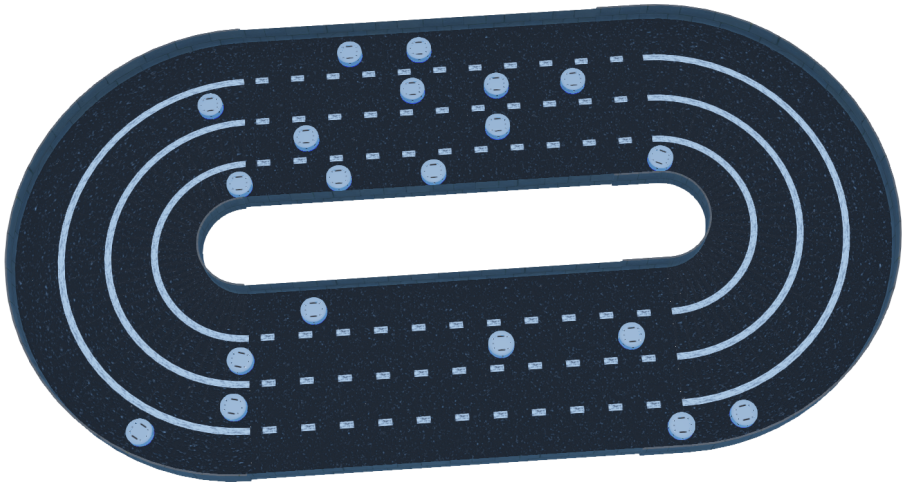
SIL program , IC-XXX, DISAL-SP111

Start: 01.07.2023
Finish: 31.08.2023

Using neural networks to model a multi-robot lane driving scenario

Oussama Gabouj



Professor: Alcherio Martinoli
Assistant: FirstName LastName

This page intentionally left blank.

Contents

Chapter 1 Introduction

In recent years, the utilization of multiple robots for various applications has gained substantial attention due to its potential to improve efficiency and productivity. One common scenario involves a group of robots navigating through designated lanes, each possessing its own unique position, finite state machine (FSM), and behavior. While simulation tools like WeBots offer an exact solution in replicating trajectories, a significant challenge emerges when aiming to enhance both scalability and computation time. For instance, a 180-second simulation can demand as much as 90 seconds for WeBots to generate all the trajectories.

This project addresses both scalability and computation time in robot lane navigation simulations by leveraging various deep machine learning approaches. Traditional simulation methods suffer from a significant computational burden. In contrast, deep learning techniques offer the potential to alleviate this issue by effectively approximating robot trajectories. However, a central concern is that conventional neural network models do not inherently account for the finite state machine (FSM) of each robot, potentially leading to less precise trajectory predictions.

The project's specific goal is to construct a neural network that can generate trajectories swiftly and efficiently while maintaining an acceptable level of precision. This involves exploring a range of deep learning techniques and methodologies to determine the most suitable approach for the given problem. The project will explore the following deep learning methods: Classic Neural Networks, Graph Neural Networks (GNNs), Reinforcement Learning and Imitation Learning.

Throughout the project, a comprehensive evaluation of each method's performance will be conducted, considering factors such as trajectory accuracy as well as computation time. Ultimately, the project seeks to determine which approach provides the best trade-off between speed and accuracy for generating scalable trajectories.

Chapter 2 Data analysis

In this section, we will focus on defining the challenges we face and identifying key variables. We will analyze the distributions of these variables and their inherent characteristics. Originating from the Webots simulation, the data's unique nature demands special attention. We aill also discuss potential preprocessing and post-processing of the raw data, going beyond description to extract its essence and align outputs with objectives.

2.1 Problem statement

Each individual robot is confined to a designated lane and retains the capability to maneuver between these lanes based on its distinctive behavior. The dataset compiled from these navigation instances comprises crucial positional information: x, y, and θ coordinates of each robot. These coordinates, illustrated in Figure 2.1, serve as vital components for understanding the precise movements of the robots within the lane-based context. This computational challenge hinges on accurately predicting the trajectories of robots as they navigate.

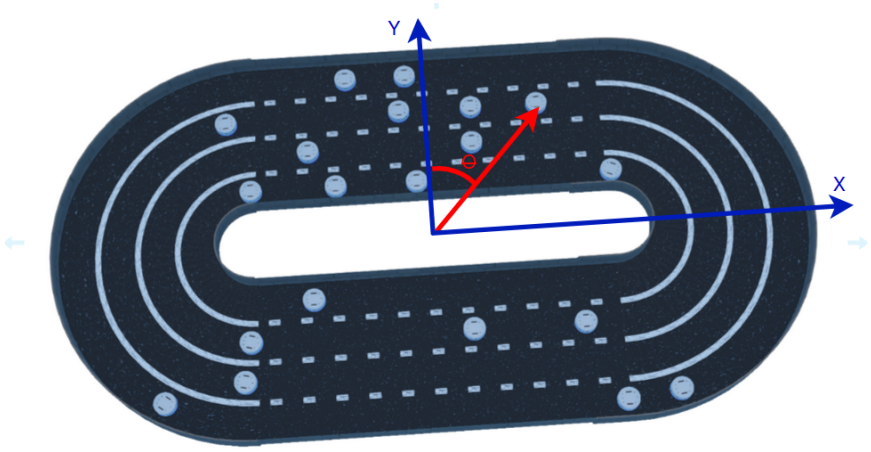


Figure 2.1: Robot simulation environment

2.2 WeBots simulation data analysis

The dataset contains comprehensive robot information, encompassing key variables such as position and orientation. The table 2.2 represents the variation of these different variables based on the simulation. It includes the variables x, y, and θ, along with their corresponding minimum and maximum values, as well as the range, which is the difference between the maximum and minimum values for each variable. There are 3712*20 samples in our dataset.

