Matthew Papesh (mrpapesh@wpi.edu) and Emma Herman (ekherman@wpi.edu)
December 12th, 2025
CS 4342 Machine Learning
Final Project Write Up

Auto-Correcting Robot Motion Control with Neural Networks

1.0 Overview

Motion planning is an area of robotics that involves how an automated system moves with constrained acceleration and upper speed limits along a determined path. A differential-drive is a two-wheeled robot that turns and moves forward by differing the speeds of each wheel. This can be simplified into the linear and angular speed of a robot. In this assignment, a differential-drive robot was simulated driving along a curved path. The path was discretized into waypoints, with each corresponding to pre-calculated linear and angular speeds. As the robot drives, its speed is determined based on the nearest waypoint. This works as a baseline speed that changes as the robot drives; however, the robot never drives perfectly. This results in position inaccuracies as the robot roughly follows the path, but drifts away from the path nonetheless.

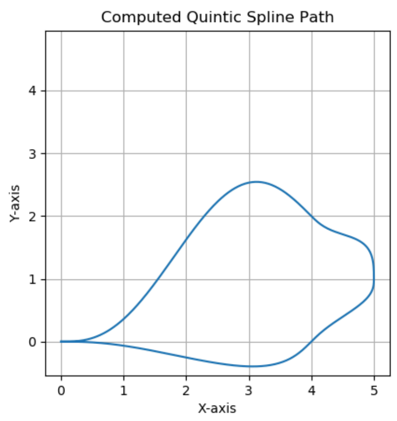
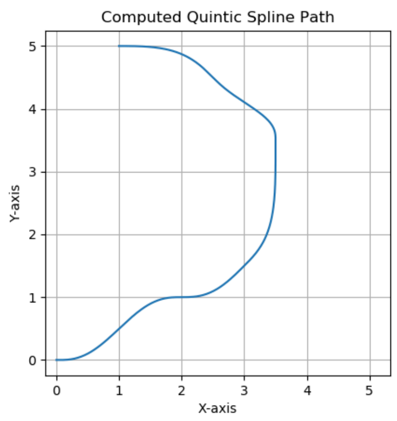
Drifting is handled by feedback controllers that adjust the baseline speed of the robot at any given time in an attempt to make the robot drift back onto the path. Feedback controllers in-take position inaccuracies between where the robot is and should be at any given time while driving. The controller then outputs the needed adjustments in linear and angular speed to correct for the inaccuracies – this is done by adjustments that encourage drifting back onto the path. As a result, the robot drives with known baseline speeds with a PID feedback controller adjusting linear and angular baseline speeds for auto-corrective positioning.

In this assignment, our group designed a feed-forward Neural Network (NN) that learns the auto-corrective behavior of PID feedback motion controllers for a robot driving along a path. To accomplish this, a robot was simulated with the Robot Operating System (ROS) Gazebo simulation environment. While driving, inaccuracies in x/y-coordinates, robot angles (heading), and actual linear and angular speed were recorded. Corresponding adjustments in linear and angular speed outputted via PID were also recorded. This data was constantly recorded until the robot finished driving along the path. Finally, the NN was trained on the input speed and position inaccuracy data to predict the needed speed adjustments to replicated the auto-corrective behavior of a PID controller.

2.0 Methodology

2.1 Data Collection

For data collection, the robot was simulated in the ROS Gazebo simulation environment. To get diversity in data that shows how a PID controller responds to different types of robot drift off path, two different paths were used: (1) a closed-loop path, the robot drives with a clockwise journey, and (2) an open-loop path that has a counter-clockwise journey. For both paths, three simulated drives occurred with slower motion, and again with faster motion. Motion was described by how quickly the robot accelerated and what its speed limits were. Collecting PID controller data this way allows the NN to learns how to correct on paths involving left and right turns at varying speeds and kinds of motion.

