthermore, there is a particular set of columns/features that we are focused on learning their relationships, thus we had to apply a filter to the entries we save as well. Considering all these factors, we created a data cleaner that is well summarized in Figure 3.

After cleaning, we were left with a folder of csv files that contains the raw metadata of the swerve (including timestamps, velocities, voltages, steering angles) that represents the dynamics while the drivetrain is active. In aggregate, it is equivalent to over 40 minutes of non-stop running the robot, discretized into 0.02 second increments. In other words, we have prepared approximately

$$40 \times \frac{60}{0.02} - n = 120000 - n \tag{1}$$

Data points for training, where n is the length of the input sequence, which can vary from 1 to 50 depending on the model choice. This massive set of data is the first step in developing reliable models.

3 Swerve Drivetrain Modeling

3.1 Two Modeling Perspectives

The first section of our research focuses on the experiment with new models for the swerve drive, which can then be used to develop high-fidelity simulations. Simulations are very important in robotics today as they would be more efficient, accessible, and economical, which can be used to train human drivers and run RL algorithms.

Two types of linear models are widely used by the FRC community, depending on the perspective of the system. The most common type of model is the DC motor feedforward, where three parameters are calculated for each individual motor through a linear regression process typically known as system identification. The parameters relate the applied voltage of each motor with the velocity and acceleration through an ordinary differential equation, and then the velocities can be grouped through forward kinematics to simulate more complex systems.

$$V = K_s \cdot sgn(\dot{d}) + K_v \cdot \dot{d} + K_a \cdot \ddot{d} \tag{2}$$

where V is the applied voltage, d is the displacement (position) of the motor, $\dot{d}$ is its velocity and $\ddot{d}$ is its acceleration. Just as mentioned earlier, this is in the form of an ODE.

Another type of model is the state space model for the full swerve chassis where the state variables are robot-relative, where the key is to take account of the kinematics properties before the regression, and then the matrices of parameters trained can be directly applied in simulation with no extra forward kinematics required. However, note that no state-space model has been proposed by any FRC teams, and most teams are stuck with system identification of DC motor feedforward. It is possible that some have tried and failed as the swerve is highly nonlinear, but we can still experiment with it to create a benchmark. The state-space approach is implemented in Section 3.3.

We will be creating benchmarks based on these two linear models, analyzing the results from the benchmarks, and then testing out some models based on neural networks. In the next couple of sections, we will start by creating DC motor feedforward models, where we predict the velocities and acceleration from the previous state and applied voltage.

3.2 DC Feedforward Benchmark (Motor Modeling)

We first start with the perspective of modeling individual motors. In Algorithm 1, the pseudocode of our system identification benchmark model is displayed. The main idea is to determine the coefficients K_s, K_v, and K_a through a linear regression process to complete the ODE as in Equation 2. There are a few key steps that we added to formalize the process that are worth noting.

Inputs & Outputs: With a simple transformation applied to Equation 2, it can be shown that when used for prediction, the output of this existing linear model is the next velocity of the motor v_{t+1}, while the inputs are the time difference, velocity, acceleration, and applied voltage of the previous states. Equation 3 describes the general form of the prediction, where ω_i are the parameters.

$$v_{t+1} = w_0 \cdot \text{sign}(v_t) + w_1 \cdot V_t + w_2 \cdot v_t + w_3 \cdot a_t + b \tag{3}$$

Data Split: To prevent overfitting and perform a more generalizable evaluation, 20% of the data points are hidden from the model and saved as a validation set. Only 80% are used to train the model. This will remain conventional for all the models in this research.

Feature Standardization: To ensure the numerical stability of the linear regression, a standardization was performed on the input vectors:

$$\mathbf{X}_{\text{scaled}} = \left[\mathbf{x}_{\text{sign}}, \frac{\mathbf{x}_V - \mu_V}{\sigma_V}, \frac{\mathbf{x}_v - \mu_v}{\sigma_v}, \frac{\mathbf{x}_a - \mu_a}{\sigma_a}\right] \tag{4}$$

where μ_i and σ_i are the mean and standard deviation of the i-th feature vector. Note that in the final period of evaluation, the predictions made are scaled back to the raw values.

Ridge Regression: In essence, we are solving a least-squares problem, which can be described by Equation 5. The λ term is used for regularization, which prevents overfitting, making the least squares problem a ridge regression problem.

$$\mathbf{w}_{\text{ridge}}^{(m)} = \left((\mathbf{X}_{\text{train}}^{(m)})^T \mathbf{X}_{\text{train}}^{(m)} + \lambda \mathbf{I}\right)^{-1} (\mathbf{X}_{\text{train}}^{(m)})^T \mathbf{y}_{\text{train}}^{(m)} \tag{5}$$

Algorithm 1 Linear Regression System Identification (train_sysid.py)

1: *// Data processing* Load and preprocess CSV files from folder
2: **for** each file f in directory **do**
3: Extract timestamp t and normalize
4: Separate drive motors (4) and steering motors (4) data
5: **for** each motor $m \in \{0, 1, \ldots, 7\}$ **do**
6: Create dataset with features and target:
7: $\mathbf{X}_m = [\text{sign}(v), V, v, a]$
8: $\mathbf{y}_m = v_{t+1}$ *// Next timestep velocity*
9: **end for**
10: **end for**
11: *// Model operation* For each motor $m \in \{0, 1, \ldots, 7\}$:
12: Standardize features: $\mathbf{X}_{\text{scaled}}^{(m)} = \frac{\mathbf{X}^{(m)} - \mu_{\mathbf{X}}^{(m)}}{\sigma_{\mathbf{X}}^{(m)}}$
13: Split data: $\mathcal{D}^{(m)} = \mathcal{D}_{\text{train}}^{(m)} \cup \mathcal{D}_{\text{val}}^{(m)}$
14: Solve linear regression:
15: $argmin_{\mathbf{w}^{(m)}} \|\mathbf{X}_{\text{train}}^{(m)} \mathbf{w}^{(m)} - \mathbf{y}_{\text{train}}^{(m)}\|_2^2 + \lambda \|\mathbf{w}^{(m)}\|_2^2$
16: Extract physical parameters:
17: $k_s^{(m)} = w_0^{(m)}$ *// Static friction coefficient*
18: $k_v^{(m)} = w_2^{(m)} / \sigma_v^{(m)}$ *// Velocity coefficient*
19: $k_a^{(m)} = w_3^{(m)} / \sigma_a^{(m)}$ *// Acceleration coefficient*
20: *// Evaluation* For each motor m:
21: Calculate MSE, MAE & R^2 metrics
22: Generate evaluation plots
23: **Return**: Parameters $\{k_s^{(m)}, k_v^{(m)}, k_a^{(m)}\}$ for each motor, evaluation metrics

The key step in the system identification process is in line 15, where we try to minimize the squared error of the prediction compared to the ground truth of the dataset. The second term on line 15 is a regularization term to prevent overfitting and ensure generalizability. A more visual representation is shown in the flow chart of Figure 6, where the process is grouped in sections.

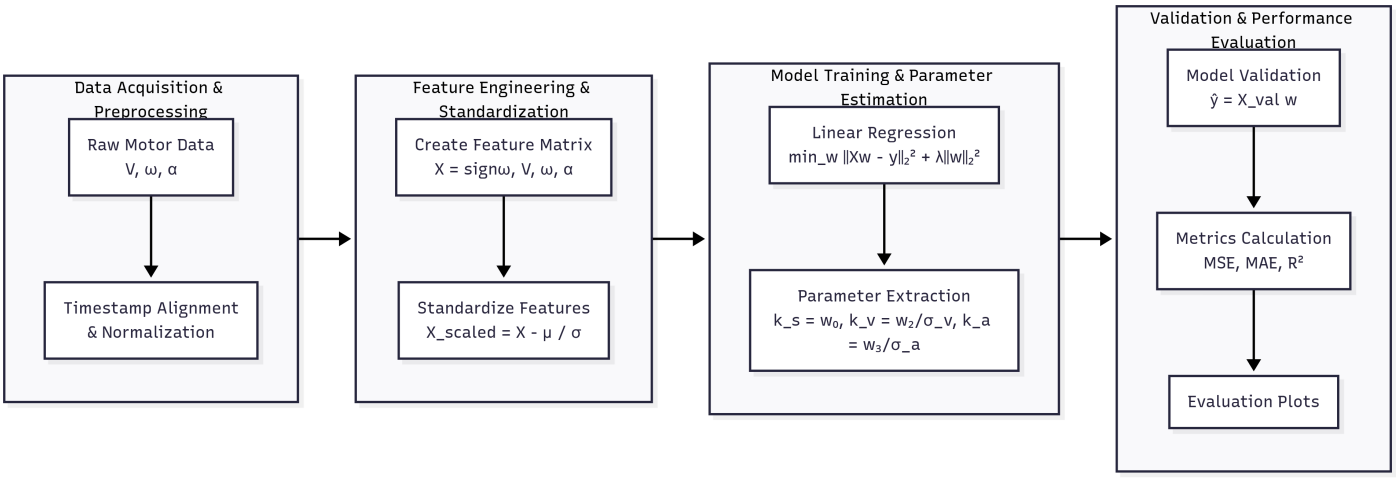

Figure 6: Flowchart of system identification for DC motor

After running the system identification, the evaluation metrics for each motor are calculated independently and summarized in the table below.

Motor ID	**MSE**	**MAE**	**R^2**
Drive Left-Front	4.703774	0.788845	0.990820
Drive Right-Front	3.355239	0.748699	0.993368
Drive Left-Back	2.870923	0.746534	0.994500
Drive Right-Back	2.796669	0.735960	0.994414
Steer Left-Front	1.283855	0.553053	0.676740
Steer Right-Front	0.898873	0.465431	0.786768
Steer Left-Back	0.947053	0.475456	0.774799
Steer Right-Back	0.785239	0.448109	0.817506

Table 1: Linear FF model evaluation for drive vs. steer motors

Observing the MSE (Mean Squared Error) term, we observe that the driving motors have a much larger MSE error (around 3 to 5), while the MSE values of most of the steering motors are lower than 1. We can see a similar trend in the MAE (Mean Absolute Error) column as well.

However, we cannot draw the conclusion that the linear feedforward model captures the dynamics of steering motors more accurately than the drive motors. Instead, this is because the velocity range of the driving motors is much wider than that of the steering motors. The steering motors typically never have to spin a full revolution in a few seconds because they are only responsible for controlling the direction of translation, while the driving motors are almost always spinning at a high RPM to enable motion. Hence, MSE and MAE tend to be higher for datasets with larger magnitudes and higher variance, which are not the most suitable metrics to compare model accuracies against other datasets.

Instead, if we move on the R-squared column of the table, we can clearly see that the data from the driving motors fit better than the steering motors. The coefficient of determination for all driving motors is greater than 0.99, while most steering motors have a coefficient lower than 0.8. To visualize the predictor's performance, evaluation plots have been created for each motor predictor as in Figure 7. To account for the length of this paper, only the results of the left-front corner driving (index 0) versus steering (index 4) motors will be displayed for comparison, which will remain as a convention for all the

following models on motor velocity prediction.

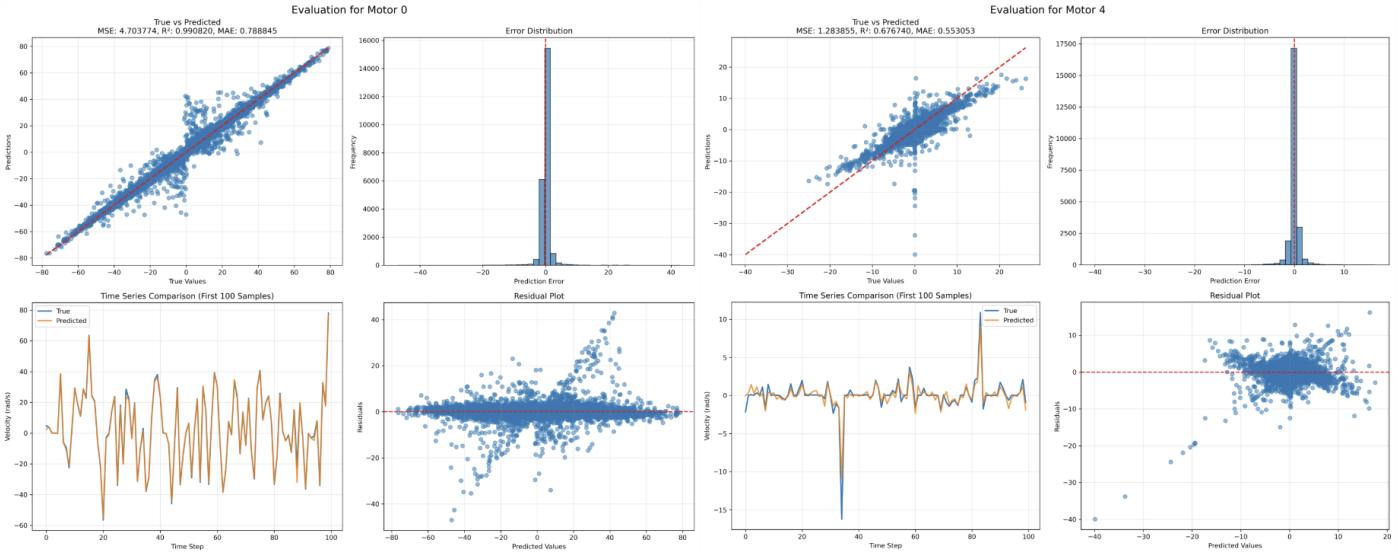

Figure 7: Diagnostic plots of drive vs. steering motors predictions of linear FF

As we can see in the true vs. predicted plot and the residual plot for both driving and steering motors, the trend of the plots seems reasonable and is close to the optimal red dotted line. However, the range of velocities for driving motors ranges from –80 to 80, while steering velocities appear to be mainly in the -20 to 20 range, justifying our claim that driving motors have greater velocity magnitudes compared to steering motors. We can also observe in the error distribution plot that the range of steering prediction errors is about half that of the driving motors, leading to smaller MSE and MAE metrics. However, as we move on to the time series comparison graph to visualize the predictor's performance, the driving predictor almost completely coincides with the ground truth, even in sharp turns and swift changes; contrarily, though with smaller error metrics, the steering predictor makes a lot of visible errors, especially for lower ground truth velocities, which justify the lower coefficient R squared.

In general, considering the magnitude of the data, the DC motor feedforward appears to be more suitable for driving motors than for steering motors. The high value of R-squares complements the suitability of the model and persuades us that the linear interpretation is already quite suitable to drive predictions. On the other hand, steering predictors have more room for improvement considering their lower coefficient of determination.

Our interpretation of this result is that driving motors are dominated by linear inertia force, whereas friction constitutes more of the force experienced by steering motors. Since the robot's constant mass is being pushed around by driving motors, the force applied is proportional and therefore linear to the mass and acceleration; on the other hand, the load of the steering motor is low inertia with small mass, as all it spins are the gearboxes. Hence, with little inertia to tackle, the main force of the steering motor is to overcome friction in gearboxes, belts, and bearings, which is famously nonlinear. Therefore, steering motors become our main target for modeling with nonlinear methods.

3.3 Innovations of NN-based Approaches (Motor Modeling)

From the diagnostics of the benchmark, our neural networks experiment mainly focused on improving the MSE and R-squared metrics of the models to predict steering motor velocities. Driving motor predictors have also been trained by the following algorithms, but as per our hypothesis, they

were underperforming the existing DC motor feedforward model, as the physics is very linear already. Therefore, diagnostic plots for driving motors will not be included in this paper but will be uploaded as supplementary information.

The first NN model experimented is the feedforward neural network (FNN). The motivation for trying out this model first is that it only takes the state vector (voltages, velocities, steering angles) of the previous timestamp into account, but as a NN model it still effectively adds nonlinearity into play, making the FNN one of the simplest models to start with. The pseudocode of our designed architecture is shown in algorithm 2, but prior to that let us decompose some of the key structures and process.

Inputs & Outputs: After training this model, the predicted values are once again the velocities of the steering motor, while the inputs passed into the network include the time difference and previous steering velocities of both the steering and driving motors (to account for the effects of the system).

Feature Standardization: Just as in the linear FF model, the features are standardized in the same way here with its mean and standard deviation.

NN architecture: Our FNN has four layers to accommodate the size of input features. For the first three hidden layers, their processing follows the structure:

$$\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k)} \tag{6}$$

$$\mathbf{h}^{(k)} = \max(0, \mathbf{z}^{(k)}) \tag{7}$$

$$\mathbf{h}^{(k)}_{\text{drop}} = \text{Dropout}(\mathbf{h}^{(k)}, 0.3) \tag{8}$$

where k ranges from 1 to 3. Equation 6 shows the linear step of the layer by performing linear combinations. Equation 7 adds nonlinearity with a ReLU activation step. Equation 8 uses a random dropout to prevent co-adaption and overfitting besides the use of regularization. Note that the fourth layer, being the last layer before output, does not have dropout.

Adam Optimizer: The model is trained using the Adam (Adaptive Moment Estimation) optimizer with learning rate $\alpha = 0.001$. For each parameter θ at timestep t:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L} \tag{9}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L})^2 \tag{10}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t} \tag{11}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \tag{12}$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \tag{13}$$

Where $\mathcal{L}$ stands for the loss function, and ∇_θ is its gradient. This module was included to help adjust the learning rates for better convergence.

Bias Correction: A post-processing step applies linear bias correction to each motor's predictions.

$$\hat{y}_{\text{corrected}} = \hat{y} - (a\hat{y} + b) \tag{14}$$

where a and b are estimated by linear regression between predictions and errors:

$$a, b = \arg\min_{a,b} \sum_{i=1}^{N} \left[(y_i - \hat{y}_i) - (a\hat{y}_i + b)\right]^2 \tag{15}$$

Notice that this bias correction was not implemented in the benchmark linear FF model for steering motor because rotation of a linear model does not add nonlinearity, thus would have been trivial.