Variable	min	max	range
x	-2.119	2.369	4.489
y	-1.1267	1.13	2.25674
θ	- π	π	2 π

Table 2.1: Robots' position range

The figure below illustrates the data distribution: the x position seems to be balanced, while the y and θ variables exhibit imbalance. It is imperative to consider these imbalanced conditions during the training of our models.

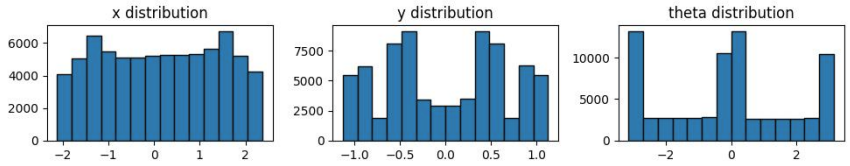


Figure 2.2: Data distribution

2.3 Robot speed data analysis

Simulating robot trajectories involves a critical consideration of the distribution of speed data, given that speed fundamentally influences the predictions generated by all models. Accurately replicating trajectories hinges on the neural network's ability to account for the variations in speed that robots exhibit during their movements. Analyzing the speed distribution is essential to ensure that the generated trajectories align with real-world scenarios, enhancing the network's capacity to generalize across different movement speeds and contribute to more reliable predictions. The table 2.3 represents the variation of these different velocities:

The figure 2.3 illustrates the data distribution. Knowing that the output delta_x, delta_y, delta_teta are not in the same units, this can lead to several problems, including difficulties in interpretation and evaluation of the

Variable	min	max	range
delta_x	- 0.0127	0.0127	0.025424
delta_y	- 0.0122	0.0122	0.024
delta_teta	0.0997	0	0.0997

Table 2.2: Robots' velocity range

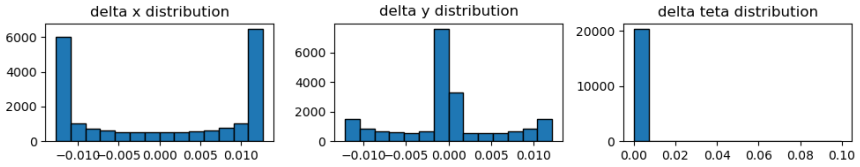


Figure 2.3: Data distribution

model's performance. We need aslo to consider these imbalanced conditions during the training of our models.

2.4 Dataset challenges and potential solutions

In the subsequent sections 2.2 and 2.3, it is imperative to address the ensuing challenges:

- Difficulty in interpretation: When the predicted values are in different units, it becomes challenging to interpret the magnitude of the predictions.
- Evaluation metrics: If the target variable and predicted values are not in the same units, standard evaluation metrics such as mean squared error (MSE) or mean absolute error (MAE) may not provide a meaningful measure of the model's performance.
- Unbalanced data: The distribution of classes in the dataset is significantly skewed. Trained on unbalanced data can become biased towards the majority class. Since the model's objective is usually to minimize overall error, it may focus more on accurately predicting the majority class, while paying less attention to the minority class. We can clearly see that δ_θ is not verry significant for $\delta_t = 32$ ms. Moreover the δ_x and δ_y are not balanced.

To address the potential problems caused by unbalanced data, data preprocessing plays a crucial role. The following steps need to be performed:

- Normalizing the data: we scale both the target variable and predicted values to a common scale. Since the data distribution is not normal using min-max scaling is better than z-score normalization.

- Balance the data: This step involves down sampling the majority class to create a more balanced dataset by randomly selecting a subset of examples from the majority class to match the minority class size.

We also need to consider a post processing procedure in order to convert the predicted values into the desired units.

Chapter 3 Deep Neural network

3.1 Neural network training

In our study, we have implemented a variety of different models, which can be classified into two groups: graph neural networks, including GAT and GCN, and non-graph neural networks, featuring well-known architectures like MLP, CNN, and LSTM. In the following sections, we will unveil the specifics of each of these neural network configurations, providing details about the outcomes they have produced.

All the models are trained in an open loop configuration, as illustrated in Figure 3.1. For each randomly chosen position, the model predicts the corresponding delta position. This allows us to anticipate the next position $\vec{x}(t+1)$ by adding the model's prediction to the current position.

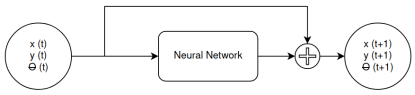


Figure 3.1: Open loop simulation

Subsequently, the models are subjected to closed loop testing, as depicted in Figure 3.2. In this scenario, the neural network is tasked with simulating the robot's positions from a given initial position. The network's output, concatenated with the input, yields the next position $\vec{x}(t+1)$. This calculated position then becomes the subsequent input for the neural network, and consequently forecast the position at the subsequent time step $\vec{x}(t+1)$.

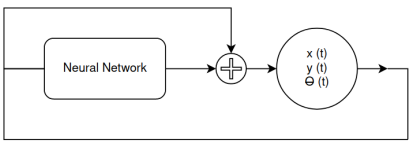


Figure 3.2: Closed loop simulation

3.2 Non Graph neural network

In this section,we will explore diverse conventional machine learning approaches. This exploration aims to gauge their performance within this context. We implemented fundamental neural network architectures, such as the multi-layer perceptron (MLP), convolutional neural network (CNN), and one example of recurrent neural network: Long short term memory (LSTM). These sequential investigations will provide valuable insights into the effectiveness of these models within the given framework.

MLP We started with MLP network because it is very simple to implement. In our implementation it takes around 50 seconds for training. However, it treats each input independently and do not consider the context or relationships between inputs. We approach each node, symbolizing a robot, as an elementary input entity. For instance, if our dataset comprises 3712 graphs, each containing 20 robots, we're working with a grand total of 3712 * 20 data samples. The model's input 3 * n and output is 3 * m where 'n' is the number of prior positions that the robots have assumed, and 'm' indicates the number of subsequent positions the model aims to predict. This introduction of hyperparameters serves a crucial purpose: preventing the model from solely focusing on immediate predictions. Instead, it encourages the model to navigate through the entire simulation horizon. We adopted a cross-validation approach to fine-tune our model's performance: We systematically adjusted the 'n' parameter from 1 to 10, and 'm' from 1 to 5.