	
The closed-loop path. Starting from position (0,0), looking East (heading=0 degrees), the robot drives from the center of origin. The robot drives North–East and completes a clockwise journey back to the center of origin.	The open-loop path. Starting from the center of origin (0,0), looking East (heading=0 degrees), the robot drives North-East. The robot completes a counter-clockwise drive to position (2,5) as its destination.

To better define the PID controller setup, the following is the mapping from position inaccuracy to corresponding speed adjustments needed for auto-correction:

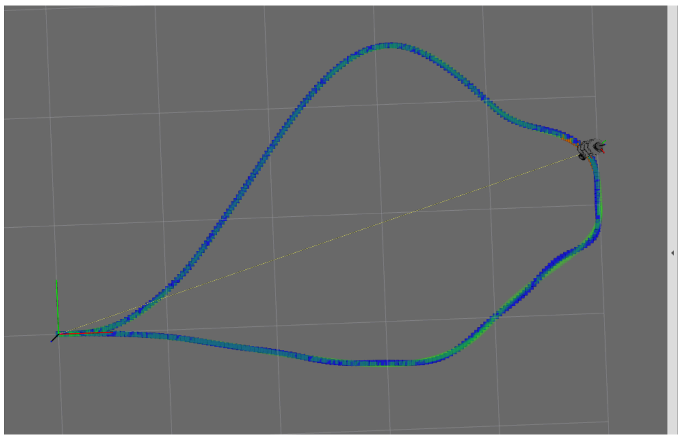
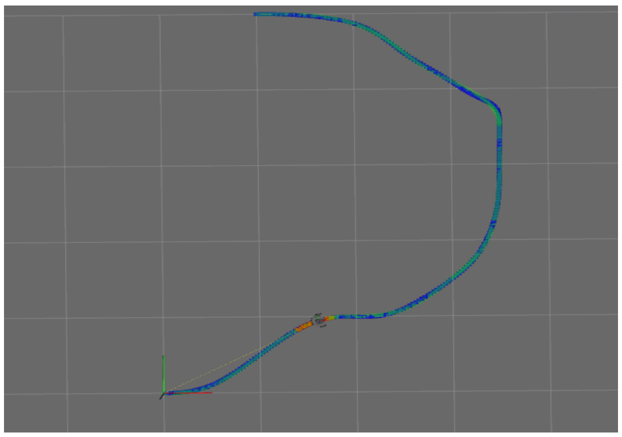
$$
\begin{array}{ccc}
VECTOR\ X & \rightarrow & VECTOR\ \hat{y} \\
(\Delta X_{ERROR},\ \Delta Y_{ERROR},\ \Delta\Theta_{ERROR},\ V_{ACTUAL},\ \omega_{ACTUAL}) & \rightarrow & (\Delta V,\ \Delta\omega)
\end{array}
$$

The controller, and our NN to that extent, considers robot position, speed, and orientation at any given time to compute linear speed adjustments (ΔV) and angular speed adjustments (Δω) – five total features mapping to a two-dimensional output. The NN is trained based on this mapping from the recorded data from the simulation.

We captured four different scenarios, where position error, robot speed, and corrective adjustments were collected. These scenarios involved driving with faster and slower motion on the open-loop and closed-loop paths – as described below:

- **Scenario #1** – Drive the **Closed-Loop** path with **slower** motion
- **Scenario #2** – Drive the **Closed-Loop** path with **faster** motion
- **Scenario #3** – Drive the **Open-Loop** path with **slower** motion
- **Scenario #4** – Drive the **Open-Loop** path with **faster** motion

Training data was collected by running a simulated drive three times for each scenario. This method collected approximately 700-1000 data points for each drive, and as many as ~3000 data points for one scenario. Recorded data from each scenario was concatenated into a single large input and output training dataset. This required no actual data annotation as the PID controller produced the "ground-truth" outputs during simulated testing.