Algorithm 2 Feedforward Neural Network (train_fnn.py)

// *Data processing* Load and preprocess CSV files
for each file f in directory **do**
  for each consecutive timestep $(t-1, t)$ **do**
    Create input: $\mathbf{X}^{(i)} = [\mathbf{x}_{t-1}, \Delta t]$ // *Previous state & delta time*
    Create targets:
      $\mathbf{y}^{(i)}_{\text{drive}} = \mathbf{x}_t[5:9]$ // *Driving motor velocities*
      $\mathbf{y}^{(i)}_{\text{steer}} = \mathbf{x}_t[17:21]$ // *Steering motor velocities*
  end for
end for
Standardize features & targets: $\mathbf{X}_{\text{scaled}} = \frac{\mathbf{X}-\mu_{\mathbf{X}}}{\sigma_{\mathbf{X}}}$, $\mathbf{y}_{\text{scaled}} = \frac{\mathbf{y}-\mu_{\mathbf{y}}}{\sigma_{\mathbf{y}}}$
// *Model operation* Build FNN architecture:
Input layer: $\mathbf{h}^{(0)} = \mathbf{X}$
Layer 1 (256 units):
  $\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}$
  $\mathbf{h}^{(1)} = \max(0, \mathbf{z}^{(1)})$
  $\mathbf{h}^{(1)}_{\text{drop}} = \text{Dropout}(\mathbf{h}^{(1)}, 0.3)$
Layer 2 (128 units):
  $\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}^{(1)}_{\text{drop}} + \mathbf{b}^{(2)}$
  $\mathbf{h}^{(2)} = \max(0, \mathbf{z}^{(2)})$
  $\mathbf{h}^{(2)}_{\text{drop}} = \text{Dropout}(\mathbf{h}^{(2)}, 0.3)$
Layer 3 (64 units):
  $\mathbf{z}^{(3)} = \mathbf{W}^{(3)}\mathbf{h}^{(2)}_{\text{drop}} + \mathbf{b}^{(3)}$
  $\mathbf{h}^{(3)} = \max(0, \mathbf{z}^{(3)})$
  $\mathbf{h}^{(3)}_{\text{drop}} = \text{Dropout}(\mathbf{h}^{(3)}, 0.2)$
Layer 4 (32 units):
  $\mathbf{z}^{(4)} = \mathbf{W}^{(4)}\mathbf{h}^{(3)}_{\text{drop}} + \mathbf{b}^{(4)}$
  $\mathbf{h}^{(4)} = \max(0, \mathbf{z}^{(4)})$
Output layer: $\hat{\mathbf{y}} = \mathbf{W}^{(5)}\mathbf{h}^{(4)} + \mathbf{b}^{(5)}$
// *Parameter update* Train with Adam optimizer ($\alpha = 0.001$):
1_{st} moment estimate: $\mathbf{m}_t = \beta_1\mathbf{m}_{t-1} + (1-\beta_1)\nabla_\theta\mathcal{L}$
2_{nd} moment estimate: $\mathbf{v}_t = \beta_2\mathbf{v}_{t-1} + (1-\beta_2)(\nabla_\theta\mathcal{L})^2$
Correction: $\hat{\mathbf{m}}_t = \mathbf{m}_t/(1-\beta_1^t)$, $\hat{\mathbf{v}}_t = \mathbf{v}_t/(1-\beta_2^t)$
Update: $\theta_{t+1} = \theta_t - \alpha \cdot \hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$
Apply L2 regularization: $\mathcal{L}_{\text{reg}} = \lambda \sum_{l=1}^{5} \|\mathbf{W}^{(l)}\|_F^2$
Total loss: $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \mathcal{L}_{\text{reg}}$
Apply bias correction: $\hat{y}_{\text{corrected}} = \hat{y} - (a\hat{y} + b)$
// *Evaluation* Evaluate model performance:
Calculate MSE, MAE & R^2 metrics
Return: Trained models, scalers, evaluation metrics

The FNN is designed to have four layers to fit the size of input features and is facilitated by an Adam Optimizer to learn at an optimal step size. The inputs are pre-processed with a standardization scaler, and the outputs/predictions are corrected by a biasing term to preserve linear components. For the full logic, refer to the pseudocode or the flowchart in Figure 8.

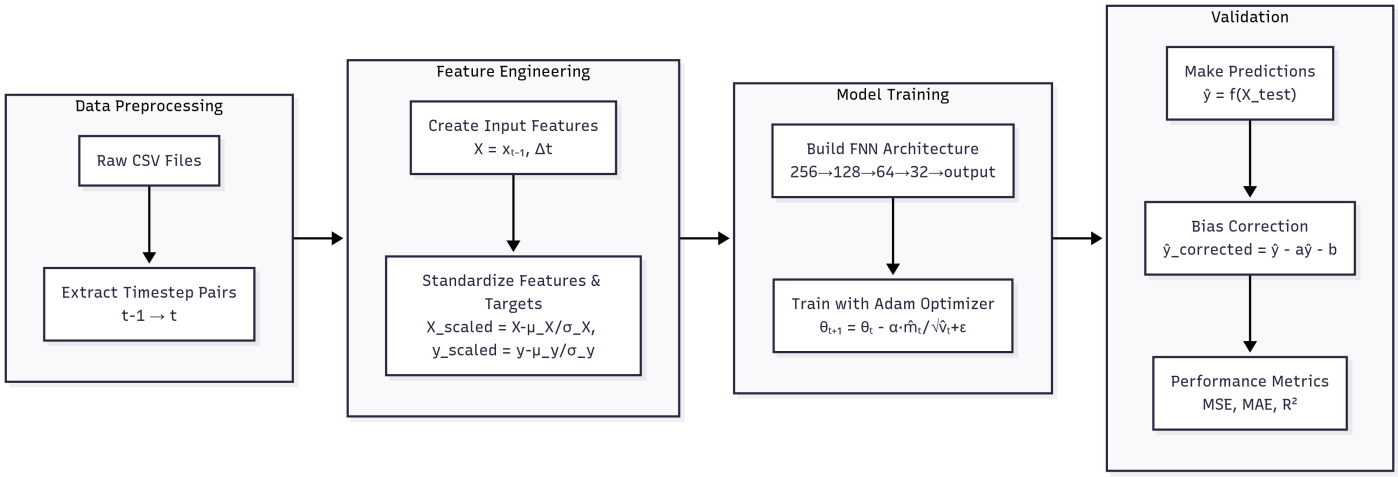

Figure 8: Flowchart of FNN training for steering motor velocity prediction

After implementing the FNN training routine, the evaluation metrics for each motor are calculated independently and summarized in the table below. Note that these metrics are calculated from a validation set that covers 20% of the entire data set, meaning that this data set has not been deployed in training and is used to prevent overfitting. These terms are the values after biasing has been applied.

Steering Motor ID	**MSE**		**MAE**		**R^2**	
	Value	Change%	Value	Change%	Value	Change%
Left Front	0.890373	(-30.65%)	0.508175	(-8.11%)	0.813989	(+20.28%)
Right Front	0.983997	(+9.47%)	0.501645	(+7.78%)	0.807215	(+2.6%)
Left Back	1.042931	(+10.12%)	0.502239	(+5.63%)	0.788101	(+1.72%)
Right Back	1.171317	(+49.17%)	0.556029	(+24.08%)	0.765384	(-6.38%)

Table 2: FFN model evaluation and improvement percentage

For the MSE and MAE metrics, we hope it is as small as possible, while we want the R-squared value to be as large as possible. From the improvement percentages, we see that the R2 metrics have improved greatly for three out of the four motors, showcasing a generally better fit of model. However, the MSE and MAE metrics only improved the left front steering motor. Another observation is that the span/range of each column of evaluation metrics is more constrained and focused, compared to linear DC motor feedforward's metrics, which has more variance.

To get a visual idea of how well the predictor worked, refer to the diagnostic plots we extracted from the left front steering motor. The blue and green scatter plots represent the predictions before and after biasing correction, where the evaluation metrics table comes from the bias-corrected models. We can see that compared to the linear DC feedforward model, our FNN's actual vs predicted plot is a lot thinner and has much less bias. If we look at the time series comparison, the predictor is also fitting quite well to impulses of larger magnitudes, and the overall shape coincides much better.

To get an idea of the impact of the bias correction term, we took the principal axis of the blue residual plot (denoted by the blue dotted line) and shifted it to fit the horizontal dotted line to account for systematic errors. However, for this model, as we can see in the error distribution comparison graph, bias correction does not have much influence, suggesting that the dynamics is highly nonlinear, as our correction term is performed linearly.

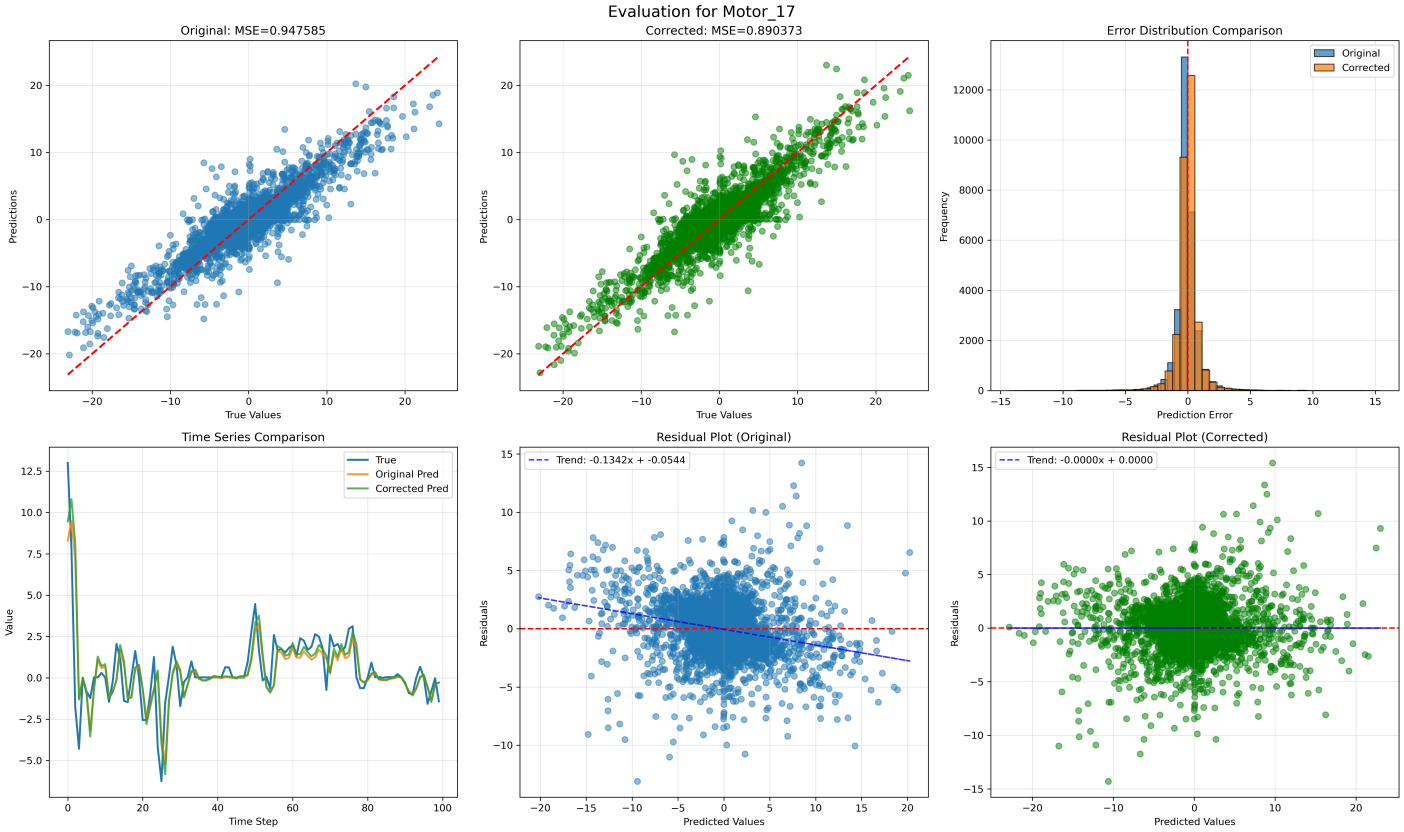

Figure 9: Diagnostic plots for steering motor using FNN

Looking back at the summary table, the greatest success of our FNN model is the improved coefficient of determination: we have pushed the R2 value from the 0.68-0.82 range to a 0.77-0.81 range. This is a sign that the FNN managed to learn some of the nonlinear relationships that could not be learned by system identification. The downside of this model is that the MSE and MAE errors have increased as a whole, which implies that the different steering modules may have varying physical properties such as friction and gearing that would make the predictions less uniform. Overall, the range of all these metrics have become more consistent after applying FNN, and this can be explained by the fact that it is a single unified model with one set of weights that learns for all four motors simultaneously and making it easier to learn average dynamics while less sensitive to unique quirks, as we mentioned earlier.

This result suggests that NN models might learn the nonlinearities well, but it might be insufficient to only provide one timestamp of input states because unique quirks can occur in higher-order states, which we suspect can be better captured in longer dependencies.

This brings us to the next experimental model - Long Short-Term Memory (LSTM). The LSTM is a type of recurrent neural network (RNN) that is especially popular in natural language processing. However, its most important property is that LSTMs can recognize patterns in sequences of data when used properly. Our main motivation for using this model is that from experience with FNNs, the behavior of the steering motor could be a function of recent history rather than an instantaneous function of the previous state, so the LSTM is a great model to test with.

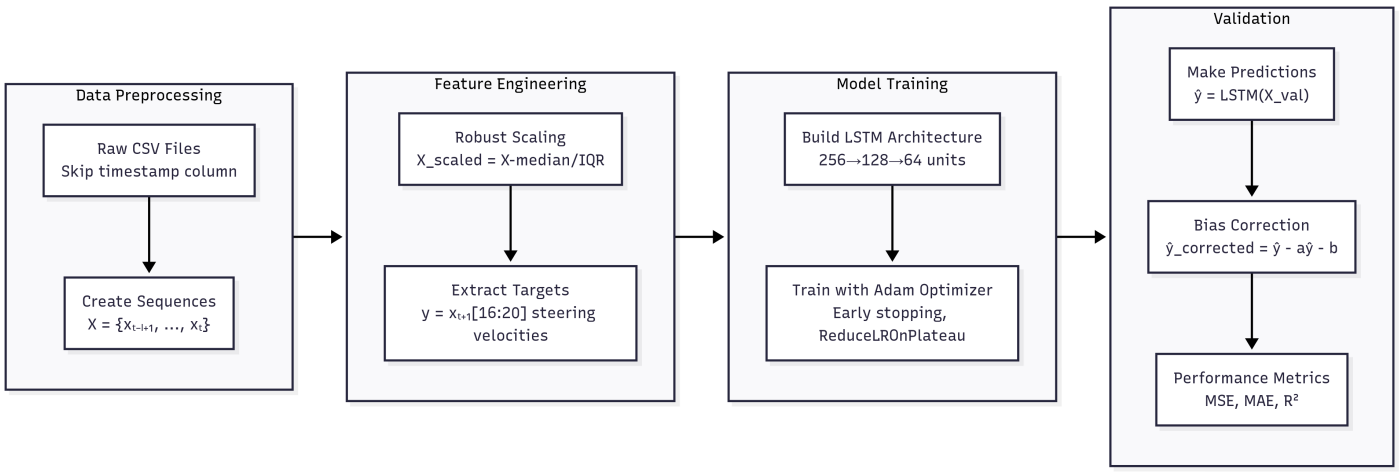

Figure 10: Flowchart of LSTM training for steering motor velocity prediction

Our testing procedure is provided as a pseudocode and is also visualized by a flowchart. Since the main hyperparameter of the LSTM is the sequence length of history states we would plug into our network, we experimented with memory sizes varying from 5 to 50 timestamps, which is equivalent to utilizing 0.1 to 1 seconds into the history. Note that the timestamps are not included as inputs for this RNN, as LSTMs assume uniform timestamp differences, and our time increments are all reasonably close to 0.02 seconds. Once again, some key modules are decomposed here:

Inputs & Outputs: The output variable has not changed, it is still the steering motor velocity of the next timestamp. The input variable has changed quite a lot: it includes a sequence of velocities and voltages of all 8 motors in a past window of timestamps. Furthermore, the timestamp differences are not included this time because of the assumption of uniformity.

Robust Scaling: Considering the increase in input size, robust scaling is applied to handle outliers in real-world sensor data, where IQR stands for inter-quartile range:

$$\mathbf{X}_{\text{scaled}} = \frac{\mathbf{X} - \text{median}(\mathbf{X})}{\text{IQR}(\mathbf{X})} \tag{16}$$

This approach is more resistant to outliers than the previously used standard z-score normalization.

LSTM Architecture: For each timestep t in the sequence,

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \text{(Forget gate)} \tag{17}$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \text{(Input gate)} \tag{18}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \text{(Output gate)} \tag{19}$$
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \quad \text{(Candidate cell state)} \tag{20}$$
$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \quad \text{(Cell state update)} \tag{21}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \quad \text{(Hidden state update)} \tag{22}$$

Where the $\odot$ is the Hadamard product of matrices (element-wise multiplication).

Batch Normalization: To stabilize and accelerate deep neural network training by reducing internal covariate shift, batch normalization was applied.

$$\mathbf{h}_{\text{bn},t} = \gamma \frac{\mathbf{h}_t - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{23}$$

This block stabilizes the vanishing/exploding gradient issue even more, which is added on top of the simplest form of LSTM considering the depth of this network.