CNN It's essential to highlight that the MLP network treats each input in isolation and doesn't factor in context or relationships between inputs. That is why we transitioned into the CNN model. In this setup, we consider each input as a simulation step represented by a 2D matrix containing data from 20 robots. Each robot was equipped with 3 * n features, much like the structure in the MLP network. Correspondingly, the model's output took the form of a vector: 20 * 3 * m. Here, 'n' represents the count of preceding positions taken by the robots, and 'm' denotes the number of anticipated future positions. The incorporation of hyperparameters proved vital in steering the model away from exclusively focusing on short-term predictions. Instead, it guided the model to extend its predictions across the entirety of the simulation's horizon. As part of our model refinement, we introduced a cross-validation approach, systematically adjusting the 'n' parameter from 1 to 10, and 'm' from 1 to 5. Within this architecture, we employed two convolutional layers and followed them up with two fully connected layers. This design aimed to harness the spatial relationships and patterns present in the data, allowing for a more nuanced understanding of robot movements throughout the simulation.

LSTM We maintained a similar architecture as that of the MLP network. However, we opted for the LSTM (Long Short-Term Memory) model due to its recurrent nature. RNNs, like LSTM, are particularly advantageous when dealing with data that exhibits dynamic changes in relation to previous steps. Similar to the MLP approach, we considered each input as a 2D matrix portraying a simulation step, containing information from 20 robots. Each of these robots retained 3 * n features, paralleling our earlier setups. Consequently, the model's output maintained its form as a vector: 20 * 3 * m. By employing the LSTM architecture, we aimed to effectively capture the temporal dependencies and evolving patterns in the data. The inherent recurrent nature of LSTM units enables them to retain and utilize information from previous time steps, rendering them particularly well-suited for scenarios where such context matters.

Results The resulting outcomes are visually depicted in the figures 3.3 and 3.4.

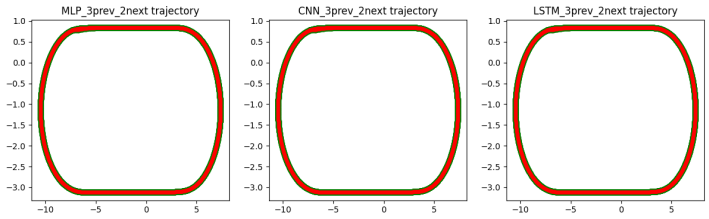


Figure 3.3: MLP, CNN and LSTM open loop simulation

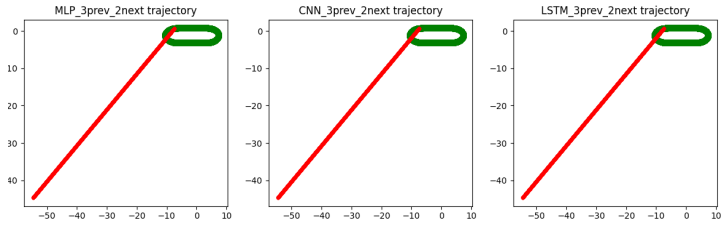


Figure 3.4: MLP, CNN and LSTM closed loop simulation

Initial findings showcas the model's proficiency in predicting outcomes for the open loop simulation. However, when we apply this neural network to the closed loop simulation, the results deviate from our intended objectives. The accumulation of residual errors became evident, leading to a situation where subsequent inputs were progressively pushed farther away from the model's latent space. Consequently, this phenomenon resulted in undefined robotic behaviors within the simulation.

3.3 Graph neural network GNN

GNNs can capture both local and global context by aggregating and propagating information through the graph structure. They can learn representations that capture the dependencies and interactions between nodes, considering both their attributes and relationships with neighboring nodes as illutrated in [1]. We are going to use the GCN (Graph Convolutional Network) and GAT (Graph Attention Network) models.

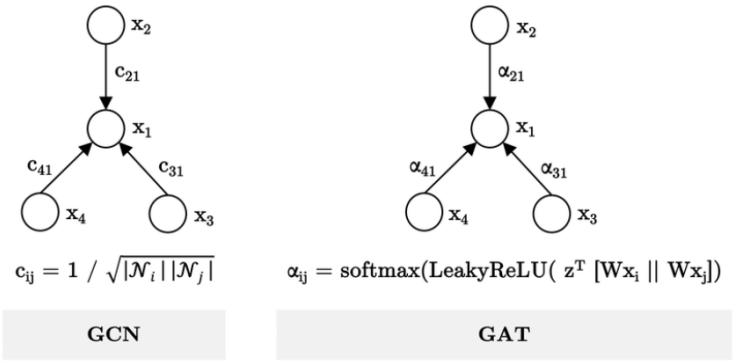


Figure 3.5: GAT and GCN architectures

As we can see from the figure 3.5, the GCN utilizes a neighborhood aggregation scheme (local aggregation), where each node aggregates information from its neighboring nodes by taking the average or sum of their features. However, the GAT uses the attention mechanism. in order to allow each node to attend to different neighbors with varying weights, capturing more fine-grained relationships in the graph. We can say that $GCN \subseteq GAT$

GCN It is a convolutional neural network (CNN) adapted for graph-structured data. This architecture has several advantages comparing with the non graphs models: Ability to capture graph structure: and the message passing and neighborhood aggregation. In fact, GNNs employ message passing schemes to propagate information across the graph. This enables GNNs to aggregate information from neighboring nodes.

GAT The GAT model extends the idea of attention mechanisms to graphs, allowing each node to attend to different nodes with different weights. It employs self-attention to determine the importance of each neighbor during the aggregation process. It has more weights (more complex than GCN) beacuse the attention weights are learned through a shared learnable attention mechanism, allowing the model to adaptively assign importance to neighboring nodes. We implemented the model as follows:

- 2 GAT layers with 2 heads each and 2 fully connected layer with Relu as activation function
- 200 epochs and an adaptive leaning rate $lr = 0.1 \cdot \frac{t}{40}$ where t in number of iterations

The provided figures 3.6 and 3.7 demonstrate the performance of the GAT model. The figure 3.6 depicts the model's predictions for a single time step concerning a specific position. The model is designed and trained for this sequential prediction task. The results indicate favorable performance.