	
Demonstration of simulating the closed-loop path for collecting input error and speeds, along with output speed corrections.	Demonstration of simulating the open-loop path for collecting input error and speeds, along with output speed corrections.

As seen in the figures above, simulations were run for the aforementioned scenarios for data collection. The robot can be seen driving along the projected green path. The blue outline is the path actually taken by the robot as it attempts to adhere to the true path.

2.2 Neural Network Architecture

As for our methods, we used a fully-connected and feedforward neural network with a five-dimensional input, 2 hidden layers, and a two-dimensional output layer. As outlined by ChatGPT, we inquired about node activation among other NN characteristics:

"Network (Strong Baseline) Input layer: 5 Hidden layer 1: 64, ReLU Hidden layer 2: 64, ReLU Output layer: 2 (linear)."

For our loss function, we used Mean Squared Error (MSE) between the predicted and PID-generated speed adjustments, and for our optimizer, we chose to use Adam with a learning rate of 0.001. Our training procedure consisted of training with 50 epochs, a batch size of 32 and the combined datasets. We evaluated by monitoring the training loss and saving the model after the final epoch was completed. Future evaluation includes comparing the predicted adjustments to the PID controller outputs and measuring tracking accuracy in the actual simulator.

Our full ChatGPT prompt is as follows:

*"Decide on a machine learning methodology: This includes the architecture, training procedure,
regularization techniques, augmentation methods, and any other design decisions. To get started, you must ask ChatGPT (or whatever other state-of-the-art AI tool you prefer) to give you a strong baseline method. For example, if you are training a pet age regressor, then ask ChatGPT
for exact instructions on how to do this (including data collection, annotation, model design, training procedures, etc.). Include your prompt and its response in your project report. It is expected that your team will then think of – and implement – some ideas that go significantly beyond ChatGPT's suggestions. Our project is a spline-path following robot that calculates linear and angular speed along interpolated points. The simulation also has adjustments for linear and angular speed at each step along the path as needed for autocorrection. This means the robot follows a path, and then computes linear and angular speed adjustments as it drives. We want to train a Neural Network (NN) on speed adjustment, where the input is (x_error, y_error, heading_error), and the output is (linear_speed_adjustment, angular_speed_adjustment) for any given point on a path. What is a grounded methodology for handling this assignment?"*

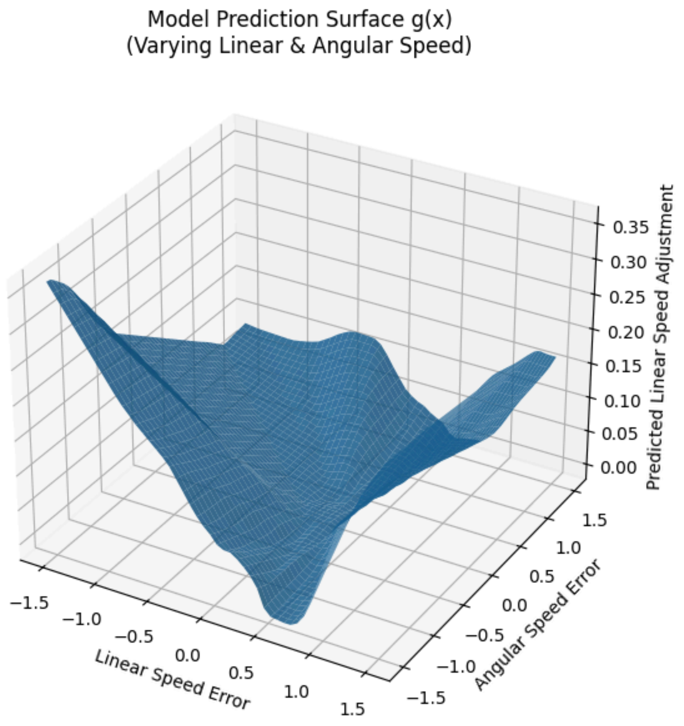
We wanted to include the exact prompt within the ChatGPT prompt in order to get an accurate output, while still including the general background of our assignment. The response will be attached at the end of this assignment.