Algorithm 3 LSTM-based Prediction (LSTM.py)

// Data processing For sequence length $L \in \{5, 10, 20, 30, 50\}$:
for each file in directory **do**
    Load and preprocess, skip timestamp column
    for each valid sequence position i **do**
        Create sequence: $\mathbf{X}^{(i)} = \{\mathbf{x}_{t-L+1}^{(i)}, \ldots, \mathbf{x}_t^{(i)}\}$
        Set target: $\mathbf{y}^{(i)} = \mathbf{x}_{t+1}^{(i)}[16:20]$ *// Steering velocities*
    end for
end for
Apply RobustScaler:
$\mathbf{X}_{\text{scaled}} = \frac{\mathbf{X} - \text{median}(\mathbf{X})}{\text{IQR}(\mathbf{X})}$ *// Robust standardization*
// Model operation Build LSTM architecture:
for each timestep t in sequence **do**
    LSTM Layer 1 (256 units):
    $\mathbf{f}_t^{(1)} = \sigma(\mathbf{W}_f^{(1)}[\mathbf{h}_{t-1}^{(1)}, \mathbf{x}_t] + \mathbf{b}_f^{(1)})$ *// Forget gate*
    $\mathbf{i}_t^{(1)} = \sigma(\mathbf{W}_i^{(1)}[\mathbf{h}_{t-1}^{(1)}, \mathbf{x}_t] + \mathbf{b}_i^{(1)})$ *// Input gate*
    $\mathbf{o}_t^{(1)} = \sigma(\mathbf{W}_o^{(1)}[\mathbf{h}_{t-1}^{(1)}, \mathbf{x}_t] + \mathbf{b}_o^{(1)})$ *// Output gate*
    $\tilde{\mathbf{C}}_t^{(1)} = \tanh(\mathbf{W}_C^{(1)}[\mathbf{h}_{t-1}^{(1)}, \mathbf{x}_t] + \mathbf{b}_C^{(1)})$ *// Candidate cell state*
    $\mathbf{C}_t^{(1)} = \mathbf{f}_t^{(1)} \odot \mathbf{C}_{t-1}^{(1)} + \mathbf{i}_t^{(1)} \odot \tilde{\mathbf{C}}_t^{(1)}$ *// Cell state update*
    $\mathbf{h}_t^{(1)} = \mathbf{o}_t^{(1)} \odot \tanh(\mathbf{C}_t^{(1)})$ *// Hidden state update*
    Batch normalization:
    $\mathbf{h}_{\text{bn},t}^{(1)} = \gamma^{(1)} \frac{\mathbf{h}_t^{(1)} - \mu^{(1)}}{\sqrt{\sigma^{2\,(1)} + \epsilon}} + \beta^{(1)}$
    Dropout: $\mathbf{h}_{\text{drop},t}^{(1)} = \text{Dropout}(\mathbf{h}_{\text{bn},t}^{(1)}, 0.3)$
end for
Repeat for LSTM layers 2 (128 units) and 3 (64 units)
Final output: $\hat{\mathbf{y}} = \mathbf{W}_{\text{out}}\mathbf{h}_{\text{drop}}^{(3)} + \mathbf{b}_{\text{out}}$
// Parameter update Train with Adam optimizer ($\alpha = 0.003$):
1_{st} moment estimate: $\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L}$
2_{nd} moment estimate: $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L})^2$
Correction: $\hat{\mathbf{m}}_t = \mathbf{m}_t/(1 - \beta_1^t)$, $\hat{\mathbf{v}}_t = \mathbf{v}_t/(1 - \beta_2^t)$
Update: $\theta_{t+1} = \theta_t - \alpha \cdot \hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$
Apply bias correction: $\hat{y}_{\text{corrected}} = \hat{y} - (a\hat{y} + b)$
// Evaluation Evaluate model performance:
Calculate MSE, MAE & R^2 metrics
Generate evaluation plots for original and bias-corrected predictions
Return: Trained model, evaluation metrics for each sequence length

After experimentation, we focused on the memory size that yielded the least MSE error, which turns out to be when we utilized 10 timestamps, or 0.2 seconds from history. The effect of increasing sequence lengths from 5 to 50 is shown in Figure 11.

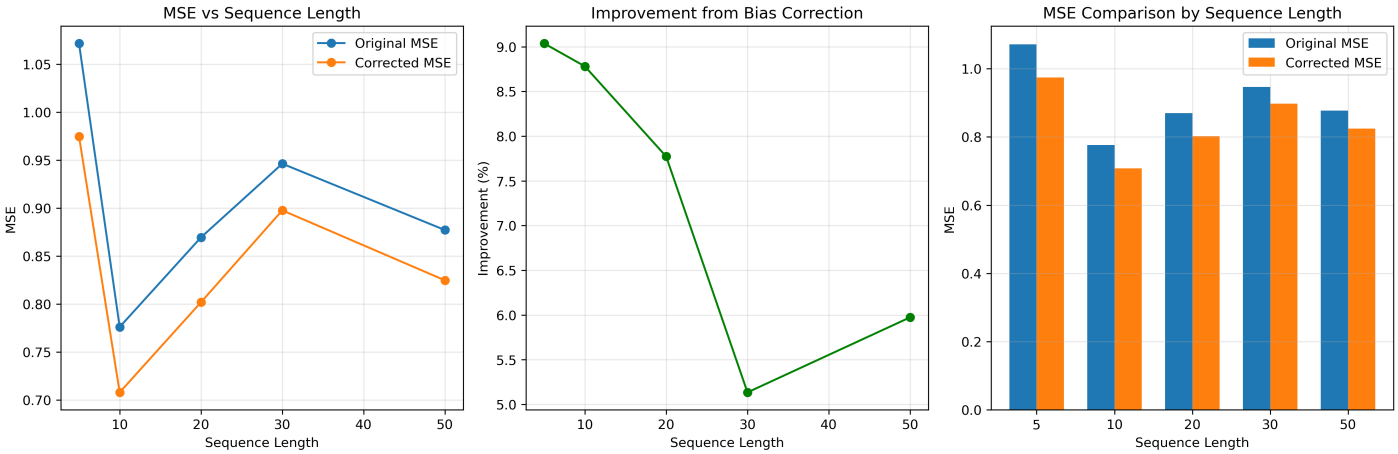

Figure 11: Comparison of LSTM sequence lengths for minimum error

In the improvement % from biasing graph, we can see that the linear correction yielded an effective error reduction ranging from 5% to 9%, but if we refer to the MSE comparison by the sequence length bar graph, it appears that a sequence length of 10 would yield the least MSE error, even without bias correction. Our hypothesis of why we reached this optimum at a sequence length of 10, but no longer or shorter, is that this could be a typical representation of the bias-variance tradeoff theory.

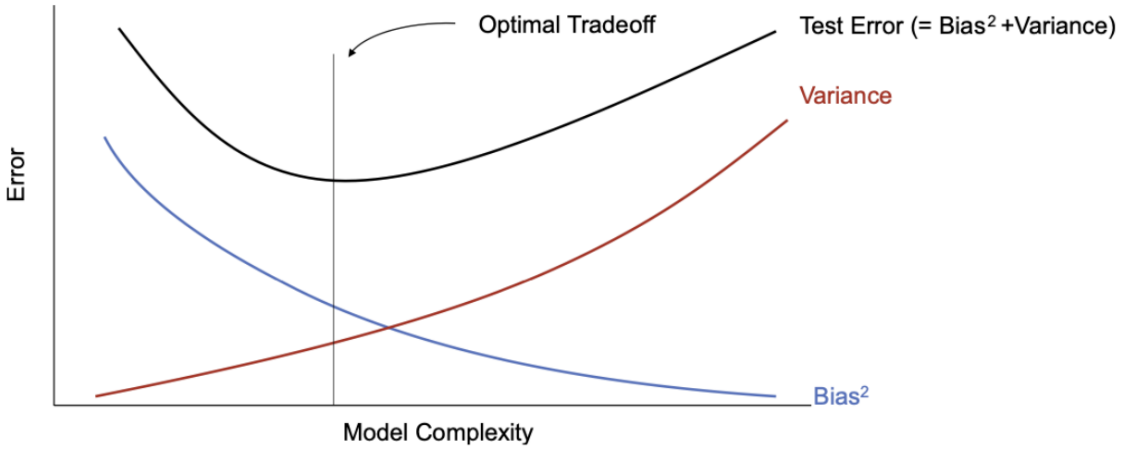

Figure 12: Bias-variance tradeoff phenomenon in ML

The idea is that at a short length of sample sequence such as 5 timestamps, the bias is large due to insufficient context. If we compare the corrected MSE for length 5 (0.98 according to the graphs) to the previous FNN results that only used one timestamp (MSE was 1.0), we can see that both MSEs are quite large and that they may be underfitting. On the other hand, if we apply a longer length of sample size from 20 – 50 or even larger, the problem is that variance grows large, thus the model will be exposed to a great amount of irrelevant noise. Note that our graphs are not strictly increasing, but the overall trend does abide by the bias-variance tradeoff explanation. This tradeoff property will be used a few more times in the models that appear later in the paper.

Now, let us focus on the evaluation metrics from the LSTM model that inputs 10 history timestamps. These metrics appear after bias correction.

Steering Motor ID	MSE		MAE		R^2	
	Value	Change%	Value	Change%	Value	Change%
Left Front	0.895551	(-30.24%)	0.487431	(-11.87%)	0.725142	(+7.15%)
Right Front	0.470273	(-47.68%)	0.375528	(-19.32%)	0.853086	(+8.43%)
Left Back	0.677826	(-28.44%)	0.412646	(-13.21%)	0.792767	(+2.32%)
Right Back	0.788263	(+0.39%)	0.453323	(+1.16%)	0.801816	(-1.92%)

Table 3: LSTM model evaluation and improvement percentage

We observe a tremendous reduction in both the MSE and MAE errors of most of the steering predictors, and the only predictor (right back motor) that performed worse in terms of error metrics only decayed by tiny amounts less than 2%. We also observed a boost in most of the R-squared values of the motor, suggesting a better fit for the dynamics.

Let us now take a closer look at the graphical diagnostics of the left front steering module. We can see that the original versus corrected models do not differ much from each other, especially in the blue residual plot, where the slope magnitude is about 0.07. Observing the time series comparison, we recognized that regardless of applying a correction term, the arithmetic average of the predicted values seems to coincide with the mean of the ground truth, therefore justifying that a linear transformation would not do much. However, a common feature of the predictors is that they would oftentimes undershoot when the actual values are relatively low, and overshoot when the ground truth is larger. Fortunately, the difference is not excessive and is likely because sharp changes in speed are still hard to learn with longer dependencies because there is an instantaneous component to it.

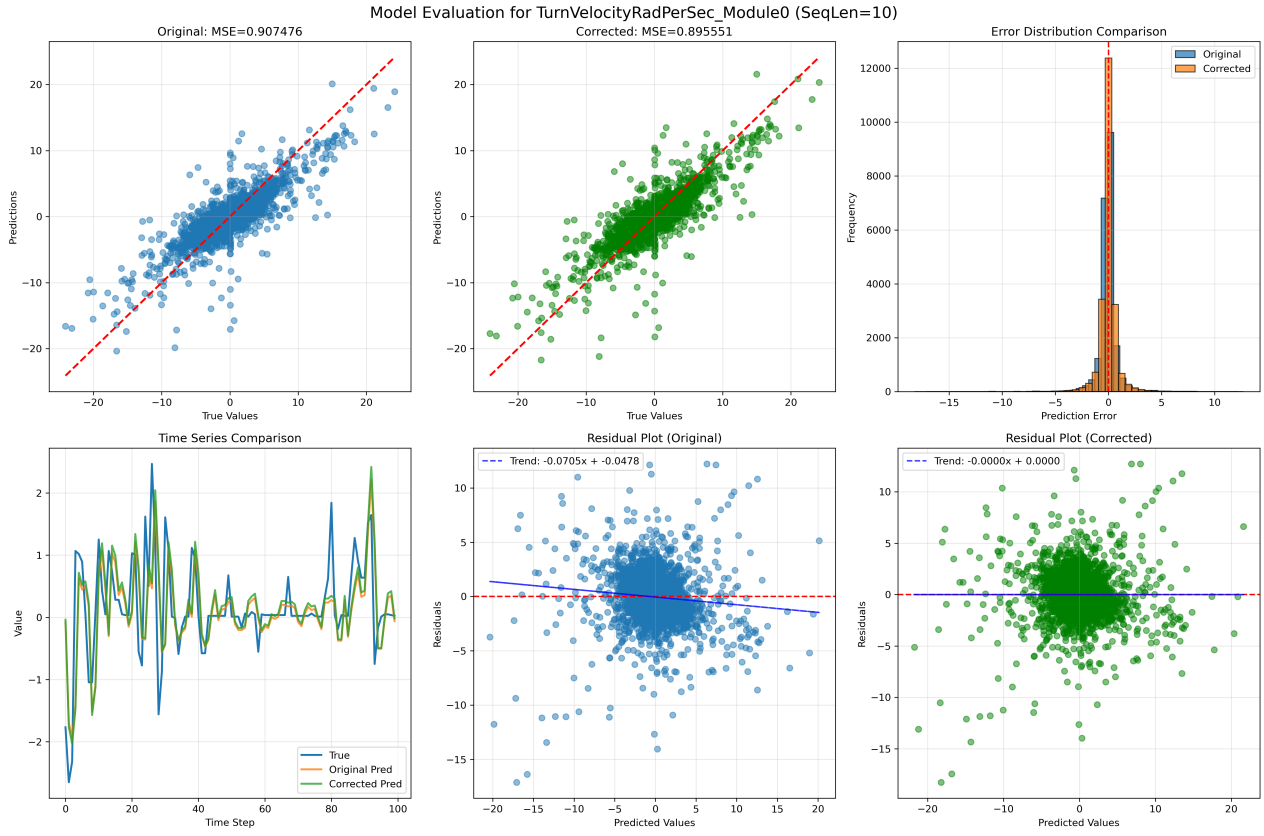

Figure 13: Diagnostic plots for steering motor using the LSTM model (seqLen=10)

To summarize the overall performance shown in the previous table, we have experienced a massive

reduction in MSE and MAE metrics, as well as a significant increase in explanatory power. These numbers suggest more accurate predictions for the steering motors and a better fundamental fit for data across all modules compared to the benchmark linear DC motor feedforward. Another key observation is that, compared to the FNN, our LSTM has a higher variability between different motors. This makes sense because LSTMs learn the unique temporal footsteps of each motor, unlike the FNN, which takes a more general approach and produces similar performance metrics.

From our analysis of the results, it seems like RNNs did better than the non-recurrent FNN, which is already outperforming the traditional system identification approach. It is a satisfactory point to stop, but to take a step further, we thought that it might be helpful to capture multi-scale dynamics across the history timestamps. While the LSTM handles the vanishing gradient problem well so that inputs that are deeper into the past can be handled with similar weight as closer timestamps, intuitively it is also to weight the contexts at different scales with stability. The fix for this is a temporal convolution network (TCN).

The TCN is also a type of RNN and is traditionally used for natural language processing and audio processing, just like the LSTM. In addition to capturing multiscale dynamics, the TCN is trained via parallelism, so the learning process is also much faster, and it would be more suitable for FRC teams that do not have access to high-performance GPUs.

Inputs & Outputs: Same as the LSTM model - historic states & inputs, no timestamps.

Dilated Convolutions: To capture temporal patterns at multiple timescales, dialated convolutions were applied to identify high-frequency and seasonal patterns.

$$\mathbf{h}_{conv}^{(d)} = \sum_{k=0}^{K-1} \mathbf{W} k^{(d)} \mathbf{h} t - d \cdot k^{(d-1)} + \mathbf{b}^{(d)} \tag{24}$$

This ensures that the network can model long-range dependencies with a stable gradient.

Residual Connections: To address degradation issues in this deep network, residual connections are included in the architecture.

$$\mathbf{h}^{(d)} = \max\left(0, \mathbf{h}_{drop2}^{(d)} + \mathbf{h}_{skip}^{(d)}\right) \tag{25}$$

This function uses ReLU activation to introduce nonlinearity, which also ensures that gradients can propagate effectively through many layers, enabling the training of very deep temporal models.

Global Average Pooling: For temporal summarization, we create a fixed-length representation here.

$$\mathbf{z} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{h}_t^{(32)} \tag{26}$$

The key advantage of adding this pooling layer is that it reduces parameter count.

So far, the key features of our design has been decomposed, and the full algorithm is shown below. Note that standardization, Adam optimizers and bias correction are performed as before.