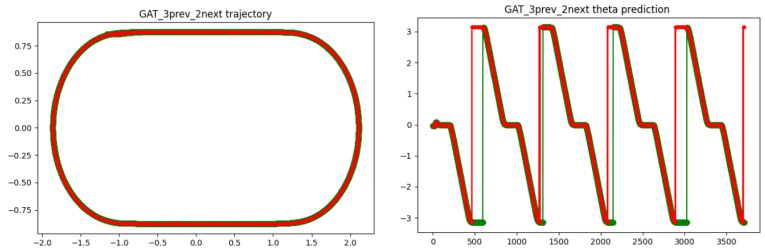


Figure 3.6: Open loop simulation

However, in the figure 3.7, the model is tested in a closed loop setup. Here, the predicted change in position (delta position, t) is added to the current position (pos(t)) to obtain the new position (pos(t+1)), which is then fed back as input to the model. Unfortunately, the model's performance in this closed loop scenario is suboptimal due to the accumulation of residual errors over time.

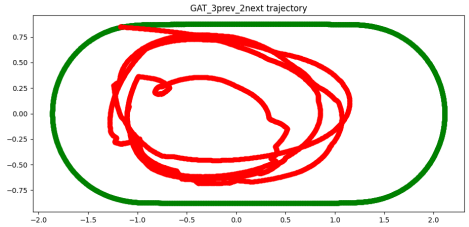


Figure 3.7: Closed loop

The GCN model also exhibits promising results when employed in an open loop configuration, yet its performance deteriorates significantly in a closed loop setting and it is worse than GAT model.

3.4 Under performance analysis

There are several plausible reasons for the underperformance of all the deep learning models:

- The primary purpose of these models isn't centered on simulating robot paths: All the previous models are initially designed for tasks other than simulating robot navigation paths. Their inherent architecture might not be optimally suited for this specific simulation task, leading to compromised results.
- Accumulated residual errors leading to untrained spaces: The accumulation of residual errors over successive time steps can force the model into regions of the latent space it wasn't explicitly trained on.
- Dynamic nature of the graph: The underlying graph in this scenario isn't static; it changes as the robot navigates, altering its neighbors. This dynamic nature poses a challenge as the robot's interactions with its environment evolve over time. This dynamic graph structure necessitates frequent updates, and the time required to update the graph for each iteration should be factored into the overall computation time. For instance, while the neural network's simulation time might be relatively short (around 5 seconds), the additional 40 to 50 seconds needed for graph updates can lead to longer and less efficient computations. The graph neural network simulation requires approximately 60 seconds, whereas the WeBots simulation takes 90 seconds for a 3-minute simulation. Despite the model's trajectory prediction deteriorating significantly, the model's speed remains unchanged, rendering it non-scalable and misaligned with our objectives.

Considering these limitations, it's essential to reassess the chosen models and explore alternatives that might better align with the task's requirements. Other models or approaches could offer more accurate and efficient solutions, especially when considering the dynamic and evolving nature of the robot's interactions within its environment.

Chapter 4 Deep Reinforcement Learning

The models we've discussed so far lack the training needed to effectively serve our study's primary purpose. These models, which predict single time-step movements, reveal limitations when used in a closed-loop simulation. To overcome this challenge, a promising alternative lies in employing reinforcement learning. This alternative holds promise for our project due to its ability to address the challenges posed by the accumulation of errors in closed-loop simulations. Unlike the traditional models focused on single time-step predictions, reinforcement learning allows the robot to learn and adapt over time through trial and error. By iteratively interacting with the environment and receiving feedback in the form of rewards, the robot can gradually refine its decision-making process. This iterative learning approach enables the robot to correct for residual errors and adjust its actions dynamically, making it well-suited for our navigation-focused study as highlighted in [2]

4.1 Environment

In the context of our study, reinforcement learning operates by having the robot interact with its environment, making decisions based on its observations and receiving rewards that guide its behavior. The environment encompasses an action space, representing the set of possible actions the robot can take, and an observation space, encapsulating the information the robot perceives from its surroundings. This setup allows the robot to learn to navigate while gradually honing its decision-making process for enhanced performance.

Observation space We set the Observation Space as follows: it is a 2D dimension containing the robot's recent positions, represented as vectors of 3D coordinates [x, y, theta]. These vectors encapsulate the robot's location (x, y) and orientation θ. To ensure consistent training, all variables within this space are normalized to fall within the range [0, 1]. This normalization process enhances the network's ability to learn effectively, accommodating

varying data scales and facilitating convergence during training.

Action space This space plays a crucial role in dictating the robot's possible maneuvers. By analyzing the normalized feature data, we discern distinct ranges for each action component. Specifically, the change in the x-direction δx spans [-0.015, 0.015], while the y-direction δy covers [-0.03, 0.03], and the change in orientation $\delta\theta$ encompasses [-0.5, 0.5]. These defined ranges shape the action space as a three-dimensional vector, with each component varying independently within its respective boundary.