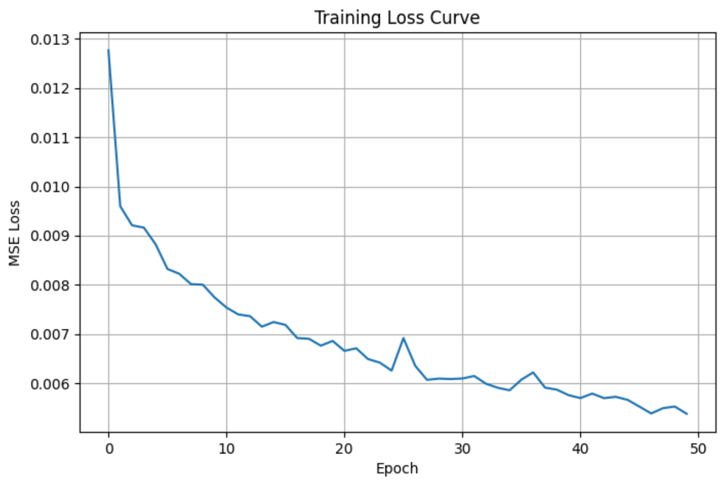
3.0 Results & Analysis:

Model/Technique	Description	MSE
MLP (2 hidden layers x 64 units, ReLU)	architecture used in training: 5 input, 64 ReLU, 64 ReLU, 2 output. Trained with 50 epochs with Adam (Ir = 0.001), batch size 32	0.005595

The final NN model consisted of 64 unit hidden layers, achieving a mean squared error of 0.00595 after 50 epochs. The loss decreased consistently across the epochs, demonstrating a stable convergence and successful learning of the PID controller's mapping. There weren't any other architectures that outperformed this configuration.

To conclude, we trained an NN by inputting data collected from a PID ROS. Normalization was a key aspect of this assignment since the data inputted needed to be stabilized in order to train smoothly and on interpretable prediction surfaces. We compared shallow models and deep models for this assignment. The shallow models tended to train faster, reached reasonable loss values, had smoother but oversimplified prediction surfaces, and overall failed to capture some of the nonlinear structure in the linear-speed adjustment surface. The deeper models required more epochs to converge, showed lower training loss, and captured curbed regions and nonlinear relationships more efficiently. Overall the deeper model performed best. Our final step is to give the ROS the ML output file to train it on that.





ChatGPT Response:

Strong Baseline ML Methodology for Predicting Speed Adjustments in a Path-Following Robot

1. Problem Definition

You want a neural network that maps robot pose tracking error → speed corrections:

Input (state error vector):

- x_{error}: lateral/longitudinal deviation from desired path
- y_{error}: perpendicular deviation from desired path
- θ_{error}: yaw/heading deviation

Output (control adjustment vector):

- Δv: linear speed adjustment
- $\Delta \omega$: angular velocity adjustment

This is effectively a **learned controller** approximating the corrective behavior your simulation or an expert controller already produces.

---

2. Data Collection Methodology

2.1. Generate Expert Demonstrations

You need pairs:
state error → desired corrective speed commands

1. Run your **existing controller** (e.g., PID or geometric bicycle controller) along many spline paths with varied:
    - curvature
    - length
    - start/end orientations
    - noise levels
    - disturbances

2. For every simulation timestep, record:
    - $(x_{\text{error}}, y_{\text{error}}, \theta_{\text{error}})$
    - $(\Delta v, \Delta\omega)$ produced by your controller

This yields a dataset of the form:

$$\mathcal{D} = \left\{ (x_e^{(i)}, y_e^{(i)}, \theta_e^{(i)}),\ (\Delta v^{(i)}, \Delta\omega^{(i)}) \right\}$$

This is identical to how “Imitation Learning” datasets are collected.