Algorithm 4 Temporal Convolutional Network (TCN.py)

// *Data processing* For sequence length $L \in \{5, 10, 20, 30, 50\}$:
for each file in directory **do**
    Load and preprocess, skip timestamp column
    for each valid sequence position i **do**
        Create sequence: $\mathbf{X}^{(i)} = \{\mathbf{x}_{t-L+1}^{(i)}, \ldots, \mathbf{x}_t^{(i)}\}$
        Set target: $\mathbf{y}^{(i)} = \mathbf{x}_{t+1}^{(i)}[16:20]$ // *Steering velocities*
    end for
end for
Standardize features & targets: $\mathbf{X}_{\text{scaled}} = \frac{\mathbf{X}-\mu_{\mathbf{X}}}{\sigma_{\mathbf{X}}}$, $\mathbf{y}_{\text{scaled}} = \frac{\mathbf{y}-\mu_{\mathbf{y}}}{\sigma_{\mathbf{y}}}$
// *Model operation* Build TCN architecture:
$\mathbf{h}^{(0)} = \gamma^{(0)} \frac{\mathbf{X}-\mu^{(0)}}{\sqrt{\sigma^{2(0)}+\epsilon}} + \beta^{(0)}$ // *Input normalization (BatchNorm)*
$\mathbf{h}^{(1)} = \text{Conv1D}(\mathbf{h}^{(0)}) = \sum_{k=0}^{K-1} \mathbf{W}_k^{(1)} \mathbf{h}_{t-k}^{(0)} + \mathbf{b}^{(1)}$ // *Initial convolution*
for dilation rate $d \in \{1, 2, 4, 8, 16, 32\}$ **do**
    $\mathbf{h}_{\text{conv1}}^{(d)} = \sum_{k=0}^{K-1} \mathbf{W}_k^{(d,1)} \mathbf{h}_{t-d\cdot k}^{(d-1)} + \mathbf{b}^{(d,1)}$ // *First dilated convolution*
    $\mathbf{h}_{\text{bn1}}^{(d)} = \gamma^{(d,1)} \frac{\mathbf{h}_{\text{conv1}}^{(d)}-\mu^{(d,1)}}{\sqrt{\sigma^{2(d,1)}+\epsilon}} + \beta^{(d,1)}$ // *Batch normalization*
    $\mathbf{h}_{\text{act1}}^{(d)} = \max(0, \mathbf{h}_{\text{bn1}}^{(d)})$ // *ReLU activation*
    $\mathbf{h}_{\text{drop1}}^{(d)} = \text{Dropout}(\mathbf{h}_{\text{act1}}^{(d)}, 0.2)$ // *Dropout regularization*
    $\mathbf{h}_{\text{conv2}}^{(d)} = \sum_{k=0}^{K-1} \mathbf{W}_k^{(d,2)} \mathbf{h}_{t-d\cdot k,\text{drop1}}^{(d)} + \mathbf{b}^{(d,2)}$ // *Second dilated convolution*
    $\mathbf{h}_{\text{bn2}}^{(d)} = \gamma^{(d,2)} \frac{\mathbf{h}_{\text{conv2}}^{(d)}-\mu^{(d,2)}}{\sqrt{\sigma^{2(d,2)}+\epsilon}} + \beta^{(d,2)}$ // *Batch normalization*
    $\mathbf{h}_{\text{drop2}}^{(d)} = \text{Dropout}(\mathbf{h}_{\text{bn2}}^{(d)}, 0.2)$ // *Dropout regularization*
    if dimensions match **then**
        $\mathbf{h}_{\text{skip}}^{(d)} = \mathbf{h}^{(d-1)}$ // *Skip connection*
    else
        $\mathbf{h}_{\text{skip}}^{(d)} = \sum_{k=0}^{K-1} \mathbf{W}_k^{(d,\text{skip})} \mathbf{h}_{t-k}^{(d-1)} + \mathbf{b}^{(d,\text{skip})}$
    end if
    $\mathbf{h}^{(d)} = \max(0, \mathbf{h}_{\text{drop2}}^{(d)} + \mathbf{h}_{\text{skip}}^{(d)})$ // *Residual connection with ReLU*
end for
Global average pooling: $\mathbf{z} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{h}_t^{(32)}$
Dense layers with ReLU activation and dropout:
    $\mathbf{h}_{\text{dense1}} = \max(0, \mathbf{W}^{(1)}\mathbf{z} + \mathbf{b}^{(1)})$
    $\mathbf{h}_{\text{drop3}} = \text{Dropout}(\mathbf{h}_{\text{dense1}}, 0.3)$
    $\mathbf{h}_{\text{dense2}} = \max(0, \mathbf{W}^{(2)}\mathbf{h}_{\text{drop3}} + \mathbf{b}^{(2)})$
    $\mathbf{h}_{\text{drop4}} = \text{Dropout}(\mathbf{h}_{\text{dense2}}, 0.3)$
Final output layer (linear activation): $\hat{\mathbf{y}} = \mathbf{W}^{(3)}\mathbf{h}_{\text{drop4}} + \mathbf{b}^{(3)}$
// *Parameter update* Train with Adam optimizer ($\alpha = 0.0003$):
1_{st} moment estimate: $\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}$
2_{nd} moment estimate: $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L})^2$
Correction: $\hat{\mathbf{m}}_t = \mathbf{m}_t/(1-\beta_1^t)$, $\hat{\mathbf{v}}_t = \mathbf{v}_t/(1-\beta_2^t)$
Update: $\theta_{t+1} = \theta_t - \alpha \cdot \hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$
Apply bias correction: $\hat{y}_{\text{corrected}} = \hat{y} - (a\hat{y} + b)$
// *Evaluation* Evaluate model performance: calculate MSE, MAE & R^2
Return: Trained model, evaluation metrics for each sequence length

For this RNN, we experimented with the sequence length hyperparameter. Once again, we implemented a range of sizes from 5 to 50 timestamps to test, and the results of each model's performance are reflected in the three graphs.

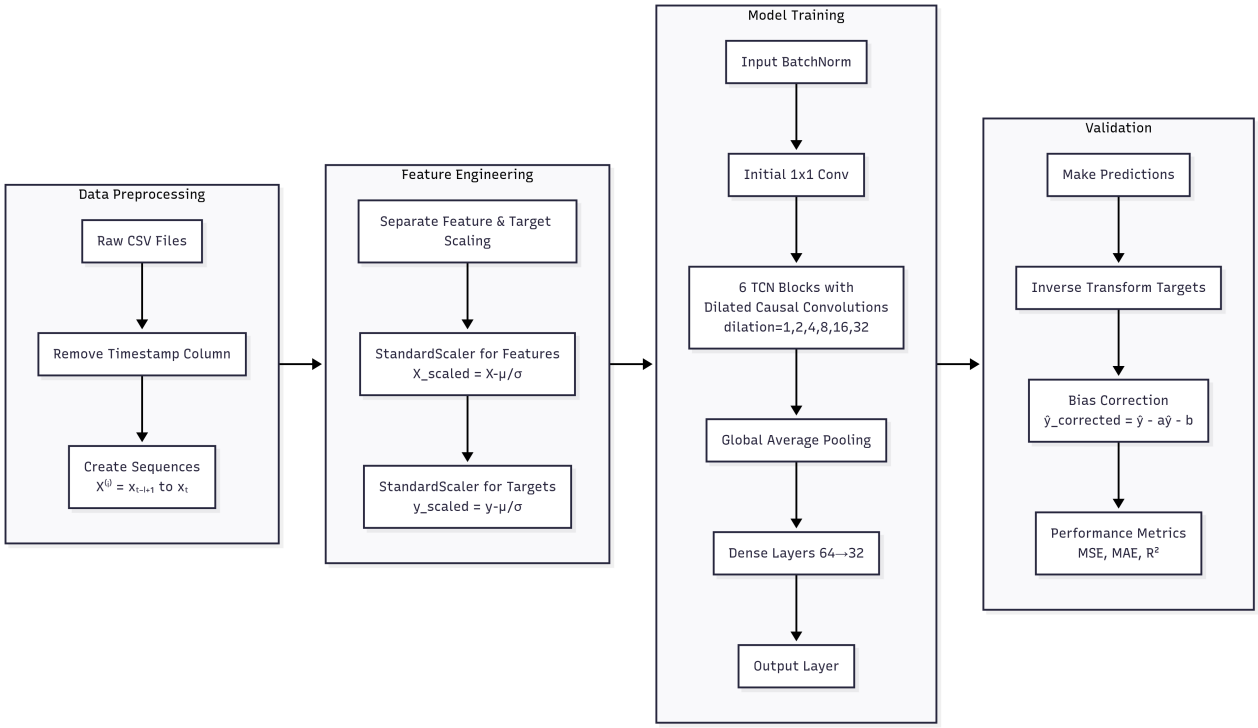

Figure 14: Flowchart of TCN training for steering motor velocity prediction

From the first graph on MSE vs. sequence length, as well as the third bar graph on MSE comparisons, we once again observe that the error trends across different lengths do reflect the bias-variance tradeoff, but for this model, the optimum is when the sequence has 20 timestamps. Although this does not align with the minimum LSTM at 10 timestamps, it makes sense because we designed the final layer of the TCN to have a receptive field of $1+2+4+8+16+32 = 63$ steps (irrespective of sequence length) in the past, which LSTM does not. Now, the problem with smaller sequence lengths is that the earlier layers, which have a smaller receptive field, have less data to convolve over and can starve these layers, while longer sequence lengths would dilute the important recent signals.

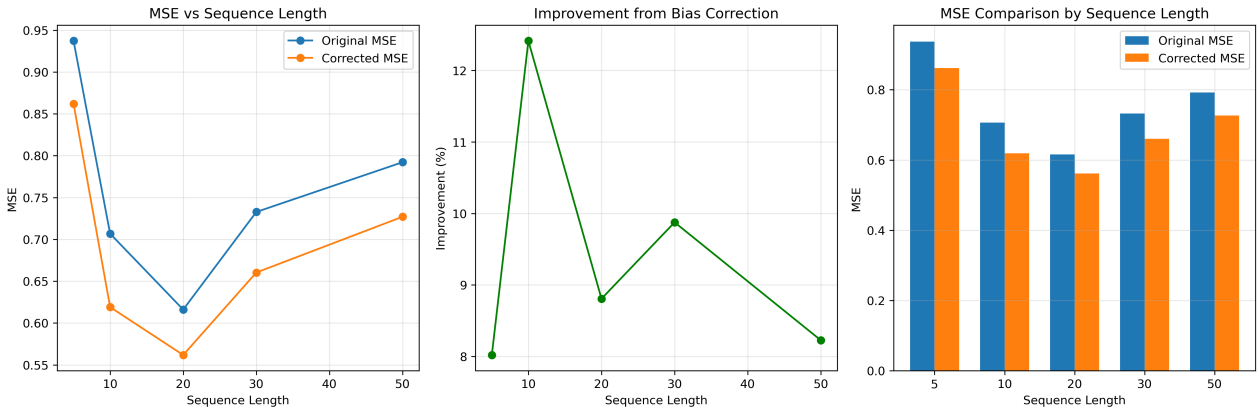

Figure 15: Comparison of TCN sequence lengths for minimum error

Once again, the bias correction terms reflected effective error reduction from the second graph. As we can see, the improvement was at least 8%, generally greater than the LSTM improvement range of

5%-9%. Let us zoom in to the TCN model with sequence length 20 and interpret the evaluation metrics, where bias correction is already applied.

Steering Motor ID	**MSE**		**MAE**		**R^2**	
	Value	Change%	Value	Change%	Value	Change%
Left Front	0.636085	(-50.46%)	0.414445	(-25.06%)	0.776164	(+14.69%)
Right Front	0.528976	(-41.15%)	0.374810	(-19.47%)	0.805545	(+2.39%)
Left Back	0.494236	(-47.81%)	0.369160	(-22.36%)	0.804721	(+3.86%)
Right Back	0.588086	(-25.11%)	0.378602	(-15.51%)	0.802782	(-1.80%)

Table 4: TCN model evaluation and improvement percentage

For MSE terms, all errors have been reduced by at least 25%. Similarly, for the MAE error, each term has improved by a minimum of 15%. These numbers reflect more consistency in the predictors across models, unlike in the LSTM table, where there was one term that experienced an increase in the MSE/MAE error. Furthermore, if we observe the R-squared values, not only are most terms consistently distributed around 0.8, almost all terms experienced an increase in determination. Even the right back motor that experienced a decrease in the R2 coefficient has a value above 0.8. In general, this indicates that the model has a stronger and more robust understanding of the system.

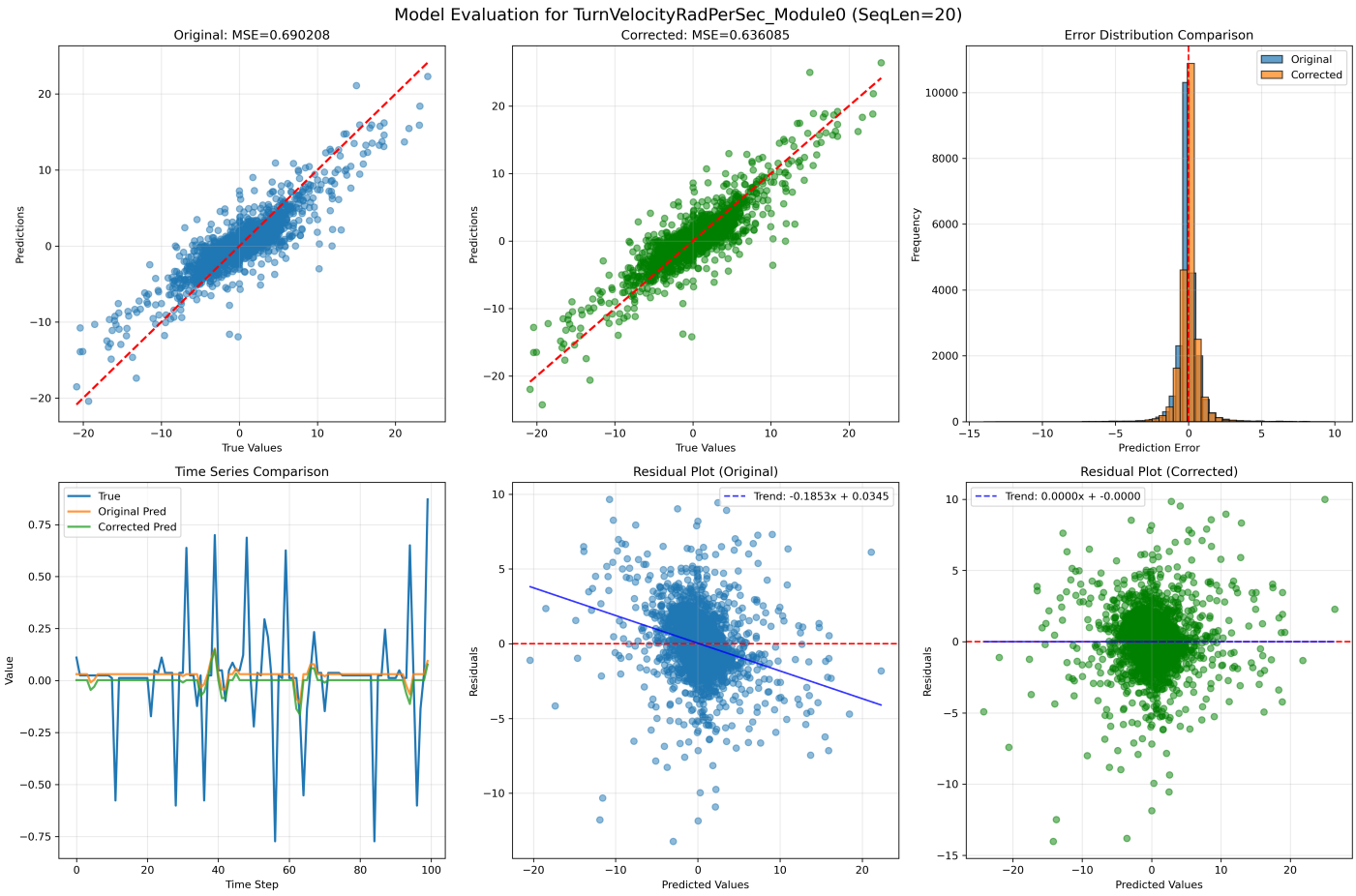

Figure 16: Diagnostic plots for steering motor using the TCN model (seqLen=20)

We can see as usual that the residual plot has identified the principal axis, depicted by the blue trendline, and it is corrected to form the green residual plot. For this motor, we see that the correction

is quite effective as the gradient offset used to be almost –0.2, and it yielded a 0.06 decrease in MSE metrics from 0.69 to 0.63. The time series comparison that has been automatically chosen by this model when it is executed is not the best representation of the performance of the velocity predictor. You can see that the ground truth range was less than 2, while previous comparison graphs had ranges of at least 4. Still, we can extract from the graph that the model is still not the best at predicting velocities with small magnitudes. This makes sense because when the velocity is so small, there is minimal inertia and more nonlinear components such as friction and backlash. However, from the evaluation metrics and the R-squared value, we can conclude that the TCN approach created the most accurate model to simulate steering motors.

Until now, we have analyzed the results of all the models we designed and experimented with to predict steering motor velocities. The mean and variance of the MSE, MAE, and R-square metrics of each model for steering predictions have been summarized in the following table for review.

Model	**MSE**		**MAE**		**R^2**	
	μ	σ^2	μ	σ^2	μ	σ^2
Linear	0.978755	0.034509	0.485512	0.001617	0.763953	0.002777
FNN	1.022155	0.010373	0.517022	0.000513	0.793672	0.000357
LSTM	0.707978	0.024756	0.432232	0.001770	0.793203	0.002074
TCN	0.561846	0.002910	0.384254	0.000315	0.797303	0.000150

Table 5: Mean and variance comparison of motor model evaluation metrics

Overall, TCN has the best performance in terms of having the lowest mean error terms and the highest R-squared value, suggesting the highest accuracy. It also has the lowest variance among all models, which reflects its consistency among the four motors. The second strongest performer is the LSTM, which has the second-best metrics in almost all criteria. FNN has certain improvements regarding the R2 value and low variance, but taking accuracy into account, it is the same level as our benchmark linear DC feedforward. For a more visual summary, see the bar graph below.