The figure 4.1 illustrates a visual representation of the environment, succinctly defining both the action space, encompassing the actions that a given agent can undertake, and the observation space, encapsulating the information the agent can gather from its surroundings

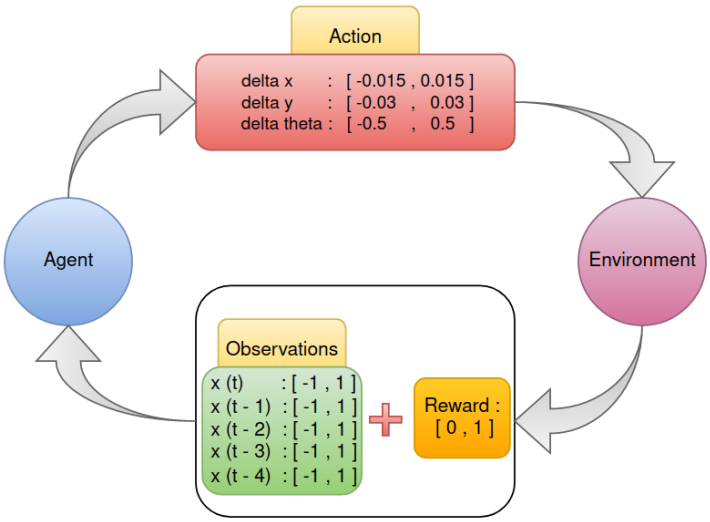


Figure 4.1: Reinforcement Learning implmentation

4.2 Agent

We have narrowed down our choices to algorithms that are compatible with environments where actions are continuous and offer effective solutions:

- PPO (Proximal Policy Optimization): PPO is a policy optimization algorithm that efficiently updates the policy in small steps, ensuring that the policy doesn't stray too far from its original form. This stability is achieved by enforcing a "proximal" limit on policy updates,

preventing drastic changes that might lead to performance fluctuations.

- A2C (Advantage Actor-Critic): A2C combines actor-critic architecture with an advantage function, allowing the agent to understand the value of taking certain actions relative to others. The actor chooses actions, and the critic evaluates their quality, aiding in more informed decision-making.

Moreover, we've opted for a Multi-Layer Perceptron (MLP) policy. Unlike Convolutional Neural Networks (CNNs) that excel with image data, MLPs are better suited for non-image data. MLPs consist of densely connected layers, making them effective for handling structured and continuous data like the observations and actions in our environment.

4.3 Reward

The rewards serve as the guiding signals that direct an agent's behavior towards achieving desired goals. They act as a feedback mechanism. In our scenario, the robot's behavior is evaluated based on a series of specific rewards, each designed to measure different aspects of its performance. These rewards are assigned values between 0 and 1, reflecting the extent to which the robot aligns with desired behaviors. The overall performance of the robot is then determined by taking the average of these individual rewards.

Lane reward : The reward function decribed by 4.1 plays a big role in how we judge the robot's performance in our simulations by keeping an eye on whether the robot behaves well in the lane or not. If the robot stays in the same lane as the simulation and doesn't get out of the lane, it gets +1. Otherwise, the robot gets -1.

$$\text{reward 1} = \begin{cases} 1 & \text{if } \|\vec{\mathbf{x}}_{t-i\ \text{pred}} - \vec{\mathbf{x}}_{t-i\ \text{sim}}\| < \frac{1}{2} \cdot \text{w} \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

Where w is the lane width.

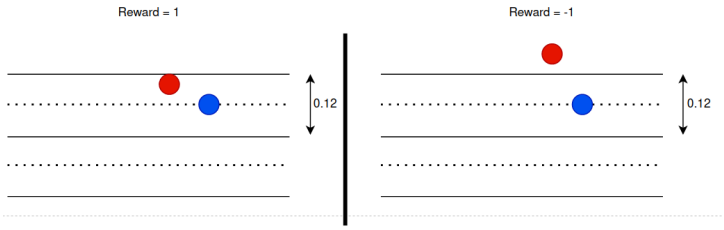


Figure 4.2: On lane reward

Trajectroy deviation : This reward,given by 4.2, has a simple goal: to keep the robot on track. By comparing where the robot is supposed to be (its predicted position, $\vec{x}$pred) with where it actually is in the simulation ($\vec{x}$sim), and using a positive real hyper parameter λ, this reward pushes the robot to stay close to the desired trajectory. The closer it sticks to the path, the higher the reward is. If the robot strays, the reward drops, nudging it back toward the right course. In Figure 4.3, you can see a visual example of how this reward changes as the robot's position shifts from the intended trajectory.

$$\text{reward } 2 = e^{-\lambda \cdot \|\vec{x}_{\text{pred}} - \vec{x}_{\text{sim}}\|} \tag{4.2}$$

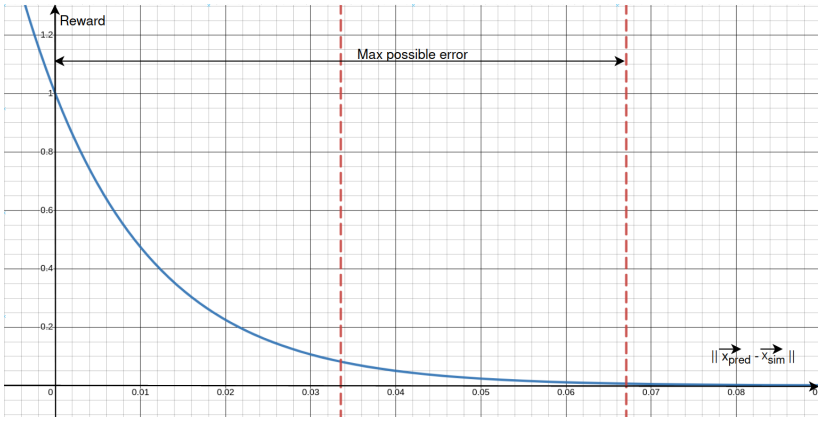


Figure 4.3: Trajectory deviation reward

Angle deviation reward : This equation 4.3, focuses on the robot's alignment with the road's curve, by checking if the robot's heading matches the curve it's on. If the cosine of the angle between the predicted heading ($\vec{x}_{\text{pred}}$) and the curve's tangent is above a certain threshold angle, the robot gets +1 reward (approval for good alignment). If not, the reward stays at 0, indicating that the robot needs to adjust its direction. The figure 4.4 shows how the alignment reward changes as the robot's heading matches or deviates from the road's direction.

$$\text{reward } 3 = \begin{cases} 1 & \text{if } \cos\left(\vec{x}_{pred}, \vec{x}_{sim}\right) > \text{threshold\_angle} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