2.2. Data Diversity

Introduce variation to prevent overfitting to ideal conditions:

- Perturb paths with noise
- Add sensor noise (sampling uniformly)
- Randomize start position/heading
- Add disturbances (lateral pushes, drift, slip)
- Vary robot speed profile

This ensures the NN generalizes to real tracking error conditions.

3. Model Architecture

Because inputs and outputs are both low-dimensional and continuous, the baseline model can be a simple **fully connected MLP**:

Network (Strong Baseline)

- Input layer: 3
- Hidden layer 1: 64, ReLU
- Hidden layer 2: 64, ReLU
- Output layer: 2 (linear)

$$f_\theta(x_e, y_e, \theta_e) = (\Delta v, \Delta\omega)$$

Justification:

- Problem is smooth → MLP with 2–3 layers is more than sufficient.
- Low-dimensional → no need for CNNs, RNNs, or transformers.

---

4. Training Procedure

4.1. Loss Function

Supervised regression:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \| f_\theta(s^{(i)}) - a^{(i)} \|_2^2$$

Optionally weight outputs differently if angular corrections are more sensitive:

$$\mathcal{L} = \lambda_v(\Delta\hat{v} - \Delta v)^2 + \lambda_\omega(\Delta\hat{\omega} - \Delta\omega)^2$$

4.2. Optimizer

- Adam, learning rate 1×10−31 \times 10^{-3}1×10−3
- Batch size: 32 or 64
- Train for ~50–150 epochs

4.3. Regularization

To meet assignment requirements:

- **L2 weight decay** (1e-5 or 1e-6)
- **Early stopping** on validation loss
- **Dropout** (0.1–0.2) — not always necessary but acceptable
- **Input noise augmentation** (see next section)

---

5. Data Augmentation (Very Important for Controllers)

Unlike image augmentation, here augmentation is applied to state-error vectors:

1. **Add small Gaussian noise to input:**
   Simulates sensor variation

   $(x_e + \mathcal{N}(0, 0.01))$, etc.
2. **Randomly scale errors**
   Makes model robust to different magnitudes of deviation.
3. **Random sign flips of x/y errors**
   Ensures model symmetry and prevents directional bias.
4. **Angular wrap augmentation**

   Replace θ_{error} with $\sin\theta_e, \cos\theta_e$ if needed.

6. Evaluation Procedure

Measure regression accuracy using:

- **RMSE on validation set**
- **Controller simulation test:** plug NN into closed-loop controller and measure:
    - tracking accuracy (lateral error, heading error)
    - number of oscillations
    - overshoot
    - total path time

The NN should approximate or exceed the classic controller.

---

7. Deployment in the Simulation

At runtime:

1. Compute (xe,ye,θe)(x_e, y_e, \theta_e)(xe,ye,θe) from robot pose + desired spline point
2. Pass through NN:

   $$(\Delta v^{\backslash *}, \Delta\omega^{\backslash *}) = f_\theta(x_e, y_e, \theta_e)$$

3. Add the NN outputs to your base speed commands
4. Apply saturation limits on physical max speed/acceleration

This yields a learned residual controller.

---

8. Suggestions for Extensions Beyond ChatGPT’s Baseline

These are ideas that your group can implement to “go beyond baseline”:

- **Residual Learning:**
  NN only predicts a residual on top of PID controller → more stable.
- **Use sinθ,cosθ\sin\theta, \cos\thetasinθ,cosθ instead of raw angle**
  Removes angle discontinuity at ±π.
- **Curriculum training:**
  Train first on small errors, then gradually introduce large disturbances.
- **Online adaptation:**
  Continue training NN during simulation (like DAgger).
- **Use a physics-informed loss:**
  Penalize control actions that cause high jerk or violate robot dynamics.
- **Use separate networks for Δv and Δω**
  Helps robustness.

All of these count as “ideas that go significantly beyond ChatGPT’s suggestions.”