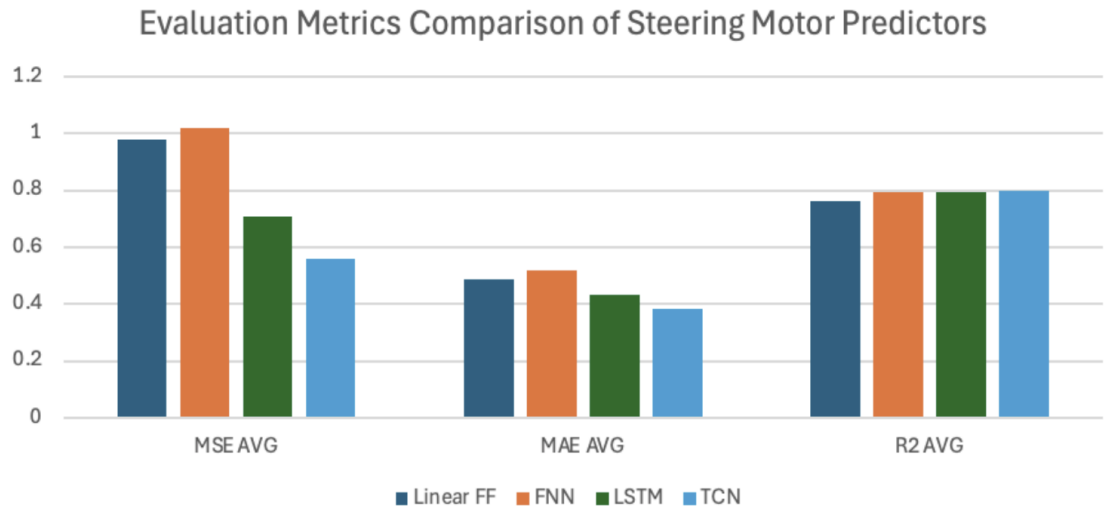

Figure 17: Comparison of motor model evaluation metrics

3.4 State-Space Benchmark (Chassis Modeling)

After the experiments on predicting single steering motor dynamics, we now shift our scope to predicting the full robot velocities. We will start with a linear state-space approach as a benchmark, and then we will experiment with the FNN, LSTM, and TCN models, respectively, to create a simulator with improved performance.

Many of the simulations in FRC use state-space models, shown in the following form. However, no teams have heard that a state-space model for swerve drivetrain has been developed. This model may have been attempted, but it could have turned out to not work. Still, we will try to create a linear state-space model as a benchmark. The main motivation of switching specs to predict for the robot-relative motion is that it would be more understandable compared to individual motors simulations, which would require the user to construct kinematics matrices themselves for custom drivetrain.

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

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$$
$$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k$$

$\mathbf{A}$	system matrix	$\mathbf{x}$	state vector
$\mathbf{B}$	input matrix	$\mathbf{u}$	input vector
$\mathbf{C}$	output matrix	$\mathbf{y}$	output vector
$\mathbf{D}$	feedthrough matrix		

The key idea is to identify the components necessary for predicting the states as well as the vector of inputs. It is quite clear that the inputs are the eight voltages applied to each motor, while the main state variables are the x and y linear velocities, as well as the rotational velocity omega. With deeper thought, it is not hard to see that absolute steering angles are also critical, as their trigonometric values determine the x-to-y linear velocity ratio. Thus, the absolute steering angles must be inserted as state variables. However, raw steering angles in radians would not work as the directions are constituted by their sine and cosine values, so we also included these trigonometric terms in the state vector.

Note that even though the trigonometric function is nonlinear, the state space model we built is still linear in its parameters. The reason why we built this model this way is derived from simple kinematics and DC motor assumptions.

For example, to predict the entire motion of the chassis, we know that it can be shown in the following form as a linear combination of the current with the sum of a coupling term between the steering angle and the velocity of the driving motor, a coupling term between the voltage and the steering angle for each of the four modules, and finally a voltage term. This is based on DC motor feedforward effects to predict driving acceleration and steering velocity from voltages.

$$\begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ \dot{\omega} \end{bmatrix} = \mathbf{A} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} + \sum_{i=1}^{4} \left(\mathbf{B}_{c,i} \begin{bmatrix} \cos(\alpha_i) \\ \sin(\alpha_i) \end{bmatrix} \omega_{d,i} + \mathbf{B}_{v,i} \begin{bmatrix} \cos(\alpha_i) \\ \sin(\alpha_i) \end{bmatrix} V_{d,i} \right) + \mathbf{C}\mathbf{u} \tag{27}$$

In this equation, the first and last terms on the RHS represent the isolated state matrix and input matrix. The middle term contains all the coupling, where the α_i term represents the angle of the i^{th}

steering motor in radians, $\omega_{d,i}$ refers to the angular velocity of the driving motors, and $V_{d,i}$ refers to the voltage applied to the driving motors.

Similarly, for driving and steering motor updates, we hypothesize according to the linear feedforward model, that there is the assumed linear relationship between voltage and velocities in the following forms, analogous to the benchmark example in section 3.2:

$$\dot{\alpha}_i = k_{s1} V_{s,i} + k_{s2} \sin(\alpha_i) + k_{s3} \cos(\alpha_i) + k_{s4} \omega \tag{28}$$

$$\dot{\omega}_{d,i} = k_{d1} V_{d,i} + k_{d2} \omega_{d,i} + k_{d3} v_x \cos(\alpha_i) + k_{d4} v_y \sin(\alpha_i) + k_{d5} \omega \tag{29}$$

Note that the prediction traget of the steering motors is angular velocity $\dot{\alpha}_i$, while the driving motors predict angular acceleration $\dot{\omega}_{d,i}$. Even though we have shown that the steering motor dynamics may have never been linear in the previous section, we are not plugging an NN model here because we would first like to create a completely linear benchmark.

Having designed the linear model, we then ran linear regression to train the parameters based on the following algorithm, which is almost identical to motor system identification:

Algorithm 5 State-Space Regression (statespace.py)

// Data processing Load and preprocess CSV files from folder
for each file f in directory **do**
    Extract timestamp t and all signal columns
    Separate drive signals (voltages, velocities) and steering signals
    Extract chassis velocities: ω, v_x, v_y
    Apply low-pass Butterworth filter: $H(s) = \frac{1}{1+(s/\omega_c)^{2n}}$
    Calculate derivatives: $\dot{v}_x = \frac{\Delta v_x}{\Delta t}, \dot{v}_y = \frac{\Delta v_y}{\Delta t}, \dot{\omega} = \frac{\Delta \omega}{\Delta t}$
end for
// Model operation Create state-space model with trigonometric features:
Create trigonometric features: $\cos\theta_i, \sin\theta_i$ for $i \in \{1, 2, 3, 4\}$
Create interaction terms: $v_x \cos\theta_i, v_x \sin\theta_i, v_y \cos\theta_i, v_y \sin\theta_i$
Create voltage interaction terms: $V_{\text{drive},i} \cos\theta_i, V_{\text{drive},i} \sin\theta_i$
Standardize features: $\mathbf{X}_{\text{scaled}} = \frac{\mathbf{X} - \mu_{\mathbf{X}}}{\sigma_{\mathbf{X}}}$
Standardize targets: $\mathbf{y}_{\text{scaled}} = \frac{\mathbf{y} - \mu_{\mathbf{y}}}{\sigma_{\mathbf{y}}}$
Split data: $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}}$
For each output derivative $d \in \{\dot{v}_x, \dot{v}_y, \dot{\omega}, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\omega}_1, \dot{\omega}_2, \dot{\omega}_3, \dot{\omega}_4\}$:
    Solve ridge regression: $argmin_{\mathbf{w}^{(d)}} \|\mathbf{X}_{\text{train}} \mathbf{w}^{(d)} - \mathbf{y}^{(d)}_{\text{train}}\|_2^2 + \lambda \|\mathbf{w}^{(d)}\|_2^2$
// Evaluation For each state derivative:
Calculate MSE: $\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
Calculate MAE: $\frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$
Calculate R^2: $1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}$
Generate evaluation plots: scatter, time series, residual, distribution
Return: Trained models, scalers, evaluation metrics for all state derivatives

After running the linear regression, we approached the following results for the error metrics of x-velocity, y-velocity, and angular velocity, which turned out to be much worse than anticipated, given the previous experience with modeling single motors. Referring to table 6, we can get a picture of how much off it is to model the swerve's behavior linearly.

Feature	MSE	MAE	R^2
v_x (m/s)	1.773928	0.749413	0.150959
v_y (m/s)	1.719082	0.738274	0.149419
Ω (rad/s)	5.743553	0.892623	0.059926

Table 6: Linear state-space model evaluation for linear and angular velocities

Note that the units of the x and y linear velocities are meters per second, so getting an MSE metric above 1 is devastating, as it stands prediction accuracy above 1 meters per second. Similarly, all the R2 coefficients are very low, which implies that the model does not fit the data points consistently. Let us pull out the diagnostics for one of the linear velocity components and the angular velocity component for better visualization.

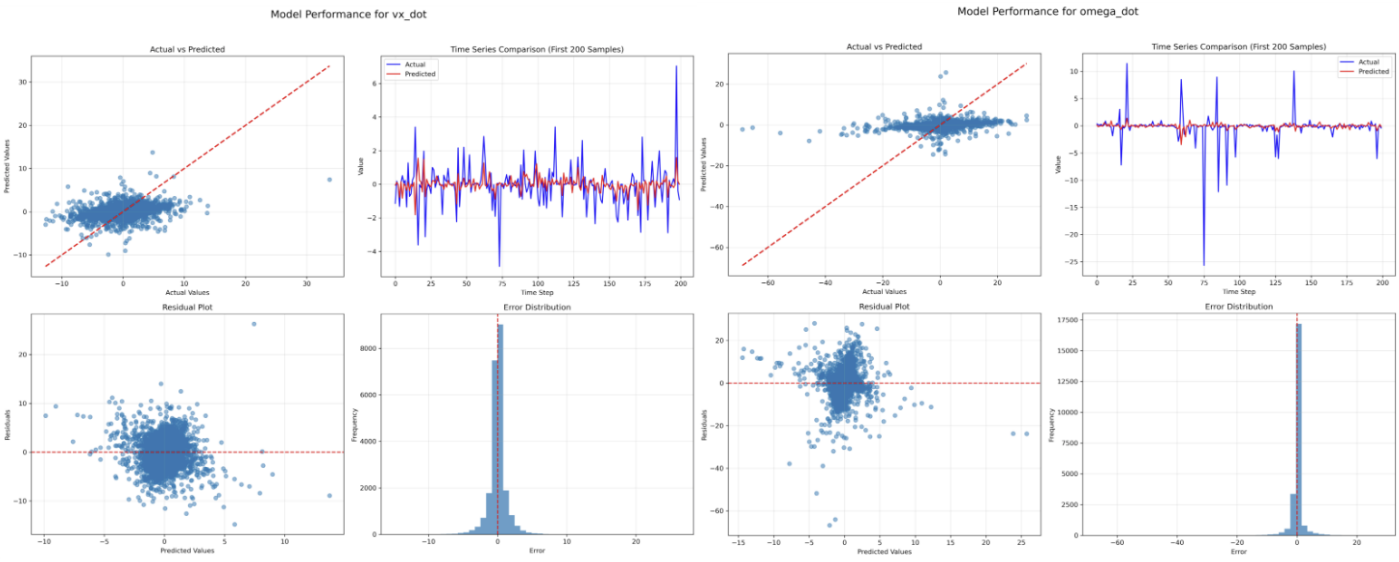

Figure 18: Diagnostic plots for v_x and ω predictions of state space model

Note that we don't show the plots for the y-component of the velocity in this paper since it is analogous to the x-component, but it is provided in the supplementary folder. As we can see from the graphs, there is a huge bias if we refer to the actual vs. predicted graphs. In fact, the stretch of the plots reflects horizontal stretches, and the small slope can also be visualized in the time series comparison graphs, where we can clearly see that the predicted values, although they seem to represent the peaks and troughs of the shape of the velocity, produce very small magnitudes and thus always undershoot. This result shows that it might not be a good idea to try to model the dynamics of the swerve only with a linear combination of voltages and velocities and motivates our deeper exploration of using alternatives to nonlinear neural networks for modeling.

3.5 Innovations of NN-based Approaches (Chassis Modeling)

Like individual motor predictors, we experimented with FNN, LSTM, and TCN models to simulate the entire chassis. The main architecture for pre-processing, training and evaluation is like in the previous section; only a few hyperparameters were changed to allow the learning of the increased input and output sizes. As the design and analysis processes are analogous to our single-motor models, the following pages will focus on the model with the best performance and a summary of all the models' results and metrics. The complete set of diagnostics is included, of course, in the supplementary documents. In the table below are the MSE, MAE and R2 metrics for the three outputs. Note that bias correction has been applied and the optimal sequence length has been selected from the LSTM and TCN trials.

Model	Metrics	v_x (m/s)		v_y (m/s)		Ω (rad/s)	
		Value	Change%	Value	Change%	Value	Change%
FNN	MSE	0.006536	(-99.63%)	0.006694	(-99.61%)	0.026945	(-99.53%)
	MAE	0.047189	(-93.72%)	0.049119	(-93.35%)	0.103719	(-88.47%)
	R^2	0.988551	(+552.2%)	0.990271	(+562.6%)	0.985922	(+1542%)
LSTM	MSE	0.007249	(-99.59%)	0.006791	(-99.61%)	0.025059	(-99.56%)
	MAE	0.048061	(-93.59%)	0.052013	(-92.96%)	0.100534	(-88.83%)
	R^2	0.987910	(+550.9%)	0.985731	(+559.5%)	0.991728	(+1574%)
TCN	MSE	0.004339	(-99.76%)	0.004049	(-99.76%)	0.022012	(-99.62%)
	MAE	0.043957	(-94.13%)	0.042334	(-94.27%)	0.093128	(-90.70%)
	R^2	0.991173	(+563.9%)	0.992641	(+570.8%)	0.988810	(+1582%)

Table 7: Nonlinear NN model comparison and improvement percentage

As we can see in this table, the TCN model seems to have the best performance in terms of having the least MSE and MAE metrics for all three velocity decompositions of Vx, Vy, and omega. Furthermore, the coefficient of determination is also the highest for the two linear velocities when we apply the TCN. The only metrics where the TCN is not the numerical best, by an insignificant margin, are the R-squared value of the angular velocity.

The algorithm of the TCN model can be summarized as a flow chart.

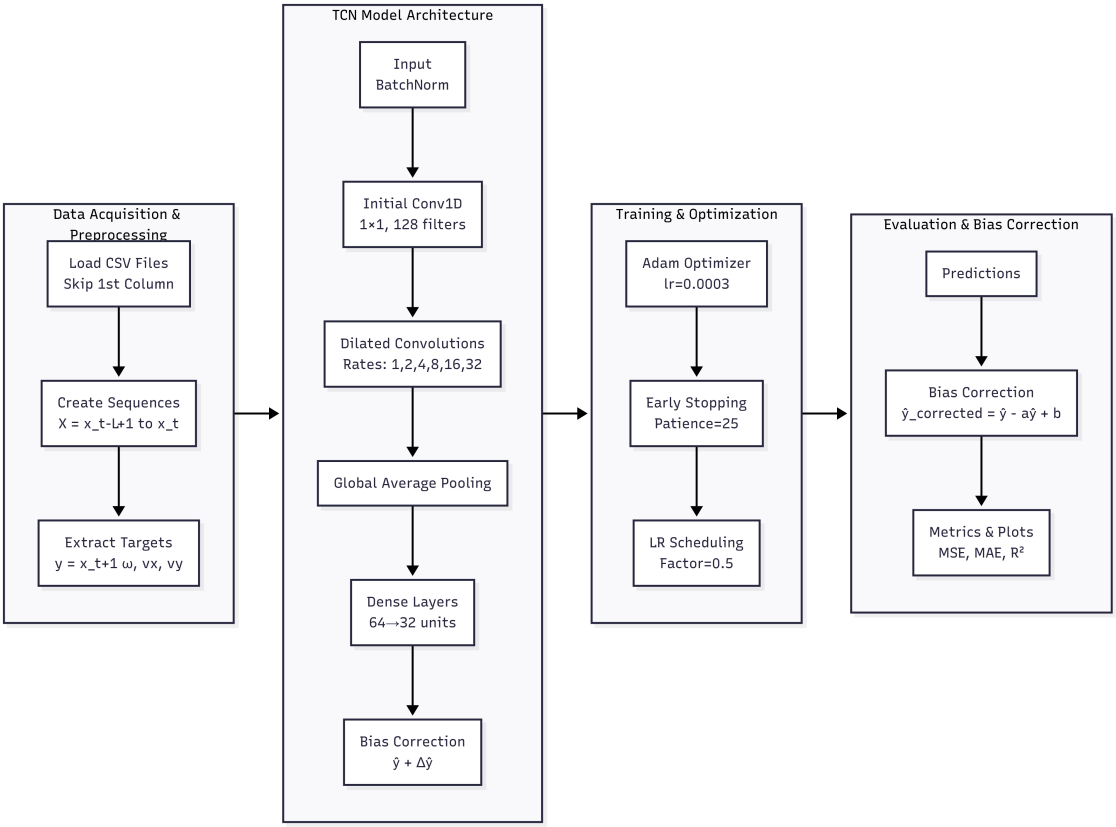

Figure 19: TCN architecture for full-chassis velocity prediction

Next, we will take a closer look at the results of the TCN model. Just like before, the model experimented with different sequence lengths. Interestingly, the model with the least error metrics also turns

out to be when the past 20 timestamps are fed in, just like the single motor models.