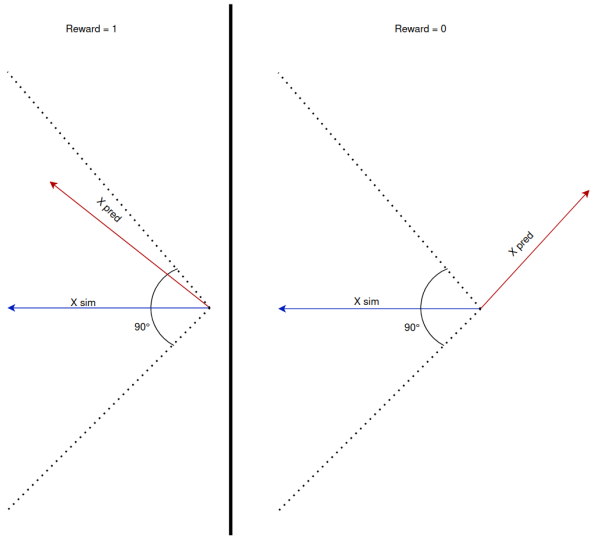


Figure 4.4: Angle deviation reward

Smoothness reward : This reward function aims to minimize the disparities in velocity changes between the robot's predicted and simulated paths, resulting in smoother and more predictable motion. The figure 4.6 ilustrates the area that we want to minimize. Just like the previous rewards, this one also operates on the concept of exponential decay. This reward computes the absolute differences in velocity changes for the robot's predicted trajectory ($\vec{\Delta}\mathbf{x}i$ pred) and the simulated trajectory ($\vec{\Delta}\mathbf{x}i$ sim) over a span of 5 time steps. These differences are then averaged, multiplied by a positive real parameter λ, and passed through an exponential function as shown in the equation 4.4. The resulting value is used as the reward.

$$\text{reward } 4 = e^{-\lambda \cdot \frac{1}{5} \sum_{i=0}^{5} \| \vec{\Delta}\mathbf{x}_{i \text{ pred}} - \vec{\Delta}\mathbf{x}_{i \text{ sim}} \|} \tag{4.4}$$

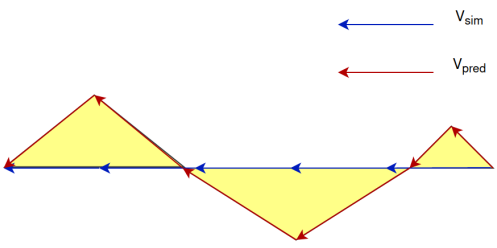


Figure 4.5: Smoothness reward: Area to minimize

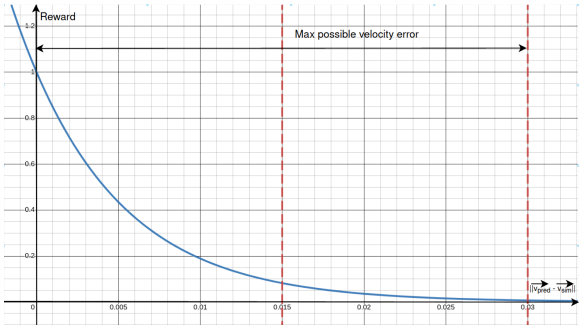


Figure 4.6: Smoothness reward

Simulation length reward : The purpose of this reward is to encourage the robot to complete the entire simulation rather than just focusing on specific segments. This can be especially useful when the robot develops strategies to follow only parts of the lane and then end the simulation early. The reward function works by considering the ratio of the current episode length to the total length of the simulation. The equation exponentiates this ratio after multiplying it by a positive real parameter λ, and then subtracts the result from 1. In simpler terms, if the robot manages to complete the entire simulation (episode length equals total simulation length), this reward becomes 1, indicating successful completion. As the robot tends to end the simulation early or deviate from the desired lane, this reward decreases. This reward encourages the robot to stay committed to the full simulation duration and stay on track throughout the entire scenario so that the robot doesn't take shortcuts or neglect parts of the task.

$$\text{reward } 5 = 1 - exp^{-\lambda \cdot \frac{\text{episode length}}{\text{total simulation length}}} \tag{4.5}$$

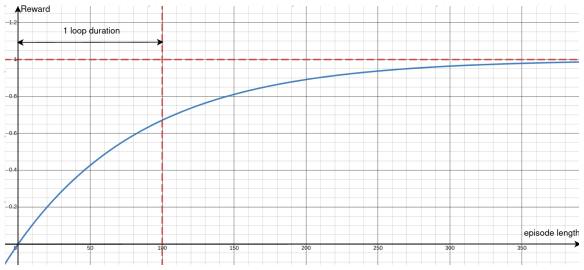


Figure 4.7: Simulation length reward

Theta reward The focus of this reward function is to guide the robot in maintaining a precise alignment with the path's center direction. The reward

operates by quantifying the orientation error between the robot's predicted trajectory ($\theta_{i\ \text{pred}}$) and the simulated trajectory ($\theta_{i\ \text{sim}}$) over a span of 5 time steps. It does so by computing the absolute difference between these angles, considering both the direct difference and the wrapped difference due to circular orientation and then computes the minimum between these two angle differences across the time steps. This average error is used as an exponent for the base of the exponential function. The idea here is to strongly penalize any orientation errors, encouraging the robot to closely match the path's center orientation.

$$\text{reward } 6 = e^{-\frac{1}{5}\sum_{i=1}^{5} \min\left(\left|\theta_{i\ \text{pred}} - \theta_{i\ \text{sim}}\right|, \left|\theta_{i\ \text{pred}} - \theta_{i\ \text{sim}} - 1\right|\right)} \tag{4.6}$$

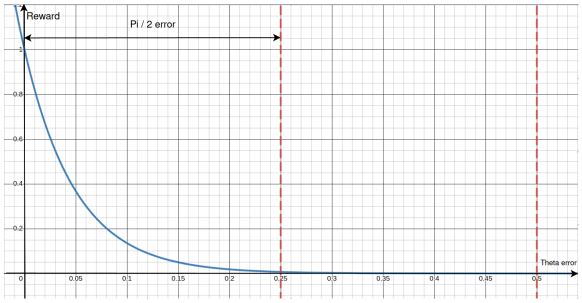


Figure 4.8: Theta reward

4.4 Reinforcement learning simulation result

In Figure 4.9, the simulation in a closed loop configuration is demonstrated using a reinforcement learning neural network. The green trajectory represents the desired path generated by the WeBots that the robot should ideally follow, while the red trajectory depicts the output from the neural network.