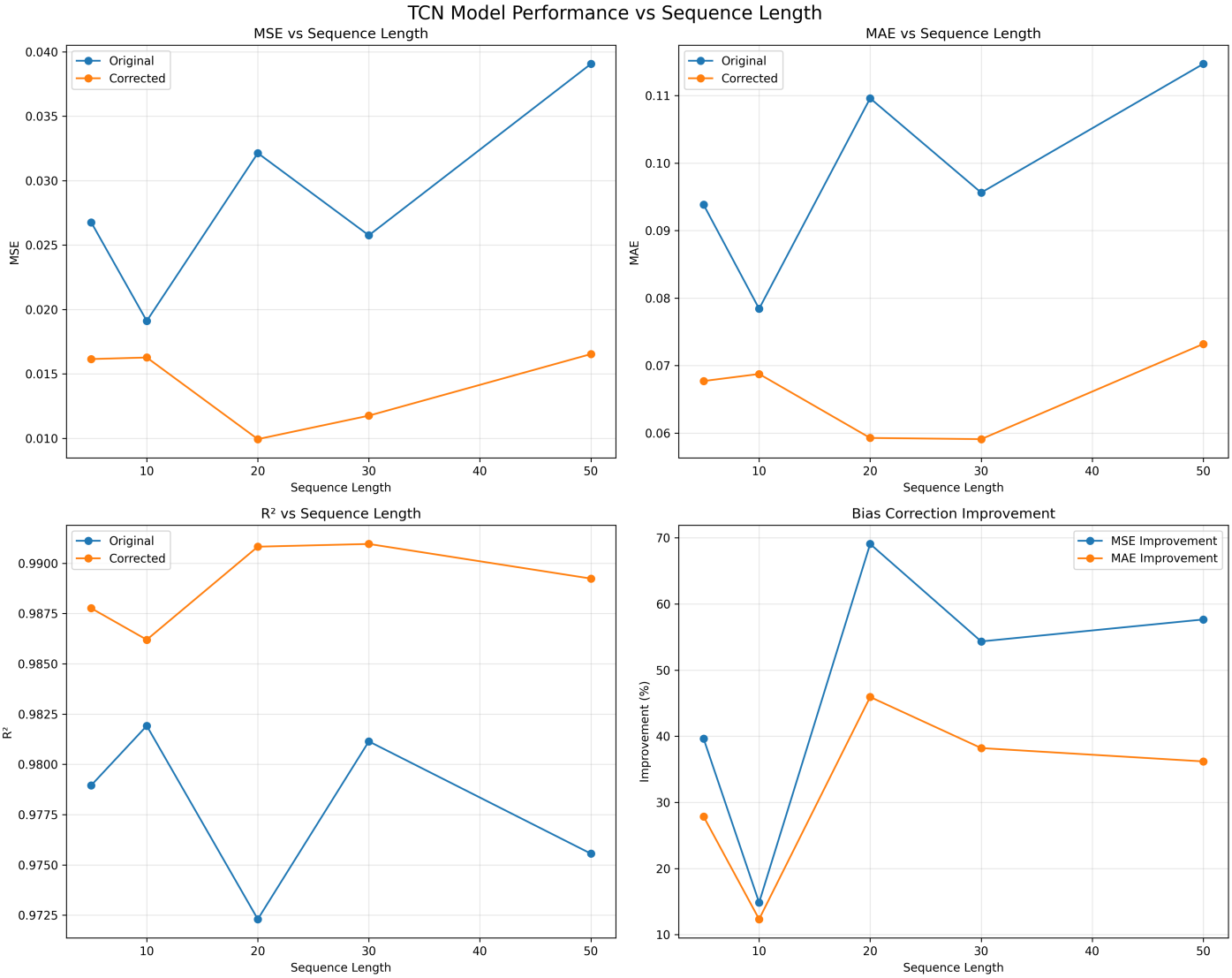

Figure 20: TCN full-chassis performance vs. sequence length

Just as observed before, after the bias correction, the MSE metrics of the model represented by the orange line graph resemble a convex function, which is suggested by the bias-variance tradeoff as we modify the sequence length.

Next, let us zoom into the specific diagnostics of the TCN model with 20 timestamps of memory. As the graphs display, a key observation is that for both graphs, the principal axis that has been identified by the linear regression on the blue residual plot deviates greatly from the horizontal red axis, showing that we expect a great reduction in error after correction. Indeed, if we observe the actual vs predicted plot after correction, the cluster of dots fits the optimal red diagonal very well. This change is more noticeable in the evaluation plots for x-velocity. The time series comparison graph also shows a great fit for the VX velocity at high magnitude.

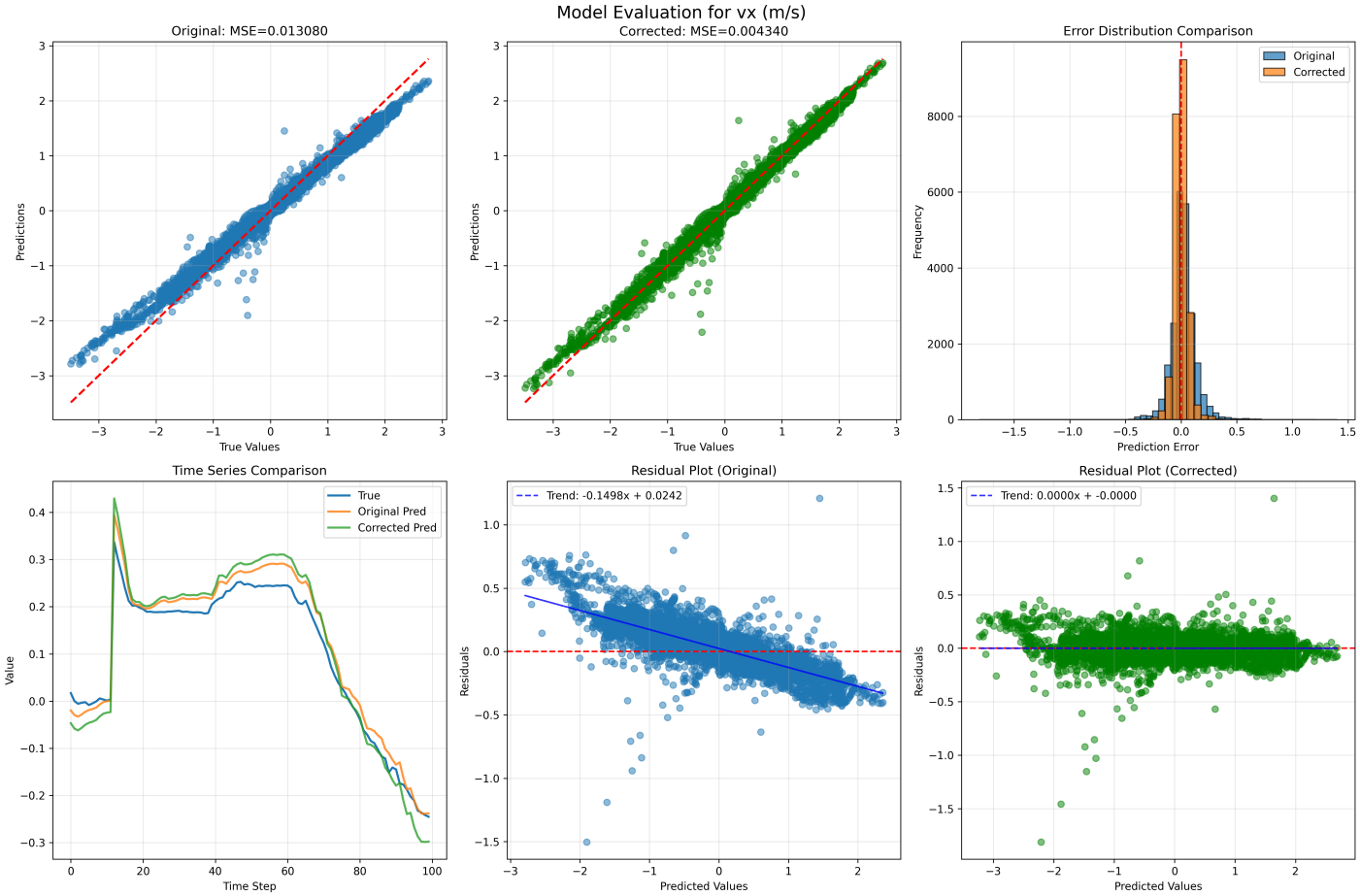

Figure 21: Diagnostic plots for TCN full-chassis v_x prediction (seqLen=20)

Next, if we move on to review the diagnostics for angular velocity, the error distribution comparison graph shows how correction shifted more predictions to a near-zero error. However, if we loop at the time series comparison, while the green corrected line does seem to move its average closer to the average of the ground truth, it is still hard to capture the rotation dynamics when magnitudes are small. Luckily, the displayed time series sample had less than 3 degrees per second of rotation, thus it is trivial.

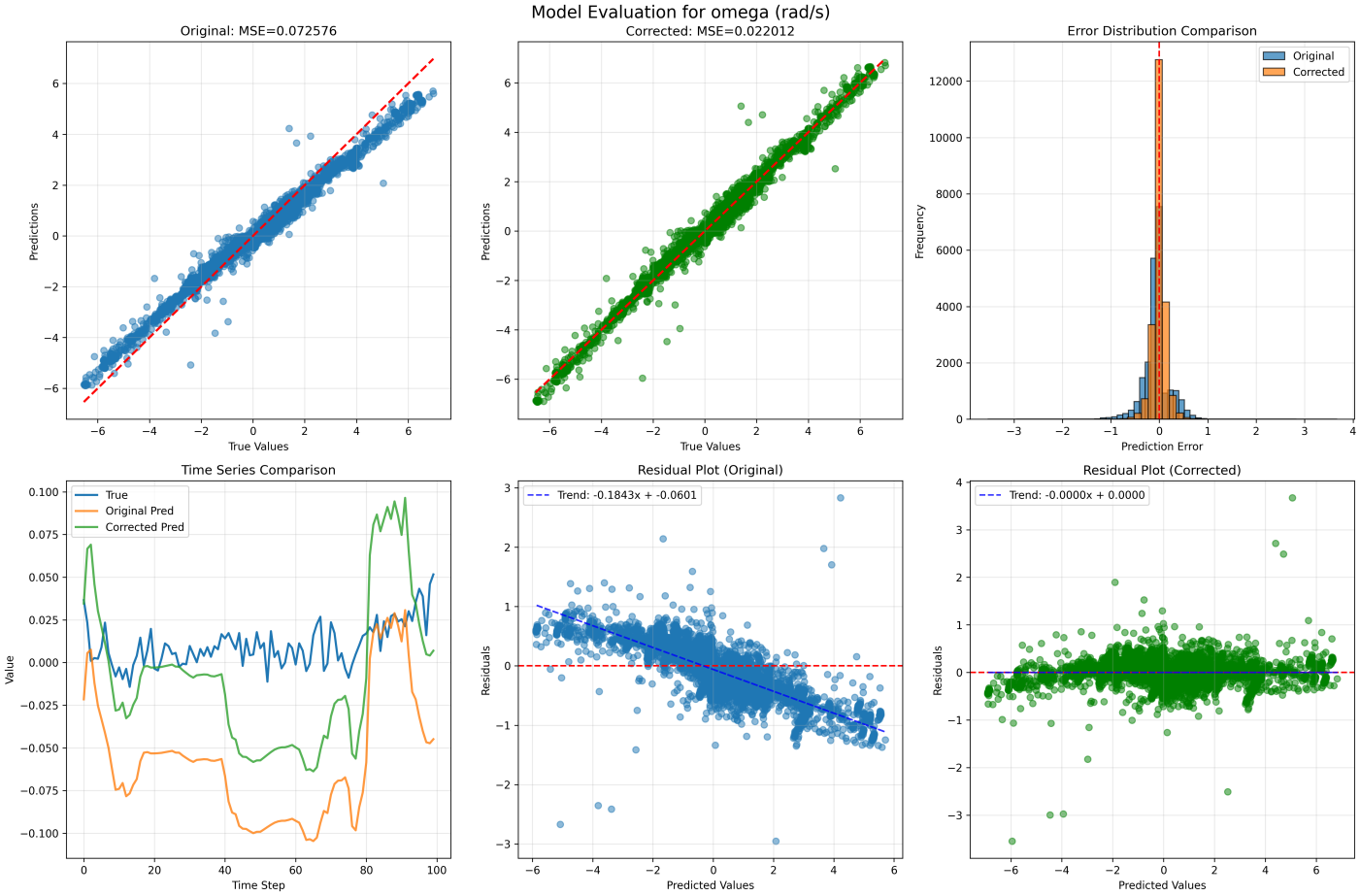

Figure 22: Diagnostic plots for TCN full-chassis ω prediction (seqLen=20)

Finally, looking back at the improvement in leveraging the power of neural networks to model the dynamics of a swerve drivetrain, nonlinear predictors perform significantly better than linear models in predicting full chassis dynamics.

This concludes the first main section on modeling. In retrospect, when focused on modeling a single motor in a complex system, the linear DC motor feedforward benchmark was a very high starting point, and it justifies the popular model that has been applied across many robotics applications, especially in the FRC community. Throughout the experiment, we found that adding nonlinearity can yield a better fit and smaller error when we take the state information from a few past timestamps, as in the LSTM and TCN models, but we need to find an optimal sequence length for bias-variance tradeoff considerations. Next, for the second section where we attempted to model the complex system directly, the state-space model benchmark performed very poorly and barely reflected the swerve's behavior. Hence, adding nonlinearity with all three methods of FNN, LSTM, and TCN yielded tremendous improvements and made great predictors.

4 Swerve Motion Profiling

4.1 Motion Profiling Task

In this section, we will focus on the second topic of the research, which is planning an optimal policy with machine learning approaches. This section has less focus compared to the first topic on modeling the swerve drivetrain with different scopes and methods.

Our choice of optimal policy is to find a motion profile (velocity curve) for the swerve chassis to execute while following a simple straight path: driving forwards for 3 meters with no rotation and changes in direction, and the value that we try to minimize is the time taken to reach a designated region of tolerance around that point. Notice that for this path, many simpler drivetrain models would suffice. However, the swerve chassis is the only mechanism available, but it happens to reflect the power of machine learning techniques quite well.

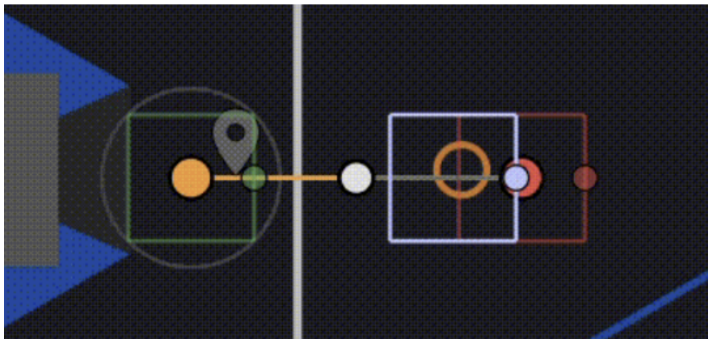

Figure 23: Visualization of the 3-meter straight path via FRC path planning tool

The graphics show the path in orange, the starting and ending poses in the green and red boxes, and the position of the simulated robot in the blue box. This visualization is done in a path-planning tool developed by the FRC community.

4.2 Trapezoidal Profile Benchmark

For building a benchmark policy, a trapezoidal motion profile was used, which is written as a method in the FRC coding library. Exactly like the trapezoid shape, the linear velocity curve of the robot's movement would loop like an isosceles trapezoid with fixed acceleration and deceleration, as well as a flat velocity limit. A visual representation is included in Figure 24.

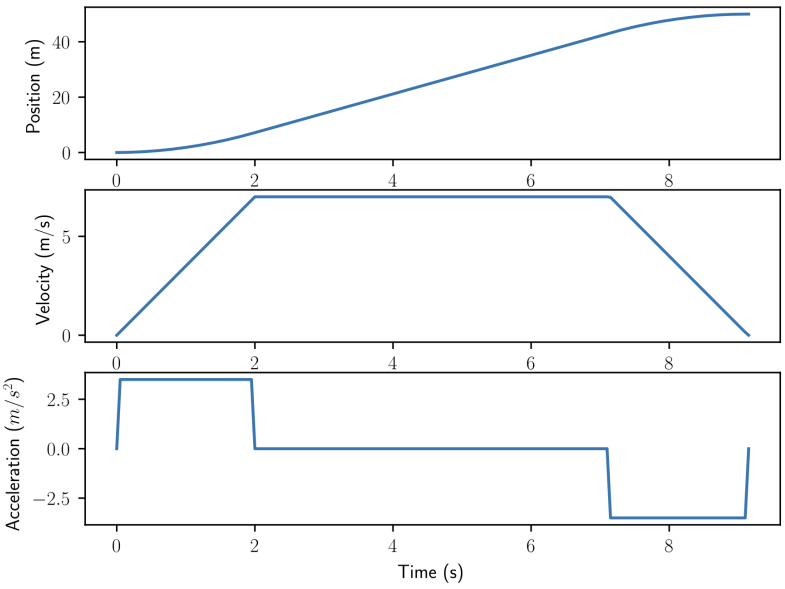

Figure 24: Trapezoidal profile graphs

The strength of a trapezoidal motion profile is primarily its simplicity of motion and calculation, with an efficient change of acceleration. There are only 3 phases of the motion: acceleration, linear motion, and deceleration, where the relationship between time and distance is quadratic, making modeling, computation, and coding easier. Engineers can easily set the parameters of motion that fit their needs, hence acquiring the expected motion. In contrast, the discontinuous jerk of the trapezoidal profile, caused by an abrupt acceleration shift, poses threats to the lightweight design-prioritized components of FRC robots. FRC robots usually depend on standard plastic gears, timing belts, and small-scale motors to meet weight and dimension constraints; these parts are not designed to withstand repeated impulsive jerks from the abrupt acceleration transitions of the profile.

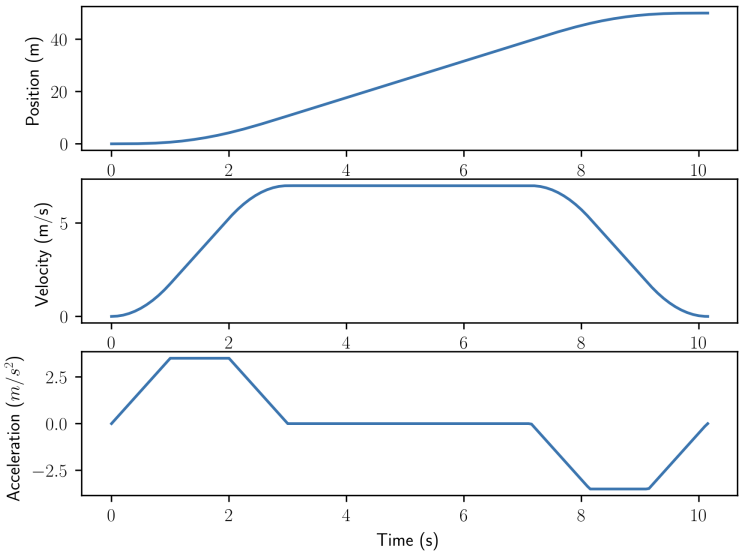

Figure 25: S-curve profile graphs

S-curve is an existing solution to the limitations above, by introducing the variable jerk over time,

hence smoothening the instantaneous change of acceleration. Without sudden jerk, the vibration and mechanical shock are minimized, which significantly stabilizes the motion, minimizes the vibration and mechanical shock, reducing wear and damage. Which improves the precision and lifespan of equipment.