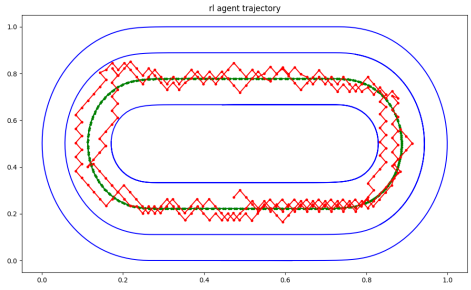


Figure 4.9: Reinforcement learning trajectory simulation using PPO policy

Although the results show improvement compared to previous models, there are still oscillations and challenges in smoothly navigating turns. This behavior might stem from the neural network's training duration of only 1 million epochs, which consumed approximately 3 hours of training time. It's worth noting that with an extended training period, more favorable outcomes could potentially be achieved by mitigating the observed oscillations and enhancing the network's ability to navigate turns effectively.

Chapter 5 Imitation Learning

Our problem involves the intricate navigation of a succession of decisions, each intricately entwined with the dynamic observations of the evolving environment, thereby encapsulating the complexities intrinsic to sequential decision-making. The performance evaluation of our system transpires via the same reward function of the reinforcement learning.

In this approach, we suppose the existence of an expert driver. Imitation learning entails enabling a novice to acquire the art of emulating this expert prowess, extracting insights from observations made at specifically designated intervals. This approach, akin to a kinship of supervised learning, entails the tuition of a policy π through the utilization of an encompassing dataset D which facilitates navigating the trajectory towards adept imitation through iterative exposure to carefully curated observations.

5.1 Behaviour cloning

Behavior Cloning is the a fundamental approach in imitation learning. This method leverages expert-generated data, comprising a collection of demonstrations encompassing observations paired with corresponding actions and resultant rewards. In our specific scenario, we engage three distinct experts:

- WeBots Data Expert: This expert is perfect. The algorithm endeavors to mimic the capabilities of this expert. However, the model is confined to learning solely from impeccable trajectories, which makes it unable to rectify its own mistakes due to the absence of exposure to exploration within its surroundings.

- Reinforcement Learning Agent Expert: While this expert may not be flawless, as discussed earlier, it emerges as a prominent contender. Despite certain limitations, its performance can be deemed commendable, considering the constraints of the environment.

- Hybrid Expert (Combination of 1 and 2): We combine both the perfect demo set and the RL agent. The former exhibits flawless scores (288/288), while the latter, though not impeccable, offers a more dynamic approach to learning.

The results depicted in Figure 5.1 illustrates the outcomes of three models: three behaviour cloning models and the hybrid models. As anticipated, the hybrid models and the RL models outperform the WeBots expert. This outcome is coherent with expectations, given that the models possess the capability to learn from their mistakes and explore the surrounding space more effectively.

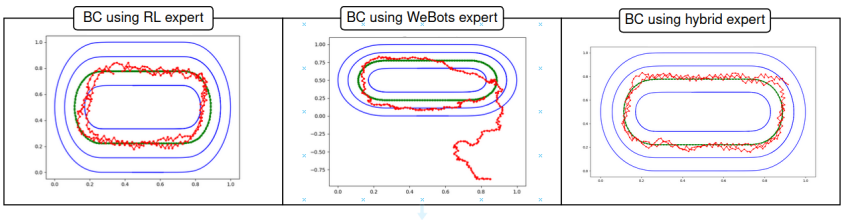


Figure 5.1: Behaviour cloning trajectory simulation

However, this Supervised imitation learning algorithm using only WeBots Data isn't without its challenges, particularly the issue of compounding errors. When the novice model, deviates from an expert trajectory, rectification becomes an arduous task. The critical concern lies in the fact that when a Behavior Cloning-trained policy falters in predicting at time step "t," it can inadvertently perpetuate errors over subsequent steps. This predicament arises due to the model's lack of exposure to unseen states during training, hindering its ability to navigate unfamiliar circumstances effectively.

To surmount this intricate challenge, a potential solution lies in using the power of the hybrid expert or the Dagger algorithm (see section 5.2), both of which warrant thorough exploration in the upcoming section. These approaches offer novel strategies to enhance the learning process, mitigating the issue of compounding errors and equipping the model with the versatility to navigate through a wider array of circumstances adeptly.

5.2 Behaviour cloning with Dataset Aggregator

Unlike Behavior Cloning, DAgger refines its policy by iteratively correcting its missteps, culminating in a more robust and adeptly navigational policy. by enabling the model to learn from its own errors. This entails training an agent using the mistakes it encounters during its operation. This process unfolds by first testing agent policy and recording the states it navigates through. Subsequently, a new agent, is trained using the recorded states, building on the experiences of its predecessor agent. The figure 5.2 shows the result.