In the benchmark experiment, we capped the velocity and acceleration to $3M/S$ and $2M/S^2$, which is slightly below the empirical max values for safety. We ran five trials on the swerve for it to execute the 3-meter straight path, and recorded the time taken:

	Tr 1	**Tr 2**	**Tr 3**	**Tr 4**	**Tr 5**	**Avg**	**Abs Uncertainty**
Time (s)	4.02	3.99	4.10	4.08	3.88	4.014	± 0.11

Table 8: Time Measurements with Average and Uncertainty

These results show competent speed and consistency, so that the trapezoidal profile can guide the robot to its target destination's tolerance range. This justifies why many teams still use trapezoidal profiles on a regular basis and as their elementary path-planning policy. Nevertheless, our nonlinear velocity policy trained from reinforcement learning algorithms reflected interesting patterns and performance boosts.

4.3 Innovations of RL-based Approaches (Motion Profiling)

Our idea is to design a reinforcement learning algorithm using OpenAI's gymnasium library. The algorithm we implemented is a model-free, policy-based Proximal Policy Optimization (PPO) algorithm. The PPO is especially useful for continuous action spaces, which is helpful in many robotics applications.

As the path is relatively simple, we tried to minimize the dimensions of states and actions, which are shown below, along with the transition dynamics. Note how the transition dynamics are modeled by the linear DC motor feedforward model because this path requires only highly linear driving motors and because we wanted to isolate the effects of reinforcement learning on the performance of the motion profile. The task is defined as a Markov Decision Process (MDP) with the following components.

State Space: This vector describes all the possible states that the robot can be in, including the chassis velocity and the angular velocity of each motor.

$$\mathbf{s}_t = \begin{bmatrix} y_t \\ \omega_{0,t} \\ \omega_{1,t} \\ \omega_{2,t} \\ \omega_{3,t} \end{bmatrix} \tag{30}$$

where y_t is the robot position and $\omega_{i,t}$ is the angular velocity of the motor i. Considering that this path does not have any rotation to account for, all $\omega_{i,t}$ are equal, but are still included for their respective feedforward predictions.

Action Space: The only action is the velocity setpoint, hence the action space is only one dimensional and easier to optimize. However, still recall that the action is clipped to a limit of 3 mps.

$$\mathbf{a}_t = [v_{\text{desired}}], \quad a_t \in [-v_{\max}, v_{\max}] \tag{31}$$

The agent outputs the desired linear velocity.

Transition Dynamics: This module reflects how the RL environment blends the state space with the action space to predict the next state. Note that the following are based on DC feedforward.

$$V_{i,t} = k_{s,i} \cdot \text{sign}(\omega_{\text{desired}}) + k_{v,i} \cdot \omega_{\text{desired}} + k_{a,i} \cdot \alpha_{i,t} + c_i \tag{32}$$

$$\omega_{i,t+1} = \omega_{i,t} + \alpha_{i,t} \cdot \Delta t \tag{33}$$

$$y_{t+1} = y_t + \left(\frac{1}{4} \sum_{i=0}^{3} \frac{\omega_{i,t}}{R_{\text{gear}}} \right) \cdot r_{\text{wheel}} \cdot \Delta t \tag{34}$$

The voltages $V_{i,t}$ are reduced to $[-12, 12]$ V. The dynamics are deterministic and physics-based.

If we had used a nonlinear NN-based predictor as what we experimented with in section one, we already knew empirically that they are worse at modeling driving motors, and we also wanted to isolate to analyze the effects of reinforcement learning, which would be much harder if we applied an NN model for simulation as well.

In essence, our state and action designs made sure that there was completely zero rotation of the drivetrain, and that motion was only on one linear axis, which is the preplanned path. The single variable trained is the robot's velocity, where motor velocities are calculated by applying a constant gear-ratio reduction, and then it propagates to determining the motor voltages via DC motor feedforward to ensure that voltages are within the legal range of [-12V, +12V].

Reward Function: Our reward function, which the learned policies try to maximize, focuses on the third term under the terminal reward, which is a penalty on the chassis' linear velocity multiplied by a factor of –50. This is to ensure that the robot learns proper deceleration such that it reaches zero velocity within the tolerance range of 5 centimeters.

The reward function R is calculated at each time step as follows:

$$\begin{aligned} R = {} & 10 \cdot \max(0, d_t - d_{t+1}) \boxed{\text{distance}} - 0.05 \cdot \sum_{i=0}^{3} \max(0, |V_i| - 12) \boxed{\text{voltage}} \\ & - 0.00005 \cdot \Delta t \cdot \sum_{i=0}^{3} V_i^2 \boxed{\text{energy efficiency}} - \text{Var}(\omega_0, \omega_1, \omega_2, \omega_3) \boxed{\text{consistency}} \\ & - 0.02 \cdot |v_t - v_{t-1}| \boxed{\text{acceleration smoothing}} + R_{\textbf{guidance}} + R_{\textbf{terminal}} \end{aligned} \tag{35}$$

Where the guidance reward R_{guidance} is defined as:

$$R_{\text{guidance}} = \begin{cases} -0.5 + 0.3 \cdot \min(v_t, 2.5) & \text{if } y_t < 2.8 \text{ and } d_{t+1} > 0.5 \\ -0.1 \cdot \left(1 - e^{-5(3 - y_t)}\right) \cdot v_t^2 & \text{if } y_t \geq 2.0 \end{cases} \tag{36}$$

And the terminal reward R_{terminal} is defined as:

$$R_{\text{terminal}} = \begin{cases} -100 & \text{if } y_{t+1} < y_{\min} \text{ or } y_{t+1} > y_{\max} \\ 1000 & \text{if } d_{t+1} \leq 0.05 \text{ and } |v_t| < 0.01 \\ -50 \cdot |v_t| & \text{if } d_{t+1} \leq 0.05 \text{ and } |v_t| \geq 0.01 \end{cases} \tag{37}$$

The policy training process uses a curriculum learning architecture that adjusts the difficulties of the environment, such as tolerance, based on the performance of the agent in recent episodes. Hence, the initial tolerances were very loose and the idea is that this design allowed more exploration, which enabled us to discover more complex models and profiles.

To get a more holistic view of the curriculum learning algorithm, please refer to the flowchart:

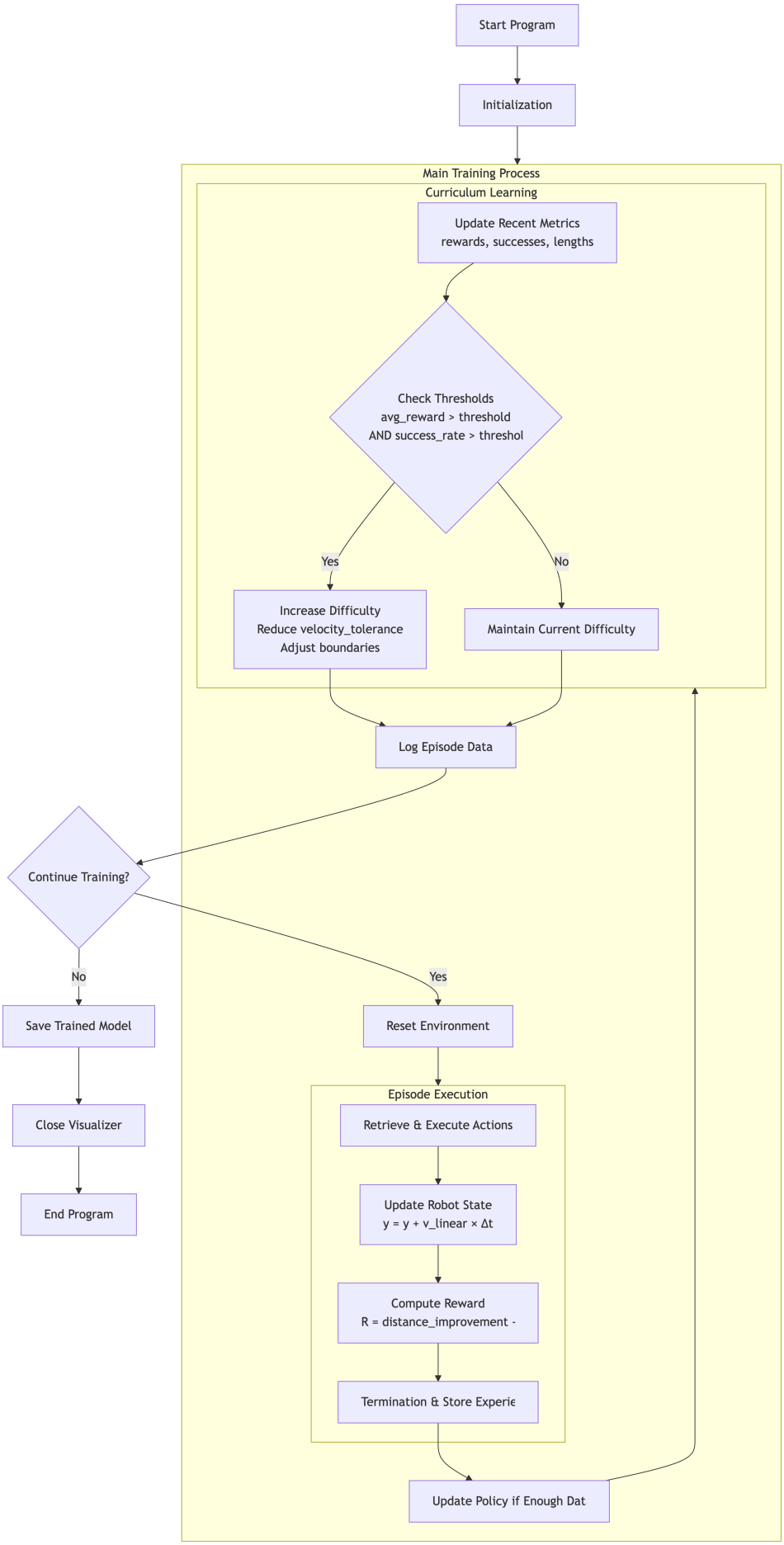

Figure 26: Flowchart of curriculum learning for RL (PPO algorithm)

To visualize the progress of training, we took a sequence of best-performing policies that yielded the maximum reward compared to its predecessors. The first two rows show the linear velocity graph of the chassis, while the third and fourth rows show the position change.

In the velocity graphs, we can see that the robot was almost randomly jittering up and down with its velocity, going back and forth but not really showing a target direction. This follows the fourth and eighth plots as well. As we move on to the 15th and 32nd velocity plots, we can see that there seems to be a trend where the robot is moving towards a target, and an important piece of evidence is that the initial velocity is increasing. As we move on towards the 186th to the 433rd graphs, we can see the acceleration and deceleration trends quite clearly, and the oscillation also smoothens out over training. Looking at the best-performing 433rd velocity graph, we can identify a few interesting features. Firstly, the shape does not look trapezoidal or like an S-curve, as it has a very pointy velocity peak and a low-velocity tail near the end of the episode. This suggests that there could be a small deceleration period after the fast activation acceleration to drop the cruising velocity and that the velocity must be significantly decreased for the final period to ensure accurate control.

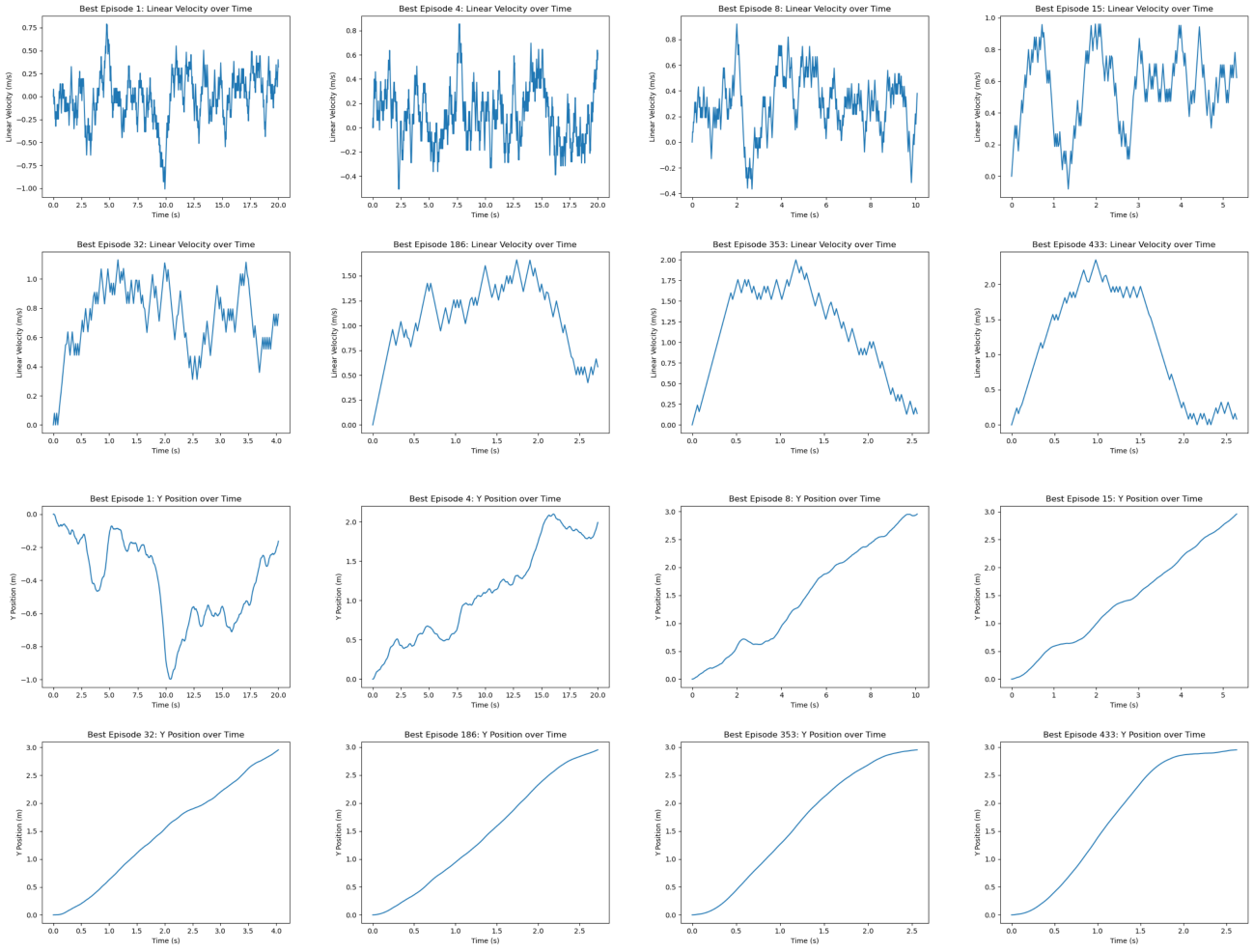

Figure 27: RL training progress (1st-2nd row: velocity; 3rd-4th row: displacement)

In the third and fourth rows, the position-time graphs can also show how the robot moved from 0 to 3 meters from another perspective. As we can see in the first graph, the robot had absolutely no idea where to go. From the coordinates on the y-axis, we can see that it traveled to the negative zone. However, it quickly learned to go to the positive axis, as shown in episode 4, yet it does not reach the designated range. Luckily, the policy evolved quickly and reached the goal from episode 8. From the 8th graph all the way to the 433rd graph, the robot is mainly focused on 1) optimizing the time

taken to reach the destination and 2) reducing the final arrival velocity to 0, which is coming to a stop. As we can see in the final 433rd time-position graph, the acceleration and deceleration periods do not look balanced or symmetric compared to the trapezoidal or S-curve graphs. Instead, the deceleration period seems to take longer, likely for execution accuracy, which matches our analysis for the previous velocity-time graph.

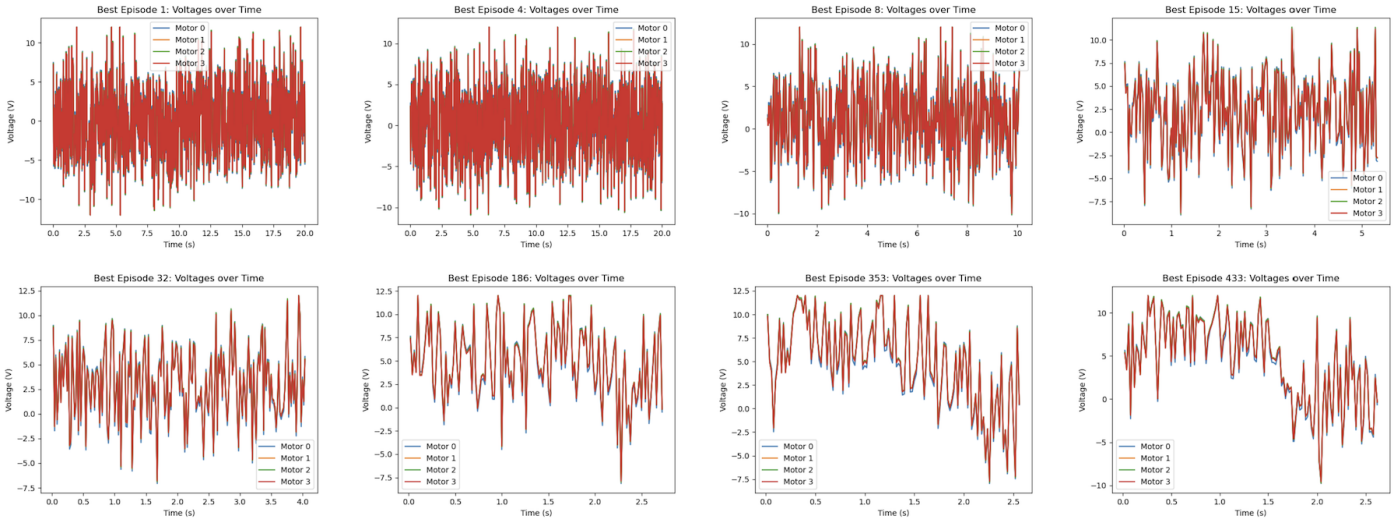

Figure 28: RL training progress (voltages)

In the next step, target voltages to be fed into the robot's driving motors were determined by linear DC feedforward parameters that were calculated in the modeling section. We can clearly see how there was excessive jittering from episodes 1 through 8, and then as we moved on from episode 15 to 186, the zigzag was less frequent. As we reached episodes 353 and 433, the trends are quite clear, and the remaining jittering can be reduced by a choice of linear filter. Looking at the trends in the applied-voltage graph of episode 433, it can be shown that the motors first had to be given an impetus voltage around 10 volts to force the 30-kilogram robot out of static friction. Then, it cruises at a median of roughly 7 volts until 1.5 seconds. After that, a decrease in voltage is observed, which shows the deceleration period of the profile. Notice how the voltages were negative for a short while in the last periods of deceleration, reflecting that the motor is applying a lot of power in counteracting the momentum of the robot. All these patterns align closely with the velocity-time and position-time graphs analyzed earlier.

Before deployment in the physical world, a crucial step is to perform filtering for noise reduction on the series of voltages trained. Notice that even though the trends are becoming visible in the voltages in episode 433, there is still noise to be reduced for safety concerns. Since all motors seem to behave almost identically in terms of voltages reverse-calculated from DC motor feedforward, we will visualize how filtering is done on one of the motor voltages.

As shown in Figure 29, a range of sequence lengths for a moving average factory was tested on the raw data, represented in the black line graph. This is once again a search for a relatively optimal sequence length, where longer sequences can lead to underfitting, whereas shorter lengths are ineffective in noise reduction. After experimentation, $n = 25$ appears to have represented the key trends of our trained profile, initializing at a voltage of 5 volts and ending at a near-zero voltage, which is what is desired. If we had increased the window size, then the deceleration curve would be greatly skewed like the purple graph; if we had decreased the window size, then zigzagging noises would be unresolved. This makes the window size of 25 the most reliable candidate to run the actual chassis with.

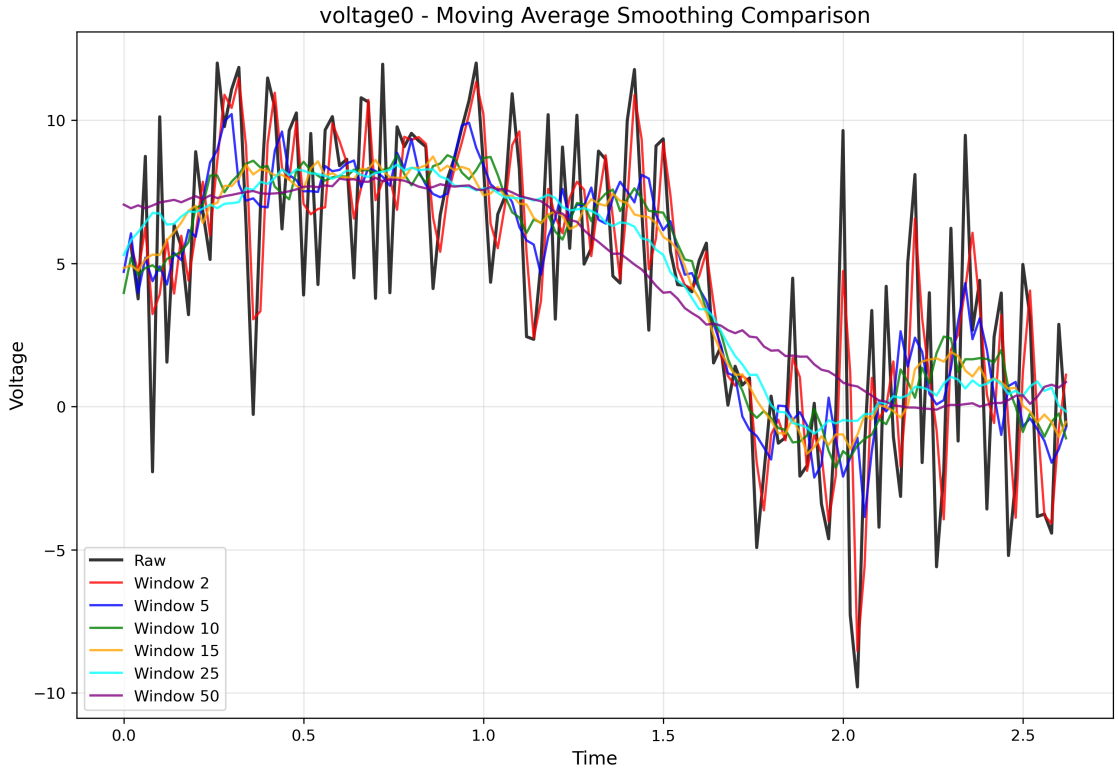

Figure 29: Moving average smoothing for RL-trained motor voltages

The complete csv file was processed with the $n = 25$ moving average factory and the robot was run by feeding the smoothed voltage values into a vector to be continuously executed by each motor at each timestamp, with increments of 20 ms, just as the RL was trained.

The results of five trials are summarized in the table below. The motion profile trained from the RL routine in a linear DC feedforward predictor environment was shown to perform 33% better than the trapezoidal profile policies in terms of the time elapsed. In terms of arrival precision, both the RL and trapezoidal policies were able to park the robot within a 3 cm radius of the target point, which is sufficient accuracy to perform most tasks in FRC games.

	Tr 1	**Tr 2**	**Tr 3**	**Tr 4**	**Tr 5**	**Avg**	**Abs Uncertainty**
Time (s)	2.57	2.61	2.67	2.66	2.59	2.62(-34.7%)	±0.02(-81.82%)

Table 9: Time Measurements with Average and Change %

Also note that the absolute uncertainty of the time elapsed also decreased significantly to less than 20% of the absolute error of the reference, suggesting that the prolonged period of mild deceleration could have improved the consistency of the arrival time. However, this needs to be further justified with more trials in the future.

This concludes the exploration and findings in the second section of our research. In summary, while traditional profiles such as the trapezoidal profile and S-curve are simpler to operate and more rigid with their form, policies trained from RL algorithms in a very high-fidelity environment can still lead to significant improvements, even in simple tasks such as traversing a straight line.

5 Conclusion

Throughout this process of intensive experimentation and analysis of an omnidirectional FRC robot drivetrain, a systematic comparison was conducted between linear versus nonlinear approaches for two main challenges in autonomous robots, system modeling and motion planning.

When we were tackling the challenge of system modeling, we tried to analyze the problem in two ways: the first way is to completely decompose the robot into its bare actuators - the eight driving and steering motors; the other perspective was to use more interpretable vectors - the translational velocities in x and y directions, as well as angular velocity Ω. A benchmark was created with a traditional linear DC motor model, and it turns out that steering motors, experiencing less inertia and more friction, making its velocity-voltage relationship more nonlinear than driving motors. Hence, we added nonlinearity with the use of a feedforward neural network (FNN), as well as history referencing with long-short-term memory (LSTM) and temporal convolution networks (TCN), which eventually outperformed the existing linear models in terms of error and determination metrics by a significant edge. Afterwards, we shifted the scope to predicting chassis-relative speeds based on input voltages and past states. The benchmark state-space control model for swerve makes extremely inaccurate predictions due to the high nonlinearity, and it was very easy for the FNN, LSTM and TCN models to do so and learn the system, predicting much more accurately.

In the second challenge of motion profiling, we first started with a traditional linear trapezoidal velocity policy, which was satisfying in terms of its swiftness. However, with a well-designed nonlinear curriculum learning model based on a simple DC motor linear model, we were able to reduce the time by more than 30% as verified in real world deployment.

In aggregate, this series of experiments have shown the strength of nonlinear systems - especially neural networks - in modeling and controlling robots with more complex dynamics and degrees of freedom. With the formalized procedure of creating high-fidelity simulators with TCN, robotics educators and enthusiasts all over the world can now collect data from their physical robot, design an interface online, and access the robot easily online. These models can also be applied in areas like reinforcement learning to adjust feedback control parameters, train human drivers, calculate odometry more accurately, plan more complex paths and maneuvers, and many other meaningful applications.

In the next few months, our team will be formalizing these approaches and creating online tutorials that help other FRC teams create their first data-driven robot model. This is a meaningful continuation of our research, as our tutorial can help hundreds of thousands of high schoolers worldwide begin to interact with these more advanced machine learning methods, instead of sticking to over-simplified control theories that are over a hundred years old.

As FRC robots hurtle toward ever-fiercer competitions, our fusion of NN-based simulations and RL-trained policies not only shatters linear shackles but ignites a blueprint for tomorrow's autonomous warehouse navigators to planetary rovers. By dispersing these tools and methodologies for high school roboticists, we envision a ripple effect: young minds, armed with code and curiosity, redefining robotic agility and inspiring a generation to chase the nonlinear edge where precision meets possibility.

References

[1] A. Ng, “CS229: Machine Learning course notes,” Stanford University, 2019.https://cs229.stanford.edu/main_notes.pdf

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need,"https://arxiv.org/abs/1706.03762

[3] E. T. B. Lundby, A. Rasheed, and I. J. Halvorsen, “Deep active learning for nonlinear system identification,” Feb. 24, 2023.https://arxiv.org/abs/2302.12667

[4] FIRST Robotics Competition, “Swerve drive kinematics,” WPILib Docs, Feb. 16, 2024. https://docs.wpilib.org/en/stable/docs/software/kinematics-and-odometry/swerve-drive-kinematics.html

[5] FIRST Robotics Competition, “Trapezoidal profiles,” WPILib Docs, Feb. 16, 2024.https://docs.wpilib.org/en/stable/docs/software/advanced-controls/controllers/trapezoidal-profiles.html

[6] G. Mamakoukas, M. L. Castano, and X. Tan, “Local Koopman Operators for Data-Driven Control of Robotic Systems,” Nov. 2, 2019.https://www.researchgate.net/publication/336990665_Local_Koopman_Operators_for_Data-Driven_Control_of_Robotic_Systems

[7] I. Georgiev, C. Chatzikomis, and T. Volkl, “Iterative Semi-parametric Dynamics Model Learning For Autonomous Racing,” Nov. 17, 2020. https://arxiv.org/abs/2011.08750

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.https://arxiv.org/abs/1707.06347

[9] P. Ryckaert, B. Salaets, and A. S. White, “A comparison of classical, state-space and neural network control to avoid slip,” *Mechatronics*, Dec. 1, 2005.https://www.sciencedirect.com/science/article/abs/pii/S0957415805000802

[10] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780,https://www.bioinf.jku.at/publications/older/2604.pdf

[11] Y. Tian, J. Li, and J. Liu, “A study on general state model of differential drive mobile robot,” ResearchGate, 2023. [Online]. Available: https://www.researchgate.net/publication/374343815_A_Study_on_General_State_Model_of_Differential_Drive_Mobile_Robot

Acknowledgments

This research focuses on exploring ML-based simulation and motion optimization of wheeled robots, with the omnidirectional swerve drivetrain in the FIRST Robotics Competition (FRC) as the key research object. Our core goal is to explore the effectiveness of nonlinear models in robot dynamics modeling and motion planning. The successful completion of this research would not have been possible without the support and assistance of our supervisor, school, and all team members, and we express our sincere gratitude here.

First, we sincerely thank our supervisor, math teacher at Keystone Academy and Tsinghua University alumni Mr. Zhongyao Sun, for his professional guidance and self-less support throughout the research process. His rigorous academic attitude, profound expertise and rich experience were crucial for the completion of this study.

Second, we sincerely thank Mr. Taryn Loveman and Mr. Christopher Hansen, leaders of Keystone Academy's Teaching and Learning Department, for their generous support throughout this project. Mr. Loveman provided critical logistical assistance during the preparatory phase, while Mr. Hansen oversaw our safety while testing at the venue throughout the entire summer.

We extend our gratitude to our robotics mentors, Mr. Guannan He and Mr. Jacob Kouassy, for providing us with the technical assistance to create the robot's hardware prior to this research; we also thank many other Keystone Design-Technology and Innovations technicians, specifically Mr. Guangyu Chen and Ms. Liyuan Liu, who helped us set up the testing venue and props.

Lastly, we are really grateful of each other, that we shared this exciting and valuable experience. Clear division of work was crucial for building this beautiful journey, here is how we did it:

- **Mechanical Hardware Preparation**: Yuran Dai took full charge of debugging the omnidirectional swerve drivetrain and jointly completed the mechanical assembly, wiring, and tuning. He used OnShape to design the drivetrain mechanism and wiring, ensuring that the hardware met the stability requirements for long-term data collection. During the experiments, they implemented regular maintenance: lubricating gears to reduce friction-induced errors and calibrating the battery to a fixed energy baseline, which effectively minimized the impact of mechanical wear and unstable voltage on data quality.
- **Data Collection and Processing**: Tianlang Zhang was responsible for collecting physical data during the summer break. Following the experimental cyclic path scenario designed by Yuran Dai, he operated the robot to record more than 20 minutes of continuous motion data - a dataset covering various motion states critical for model generalization. After collection, Yuran Dai developed Python scripts to filter invalid samples, ultimately organizing a high-quality training data set for subsequent model development.
- **Model Technical Implementation**: Yuran Dai, having aced the CS229 Machine Learning course at Stanford Summer Sessions over the past months, designed and implemented all the codes for visualization, training and evaluating neural network models, assisting in the training and verification of FNN, LSTM, and TCN models. He also calculated and compared key evaluation metrics across models, providing quantitative data support for analyzing performance differences.
- **Thesis Writing and Logistics**: The three members evenly divided core writing tasks, analyzing model results, organizing motion planning experiment data, and deriving research conclusions, ensuring the rigor and completeness of the paper. In addition, Zekai Ma took on logistics responsibilities: negotiating with the school to secure experimental venues, coordinating equipment storage, and handling official formalities.

This research encountered challenges centered on logistic and hardware constraints that disrupted progress at different stages. For example, our original experimental venue was under construction during summer vacation, leaving us without a dedicated space for equipment storage and data collection. Zekai Ma resolved this by collaborating with the dean of engineering and assistant principal for weeks, eventually securing a temporary classroom. Another obstacle was time coordination: conflicting summer schedules and Yuran Dai's 8-week course at Stanford Summer Sessions hindered real-time collaboration, requiring asynchronous work modes. Unforeseen hardware problems also arose: water leakage from air conditioning pipes damaged the swerve chassis, halting data collection for two weeks - this also forced the team to do more data collection afterwards to account for a possibility of mechanical/electrical degradation. In the model training process, reliance on school-provided M2 Mac computers led to slow simulation speeds and frequent crashes, risking progress loss. Overcoming these challenges tested our problem-solving abilities and deepened our teamwork and resilience.

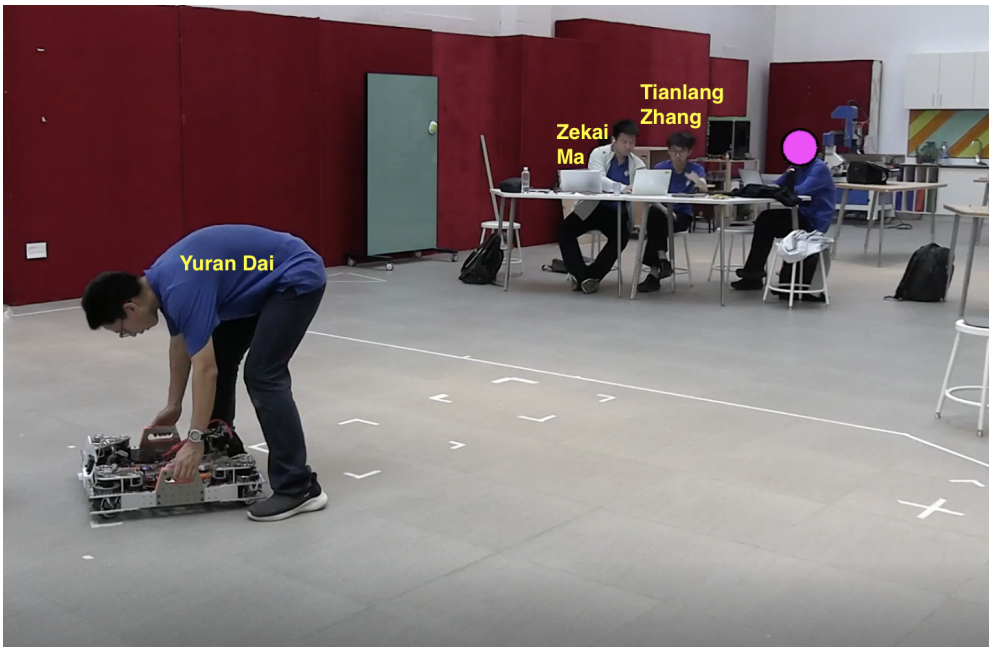

Figure 30: Three teammates preparing the robot prior to this research

This research is more than an academic practice; it is a journey of growth in scientific research, engineering implementation, and team collaboration. In the future, we hope to continue to optimize the performance of the model and explore the application of our findings in more FRC scenarios.