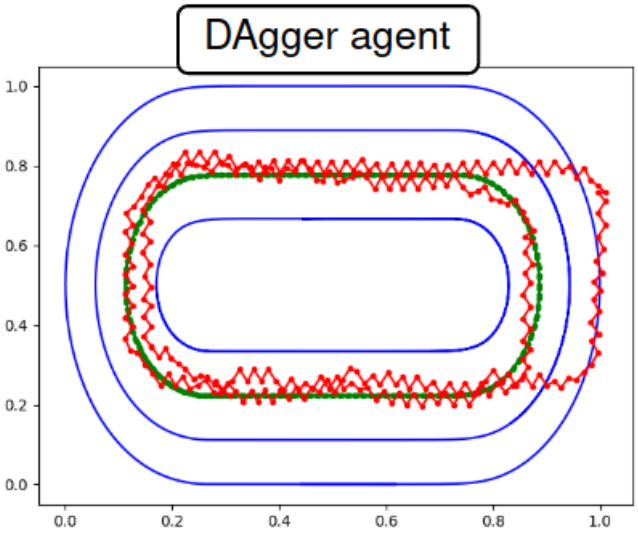


Figure 5.2: DAgger trajectory simulation

5.3 Imitation learning models comparison

In this section, a comprehensive analysis of all the trained models using imitation and reinforcement learning is presented in the table below. The assessment takes into account key metrics such as the mean reward, and training time. This comparison offers valuable insights into the performance and efficiency of each model.

Models	mean reward	training time (min)
RL agent	80 / 288	~ 200
BC WeBots expert	36 / 288	~ 75
BC RL expert	100 / 288	~ 76
BC hybrid expert	110 / 288	~ 126
IL (DAgger)	50 / 288	~ 136

From the provided table, it becomes evident that the BC hybrid model stands out as the most favorable option, boasting the highest mean reward. However, it's worth noting that achieving this performance requires the integration of an RL agent, resulting in a combined training time of 326 minutes (RL agent training time of 200 minutes plus BC model training time of 126 minutes).

Chapter 6 Models comparison

When assessing the performance of various models in both open-loop and closed-loop scenarios, distinct trends become apparent. In the open-loop format, all models exhibit remarkable accuracy in trajectory prediction. Deep neural networks particularly shine, outperforming reinforcement learning and imitation learning while also requiring less training time.

Transitioning to closed-loop scenarios, a divergence emerges. Reinforcement learning and imitation learning models exhibit the most promising results. These models generate trajectories that closely mimic the simulated ones within a remarkably short timeframe; for instance, they can complete a 3-minute experiment simulation in just approximately 7 seconds. However, it's worth noting that the training process for these closed-loop models can be more intricate and time-consuming.

The comparative assessment is synthesized in the table below, offering a comprehensive overview of the performance of each implemented model across open and closed-loop scenarios. This analysis serves as a valuable guide for determining the most suitable model for different real-world applications, striking a balance between accuracy, training time, and the ability to account for complex robot behaviors.

Models	open_loop simulation	closed loop simulation	training time (min)	Simulation time (s)
WeBots	✓	✓	-	~ 90
MLP	✓	×	~ 1	~ 2
CNN	✓	×	~ 3	~ 3
LSTM	✓	×	~ 2	~ 4
GNN: GCN	✓	×	~ 2	~ 3
GNN: GAT	✓	×	~ 3	~ 5
RL (PPO)	✓	✓	~ 200	~ 7
BC WeBots	✓	✓	~ 75	~ 7
BC RL	✓	✓	~ 76	~ 7
BC hybrid	✓	✓	~ 136	~ 7
IL (DAgger)	✓	✓	~ 126	~ 7

Moving forward, our focus will primarily be on models suitable for closed-loop simulations specifically, reinforcement learning and imitation learning algorithms. In the context of these models, we can achieve considerable efficiency gains. For instance, we can simulate 20 robots for 180 seconds simulation length and it takes only 7 seconds to compute the whole simulation. This starkly contrasts with WeBots, which requires approximately 90 seconds for the same task. This achievement signifies that we have successfully reached our ultimate objective: reducing computation time, achieving scalability, and preserving trajectory fidelity as closely as possible, as depicted in Figure 5.1

Chapter 7 Conclusion

Throughout this study, we conducted an in-depth exploration of various models to address the challenges posed by robot trajectory prediction. Our findings underscore the effectiveness of reinforcement learning (RL) and imitation learning for such applications. RL, despite its lengthier training times, excels in complex scenarios. we hold the belief that it will outperform other approaches in such intricate situations. On the other hand, imitation learning proves to be efficient in terms of training time, but demands expert guidance and operates within the confines of the patterns it has been trained to replicate.

It is worth emphasizing that while the RL model requires fine-tuning, we believe its results can be optimized for more intricate simulations involving robots with dynamic behaviors and varying lanes. This project presented its share of difficulties, from delving into lectures and online resources to the implementation of diverse models. Transitioning from conventional neural networks to graph neural networks, RL, and imitation learning introduced complexities, but the project encompassed a comprehensive spectrum of machine learning aspects while demonstrating the versatility and adaptability of machine learning techniques in solving complex problems.

Bibliography

[1] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *ICLR*, 2018.

[2] D. Kamran1, J. Zhu1, and M. Lauer1, "Learning path tracking for real car-like mobile robots from simulation," *IEEEXplore*, 2019.

[3] D. Garg, "Learning to imitate." http://ai.stanford.edu/blog/learning-to-imitate.

[4] Divyansh, "A brief overview of imitation learning." https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c.

[5] Datasolut, "Reinforcement learning: Wenn ki auf belohnungen reagiert." https://datasolut.com/reinforcement-learning.

[6] Distill, "A gentle introduction to graph neural networks." https://distill.pub/2021/gnn-intro.

[7] Datacamp, "A comprehensive introduction to graph neural networks (gnns)." https://www.datacamp.com/tutorial/comprehensive-introduction-graph-neural-networks-gnns-tutorial.

[8] OpenAI, "Stable Baselines3." https://github.com/DLR-RM/stable-baselines3.

[9] pyg-team, "pytorch geometric." https://github.com/pyg-team/pytorch$_g eometric$.

[10] M. Labonne, "Graph attention networks: Self-attention explained." https://towardsdatascience.com/graph-attention-networks-in-python-975736ac5c0c.

[11] Neptune.ai, "Graph neural network and some of gnn applications: Everything you need to know." https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications.