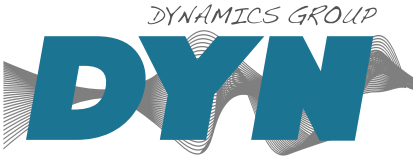


Dynamics Group (M-14)
Schloßmühlendamm 30
21073 Hamburg

Reinforcement Learning for Brake Squeal Mitigation: A Data-Driven Approach

Created by Amr Ahmed

Hamburg, Dezember 2024

Examiner: Prof. Dr. N. Hoffmann
Supervisor: Ernst Nathanael Winter

Declaration of Academic Honesty

I'm, Amr Ahmed (student of Msc. Mechatronics engineering at the Technical University of Hamburg 568447), solemnly declare that I have independently written the present Project Arbeit and have not used any resources other than those specified. The thesis has not been submitted to any examination board in this form before. .

Hamburg, December 3, 2024

Contents

List of Figures

Nomenclature

Following a common convention, matrices and vectors are denoted with bold letters and constants and variables with normal letters. If they are time-discrete quantities, the indication (matrix, vector, variable) is understood for a point in time.

List of Symbols

Latin List of Symbols

Symbol	Unit	Meaning
A_t	-	Action at time t
G_t	-	Return at time t
$Q(s, a)$	-	Action-value function
R_t	-	Reward at time t
S_t	-	State at time t
$V(s)$	-	State-value function
FZ	N	Normal force applied on the pin
FY	N	Lateral friction force (tangential force)
$FY\_Smoothed$	N	Smoothed tangential force exerted on the pin
V	m/s	Velocity of the brake rotor
$VitesseMoteur$	tr/min	Rotational velocity of the disc
DEP_D	V	Normal displacement of the disc
TC_BE_S	°C	Temperature at the exterior edge surface
TC_BE_M	°C	Temperature 2mm under the exterior edge
TC_BI_S	°C	Temperature at the interior edge surface
TC_BI_M	°C	Temperature 2mm under the interior edge
TC_BI_E	°C	Temperature at the entry point of the interior edge
TC_BE_10	°C	Temperature at 10mm below the exterior edge
$isSqualing$	-	Binary indicator for the presence of squeal noise

Symbol	Unit	Meaning
a	-	Action
r	-	Reward
s	-	State
t	s	Time step for simulation
μ	-	Coefficient of friction

Greek List of Symbols

Symbol	Unit	Meaning
α	-	Learning rate
γ	-	Discount factor
ϵ	-	Exploration-exploitation parameter
π	-	Policy
θ	-	Policy parameters

Indexes

Index		Meaning
t	=	Time step
i	=	Episode index

Abbreviations

Abbr.		Meaning
DRL	=	Deep Reinforcement Learning
DQN	=	Deep Q-Network
MDP	=	Markov Decision Process
RL	=	Reinforcement Learning
LSTM	=	Long Short-Term Memory
IQR	=	Interquartile Range (used for outlier detection)

1 Introduction

1.1 Overview

The automotive industry is undergoing a transformative shift towards data-driven methodologies, revolutionizing vehicle safety, efficiency, and comfort. One critical area that stands to benefit significantly from these advancements is the braking domain. This thesis focuses on tackling a persistent and crucial problem in braking systems: Brake squeal, a high-frequency noise emitted during braking, poses a major concern for automakers due to its detrimental impact on vehicle comfort and perceived quality.

1.2 Goal of the Thesis

The primary goal of this thesis is to utilize reinforcement learning (RL), an advanced machine learning paradigm, to simulate a Pin-and-Disk experiment conducted by Lille University. The focus is on controlling the most dominant features that directly or indirectly influence brake squeal sound while ensuring the overall system remains stable. The simulation accounts for continuous changes in the braking system, influenced by factors such as wear, temperature fluctuations, and varying load conditions. States in the simulation are generated using a state predictor based on a pretrained machine learning model, alongside a squeal generator.

1.3 Objectives

The thesis will pursue the following specific objectives:

- **Brake Squeal: Causes, Effects, and Mitigation approaches:**
  - ▷ investigate the nature of brake squeal, identifying its root causes and consequences.
  - ▷ Review existing methods and techniques employed to address brake squeal in the automotive industry.
- **Reinforcement Learning: A Literature Review:**
  - ▷ Conduct a comprehensive review of various RL techniques, their applications in control systems, and the latest research findings.
  - ▷ Examine the theoretical foundations, algorithms, and determine which of these options would be most effective in addressing the current brake squeal challenge we are working to mitigate, while maintaining system stability.

- **Dataset Understanding and Environmental Modeling:**
  - ▷ Gain in-depth knowledge of the dataset recorded during a pin-on-disc experiment conducted by the University of Lille.
  - ▷ Analyze the system inputs, outputs, and experimental procedures used for data collection.
  - ▷ Define relevant environmental states and determine their operational ranges for accurate modeling.
- **RL-Based Controller Implementation and Deployment:**
  - ▷ Design and implement an RL-based controller to manage the simulated pin-on-disc experiment.
  - ▷ Perform hyperparameter studies to fine-tune the controller and find an optimal control policy.
  - ▷ Transition the controller to experimental pin-on-disc equipment in Lille, France, ensuring real-time adaptability to dynamic conditions.
- **Documentation, Dissemination, and Publication:**
  - ▷ Thoroughly document the research process, results, and insights.
  - ▷ Provide suggestions for enhancing the results in future work, focusing on improving the controller's performance and expanding the project's scope as needed to develop a comprehensive, data-driven, and reliable simulation environment.

1.4 Thesis Structure

The structure and development of this thesis draw significant inspiration from the works of [4], [5], [6], [7], [1], [2] [8], and [28]. These foundational studies have provided critical insights into the mechanics of brake squeal, data-driven methodologies, and advanced control strategies, shaping the framework and approach of the present research. Their contributions have been instrumental in guiding the organization, analytical focus, and methodology presented throughout this thesis.

- **Chapter 1: Brake Squeal: A Multifaceted Challenge** Introduces the phenomenon of brake squeal, its causes, effects, and conventional mitigation strategies. It further explores the emerging trend of leveraging data-driven solutions in this domain.

- **Chapter 2: Introduction to Machine Learning: The Foundation of Intelligent Systems** Establishes the theoretical groundwork for reinforcement learning (RL), covering essential concepts such as Markov Decision Processes (MDPs), value and policy functions, and Bellman equations. It also reviews various RL algorithms and their applications in control systems.
- **Chapter 3: Implementation: Training a DQN Agent to Mitigate Brake Squeal** Elaborates on the implementation of the RL-based controller, detailing the reward function design, neural network architecture, and the training and validation methodologies. This chapter also presents the experimental results from simulations and real-world deployments, analyzing their implications and proposing future research directions.
- **Chapter 4: Summary of Findings and Perspectives for Future Work** Summarizes the key insights gained from the literature review and the implementation of reinforcement learning for brake squeal mitigation. It highlights the effectiveness of combining Long Short-Term Memory (LSTM) architectures with a carefully designed reward function to address the dynamic and noisy environment of braking systems. The chapter also presents a detailed discussion on the simulation environment and pretrained models used, emphasizing their role in enhancing prediction accuracy and ensuring the RL agent's policies are transferable to real-world scenarios. Finally, it provides recommendations for future work, including extended training cycles, diverse braking scenarios, and alternative reward function designs, to further refine the model and expand its applicability in the automotive industry.

1.5 Significance and Impact

By addressing the pervasive problem of brake squeal through the innovative application of reinforcement learning, this thesis aims to contribute to the development of more advanced and adaptive braking systems. The proposed dynamic brake squeal map has the potential to significantly reduce brake squeal and associated particle emissions, leading to improved vehicle performance, enhanced passenger comfort, and increased customer satisfaction. Furthermore, this research may pave the way for the broader adoption of RL in other automotive applications, unlocking new possibilities for intelligent vehicle control and optimization.

2 Brake Squeal: A Multifaceted Challenge

2.1 Fundamentals of Brake Squeal

When you press the brake pedal, the braking system is engaged—a mechanical device designed to inhibit motion by absorbing energy from a moving system. In this process, the vehicle's kinetic energy is converted into heat through friction generated between the brake pads and the rotating disc (rotor) [1]. However, this energy transfer is not entirely smooth; a portion of the energy induces vibrations within the brake components. When these vibrations align with the natural frequencies of the brake system, they create a phenomenon known as brake squeal [2].

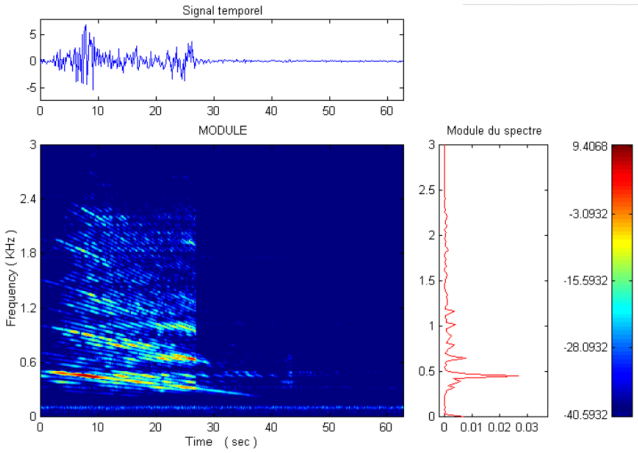


Figure 2.1: A spectrogram depicting the high-frequency noise (squeal) emitted during braking. Source: [33].

Brake squeal typically falls within the frequency range of 1 to 16 kHz [1]. While not necessarily indicative of brake malfunction, this high-pitched sound is a common nuisance that automakers strive to minimize through careful design and material selection [5]. The phenomenon arises from complex mechanical interactions within the braking system, involving factors such as:

- **Friction:** The force resisting the relative motion between the brake pad and rotor [2].
- **Vibration:** The oscillating motion of the brake components [1].
- **Material Properties:** The stiffness, damping, and friction characteristics of the brake pads, rotor, and other components [5].

Understanding these fundamental principles is crucial for developing effective strategies to predict and mitigate brake squeal [4].

The information presented in this chapter is based on several foundational works that have shaped the understanding of brake squeal and its mitigation. Specifically, insights into vibrational mechanics and modal analysis are drawn from Ewins [1] and Dufrénoy et al. [2]. Discussions on the role of material properties and damping measures are informed by Gräbner et al. [5]. Finally, the exploration of data-driven approaches and machine learning applications builds upon the works of Geier et al. [4] and Hoffmann et al. [6].

2.2 Recent Approaches to Brake Squeal Mitigation

The automotive industry has invested significant effort in tackling brake squeal, employing both active and passive measures.

2.2.1 Active Measures

Active measures involve the use of external devices or systems to counteract the vibrations that cause squeal. One promising approach is the integration of piezoelectric devices into brake pads. These devices can generate forces in response to vibrations, effectively damping them and reducing noise. While effective, active measures can be expensive and add complexity to the brake system.

2.2.2 Passive Measures

Passive measures, more commonly used due to their simplicity and cost-effectiveness, involve modifications to the brake system's design and materials to dampen vibrations. Examples include:

- **High-Damping Materials:** Utilizing materials with inherent damping properties for brake rotors and pads can help absorb vibrational energy.
- **Damping Rings/Shims:** These additional components, attached to the brake rotor or between the pad and caliper, can add mass and damping to the system.
- **Laminated Brake Discs:** These discs consist of multiple layers of material with different damping properties, designed to disrupt vibrational modes that lead to squeal.
- **Modified Pad/Rotor Geometry:** Chamfering or slotting brake pads, or introducing asymmetry in the rotor design, can alter the contact dynamics and reduce the likelihood of squeal.

While passive measures have shown some success, they often address the symptoms rather than the root cause of brake squeal. Additionally, they may have drawbacks like added weight, cost, and potential for uneven wear.

2.3 Data-Driven Solutions: The Rise of Machine Learning and RL

The integration of machine learning (ML) and its advanced branch, reinforcement learning (RL), presents transformative potential in predicting and mitigating brake squeal. Machine learning models excel in analyzing vast datasets of historical brake performance, identifying patterns, and quantifying conditions under which squeal events are likely to occur. These insights can be used to predict squeal with high precision, effectively functioning as a **passive approach** by identifying problematic scenarios without actively intervening.

Reinforcement learning, on the other hand, introduces a dynamic and **active-oriented solution**, enabling real-time optimization of braking strategies. By simulating brake system dynamics and learning from outcomes in a controlled environment, RL agents can explore how changes in parameters like braking force, pad composition, or temperature influence squeal generation. Based on this learning, RL can develop adaptive strategies to minimize squeal occurrences during actual operation, providing a real-time feedback loop for system adjustment.

This dual approach offers a comprehensive solution to the brake squeal problem. While ML provides predictive insights by studying historical data, RL focuses on solving the issue dynamically by adjusting system parameters in response to real-time conditions. Together, these methods do not merely address the symptoms (vibrations) but aim to uncover and address the root causes of friction-induced instabilities. As such, they represent a significant shift in the brake squeal mitigation paradigm, combining the advantages of both **passive identification** and **active resolution**.

Furthermore, current research in this domain emphasizes the role of AI in refining brake system designs, such as material selection or geometry optimization, based on predictive results. By incorporating these AI-driven insights into both simulation and real-world testing, it is possible to develop robust, data-driven solutions that enhance brake system reliability and long-term stability. .

2.3.1 Addressing the Gap: Real-Time Adaptive Control

A significant gap in brake squeal mitigation lies in the absence of systems capable of adapting to dynamic operating conditions in real time. Traditional methods rely on fixed parameters and predetermined designs that cannot adjust to variations in temperature, pressure, wear, or other factors influencing brake performance. Reinforcement learning (RL) offers a groundbreaking approach to bridging this gap by enabling braking systems to continuously adapt and optimize their performance based on real-time feedback.

RL-driven adaptive control systems leverage data from sensors monitoring the braking process, such as pad temperature, friction levels, and vibration patterns. By learning from this feedback, the system can dynamically adjust braking force, rotor geometry, or damping mechanisms to minimize squeal. This represents a shift from static, preemptive measures to a responsive,

self-correcting paradigm. Such systems promise not only to mitigate squeal but also to improve overall braking efficiency, durability, and safety.

2.3.2 Limitations and Future Directions

While RL introduces promising avenues for brake squeal mitigation, several challenges remain. One major limitation is the substantial computational resources required to train and deploy RL models capable of handling the complexity of real-world braking systems. Additionally, designing effective reward functions that accurately represent squeal mitigation objectives and account for system stability poses a significant challenge. Poorly constructed reward functions can lead to suboptimal or even unsafe control policies.

Another critical concern is the safety and reliability of RL in high-stakes environments like automotive systems. Ensuring that the RL models generalize well to unseen conditions and remain robust under extreme operating scenarios requires extensive testing and validation. The interpretability of RL-based decisions is another area needing attention, as it is essential to gain trust and regulatory approval for their deployment.

Future research should focus on:

- Developing computationally efficient RL algorithms tailored to real-time applications in braking systems.
- Designing hybrid approaches that integrate physics-based models with RL for enhanced stability and predictability.
- Exploring multi-agent RL frameworks to manage interactions between various brake system components.
- Establishing rigorous testing protocols and safety guarantees to ensure the reliability of RL-based systems in diverse conditions.

By addressing these limitations, RL can evolve from a theoretical solution to a practical tool for advancing brake squeal mitigation.

2.4 Conclusion

Understanding the complex mechanics of brake squeal, combined with the exploration of both traditional and modern data-driven approaches, is key to developing advanced mitigation strategies. Traditional methods, while effective in certain scenarios, often fail to adapt to the dynamic and nonlinear nature of brake systems. Reinforcement learning, with its ability to optimize performance in real time, offers a transformative path forward.

By harnessing the capabilities of machine learning and RL, the automotive industry can aim to achieve quieter, more reliable, and efficient braking systems. Continued research and innovation

in this domain have the potential to redefine the standards of brake system performance, ensuring safer and more sustainable mobility solutions for the future.

3 Introduction to Machine Learning: The Foundation of Intelligent Systems

Machine learning is a field of artificial intelligence that focuses on developing algorithms and models that enable computers to learn from data and make predictions or decisions without being explicitly programmed. Instead of relying on hand-crafted rules, ML systems learn patterns and relationships within the data through experience.

The information included in this chapter is derived from foundational works in the fields of machine learning and reinforcement learning, including Sutton and Barto's seminal book on reinforcement learning [8], foundational papers on Q-learning and value functions [12, 15], and research articles on advanced methods like prioritized experience replay and policy gradient techniques [11, 17, 19]. These sources provide the theoretical and practical insights upon which the concepts and methodologies discussed in this chapter are based.

3.1 Types of Machine Learning

Machine learning can be broadly categorized into three main paradigms:

1. ### 3.1.1 Supervised Learning

   The algorithm learns from labeled examples, where each input is paired with its corresponding desired output. Supervised learning finds applications across various domains, particularly excelling in tasks where labeled data is available to guide the model's training. Its strength lies in solving problems that require clear input-output mappings. Notable applications include:

   - **Image Classification**: Identifying objects, people, or scenes in images, such as classifying handwritten digits or detecting specific features in medical imaging (e.g., tumor detection in radiology).
   - **Spam Filtering**: Differentiating between legitimate and spam emails by analyzing patterns in text, sender information, and metadata.
   - **Medical Diagnosis**: Assisting in diagnosing diseases by analyzing patient data, including symptoms, lab results, or imaging, to predict conditions like diabetes, cancer, or cardiovascular issues.
   - **Speech Recognition**: Converting spoken language into text for applications like virtual assistants or automated transcription services.

- **Customer Churn Prediction**: Analyzing customer behavior to predict and prevent potential loss in subscription-based businesses or services.
- **Financial Fraud Detection**: Identifying unusual transaction patterns to flag potential fraudulent activities in banking and e-commerce.

2. 3.1.2 Unsupervised Learning

The algorithm learns from unlabeled data, discovering hidden patterns or structures within the data. Common applications include clustering (grouping similar data points, such as customer segmentation or gene classification in bioinformatics), dimensionality reduction (reducing the number of features while preserving information, often used for data visualization or preprocessing), and anomaly detection (identifying unusual data points, such as fraud detection in financial transactions or fault detection in manufacturing systems).

3. 3.1.3 Reinforcement Learning (RL)

The algorithm learns through interaction with an environment, receiving feedback in the form of rewards or penalties for its actions. The goal is to learn a policy (a strategy for choosing actions) that maximizes the cumulative reward over time. Examples include game playing (e.g., AlphaGo mastering Go), robotics (e.g., robotic arms learning to assemble products), and autonomous systems (e.g., self-driving cars optimizing navigation and decision-making in real-world traffic conditions).

3.2 The Role of Reinforcement Learning

Reinforcement learning stands out from the other paradigms due to its focus on learning from interaction and its ability to handle sequential decision-making problems. Unlike supervised and unsupervised learning, which often deal with static datasets, RL agents must learn to adapt their behavior dynamically in response to a changing environment.

3.3 Building Blocks of Reinforcement Learning

The key elements of reinforcement learning include:

- **Agent**: The learner and decision-maker.
- **Environment**: The external world with which the agent interacts.
- **State**: A representation of the environment at a given time.

- **Action**: A decision or choice made by the agent.
- **Reward**: A numerical signal that provides feedback to the agent about the goodness or badness of its actions.
- **Policy**: A strategy that maps states to actions, guiding the agent's behavior.
- **Value Function**: A function that estimates the long-term expected return from a given state or state-action pair.

The interplay between these elements forms the core of the RL process, enabling agents to learn complex behaviors and achieve their goals in a variety of settings.

3.4 Applications of Reinforcement Learning

Reinforcement learning has found applications in a wide range of domains, including:

- **Game Playing**: AlphaGo, a program developed by DeepMind, defeated the world champion at the board game Go using RL.
- **Robotics**: RL has been used to teach robots to walk, grasp objects, and perform complex tasks in real-world environments.
- **Autonomous Systems**: RL is a key technology in developing self-driving cars, drones, and other autonomous systems.
- **Resource Management**: RL can be used to optimize the allocation of resources in various domains, such as energy grids and data centers.

With its unique capabilities and broad applicability, reinforcement learning has emerged as a vital and rapidly growing area of machine learning research and development.

3.5 Reinforcement Learning: Learning from Interaction

Reinforcement learning (RL) is a unique paradigm in machine learning that focuses on how agents learn to make decisions by interacting with their environment [8]. In RL, an agent learns through a process of trial and error, receiving feedback in the form of rewards or penalties for its actions. The agent's goal is to discover an optimal policy, a mapping from states to actions, that maximizes the cumulative reward it receives over time.

3.6 Key Characteristics of RL

- **Goal-Oriented**: RL agents are driven by a specific goal or objective, typically defined by maximizing the cumulative reward.
- **Sequential Decision Making**: RL problems involve making a series of decisions over time, where each action can influence not only the immediate reward but also future states and rewards.
- **Exploration and Exploitation**: RL agents must balance the need to explore new actions and states with the need to exploit the knowledge they have gained so far to maximize their rewards.

3.7 Markov Decision Processes (MDPs): The Mathematical Framework

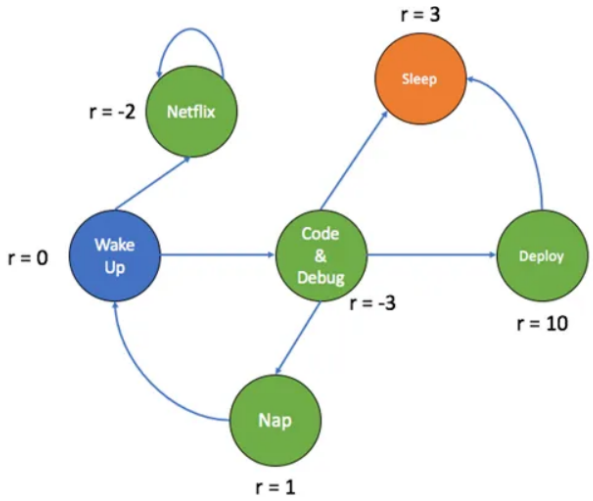


Figure 3.1: A diagram illustrating a Markov Decision Process (MDP), showing states (circles), actions (arrows), transitions (probabilities on arrows), and rewards (numbers). Source: [34].

To formally model reinforcement learning problems, we use the framework of Markov Decision Processes (MDPs). MDPs provide a mathematical way to represent the interaction between an agent and its environment, capturing the dynamics of the environment and the agent's decision-making process. Each state, action, and reward in an MDP defines the possible transitions the agent can make, as well as the immediate feedback it receives. The diagram 3.1 above illustrates a simplified example of an MDP, where states like *Wake Up*, *Netflix*, and *Deploy* represent the agent's options, and the rewards guide the agent's strategy to maximize its long-term benefit.

3.7.1 Defining Markov Decision Processes

An MDP is characterized by the following key components:

- **Set of States** (S): Each state $s \in S$ represents a specific configuration or situation of the environment at a given time.
- **Set of Actions** (A): Each action $a \in A$ represents a possible choice or decision that the agent can make in a given state.
- **Transition Function** ($P(s'|s, a)$): This function defines the probability of transitioning from state s to state s' when action a is taken. It captures the dynamics of the environment, specifying how the agent's actions influence the state transitions.
- **Reward Function** ($R(s, a, s')$): This function defines the immediate reward the agent receives upon transitioning from state s to state s' after taking action a. Rewards are essential for guiding the agent's learning process.
- **Discount Factor** (γ): This parameter, with a value between 0 and 1, determines the present value of future rewards. A higher discount factor emphasizes long-term rewards, while a lower discount factor prioritizes immediate rewards.

3.7.2 The Objective in an MDP

The core objective in an MDP is to find an optimal policy—a strategy that maps states to actions—that maximizes the expected cumulative reward over time. This cumulative reward, often called the return, is formally represented as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.1}$$

where:

- G_t represents the total expected return starting from time step t.
- R_{t+k+1} is the reward received at time step $t + k + 1$.
- γ^k is the discount factor raised to the power of k, which discounts future rewards exponentially based on their temporal distance from the current time step.

This formulation highlights the importance of strategic decision-making in reinforcement learning. The agent must strike a balance between immediate rewards and the potential for greater rewards in the future.

- **States (S)**: The different activities you can engage in, such as *Wake Up*, *Netflix*, *Code & Debug*, *Nap*, *Deploy*, or *Sleep*.

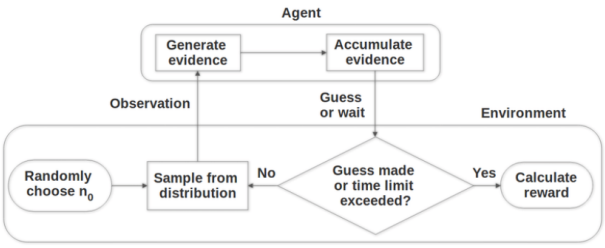


Figure 3.2: A flowchart depicting the agent-environment interaction in reinforcement learning. The agent observes the state, takes an action, receives a reward, and transitions to a new state. Source: [38].

- **Actions (A)**: The choices you make to transition between activities, such as deciding to *start coding* after *waking up* or switching to *Netflix* after *coding*.
- **Transition Function (P(s' | s, a))**: The probability of ending up in activity s' (e.g., *Nap*) after choosing an action a (e.g., *rest*) from the current activity s (e.g., *Code & Debug*).
- **Reward Function (R(s, a, s'))**: The reward or penalty associated with transitioning between activities, such as gaining a reward of 10 for *Deploy* or a penalty of -3 for *Code & Debug*.
- **Discount Factor (γ)**: Your preference for immediate rewards (e.g., enjoying Netflix now) versus delayed gratification (e.g., deploying code for long-term success).

Your goal is to find the optimal activity sequence (policy) that maximizes your overall rewards throughout the day (cumulative reward). The value function $V^{\pi}(s)$ then represents your expected total reward for starting your day in a given activity s and following your chosen plan.

3.7.3 The Agent-Environment Interaction

In an MDP, the agent and the environment interact in a sequential manner. At each time step t, the agent observes the current state of the environment S_t and chooses an action A_t based on its policy. The environment then transitions to a new state S_{t+1} based on the transition function and provides a reward R_{t+1} to the agent based on the reward function. This process continues iteratively, forming a trajectory of states, actions, and rewards:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

The goal of the agent is to learn a policy that maximizes its expected cumulative reward over this trajectory.

By understanding the fundamental concepts of MDPs, we gain the necessary foundation to explore the rich and diverse landscape of reinforcement learning algorithms.

3.7.4 Policy Function: Formulation and Decision Criteria

From what we have discussed so far, either from the Netflix activity analogy or the interaction between the agent and environment, we can define the policy function as the strategy used by an agent to determine its actions based on the current state of the environment. Now, in order to understand how policies are formulated and the criteria for their selection, we will examine the balance between essential terms like exploration and exploitation, various policy strategies such as ϵ-greedy (which will be discussed again later in this chapter), and softmax, as well as considerations for selecting an effective policy in different scenarios. Understanding these concepts is crucial for developing agents that can effectively navigate and make decisions in complex environments.

Exploration vs. Exploitation

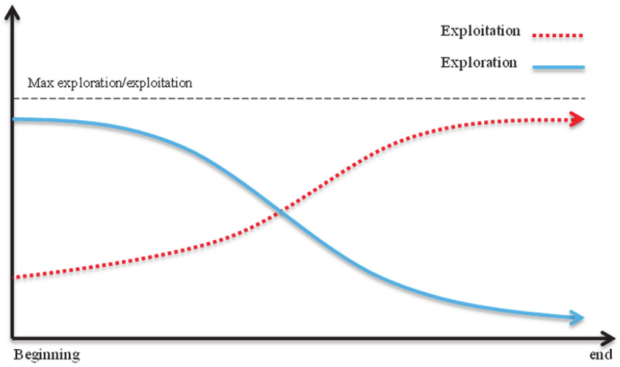


Figure 3.3: A graph illustrating the trade-off between exploration (trying new actions) and exploitation (choosing the best-known action) in reinforcement learning. Source: [37]. Licensed under Creative Commons Attribution 4.0 International.

Exploitation: The policy can exploit the known information, such as the value function, to choose actions that maximize immediate rewards. This means following high-rated routes based on current knowledge.

Exploration: The policy also needs to explore less known routes to discover potentially better options. This is crucial when the value function does not indicate a clear high-rated route.

Figure 3.3 illustrates the balance between exploration and exploitation over time. Early in the learning process, exploration dominates as the agent gathers information about the environment. Gradually, as more is learned, the focus shifts towards exploitation to maximize rewards based on the accumulated knowledge.

ϵ-Greedy Policy

One common approach to balance exploration and exploitation is the ϵ-greedy policy.

- With probability $1 - \epsilon$, the policy selects the action that maximizes the expected reward (exploitation).
- With probability ϵ, the policy selects a random action (exploration).

The ϵ-greedy policy can be formally represented as:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a =_a Q(s,a) \\ \frac{\epsilon}{|A(s)|} & \text{otherwise} \end{cases} \tag{3.2}$$

In these formulas:

- $Q(s,a)$ represents the action-value function, which quantifies the expected utility of taking action a in state s. This function integrates both the immediate reward and the discounted future rewards, thus providing a comprehensive measure of the action's value from the current state.
- $\pi(a|s)$ is the probability of selecting action a when in state s, according to the policy π. Policies can be deterministic, always choosing the action with the highest value, or stochastic, where actions are selected based on probabilities to allow for exploration and exploitation.
- ϵ represents a small positive value, typically starting near one and decaying over time. This ensures there is always a chance of exploring new actions while gradually focusing on exploiting known good actions, preventing the agent from getting stuck in a local optimum.
- $|A(s)|$ denotes the total number of actions available in state s. This value ensures the exploration step is uniformly distributed across all available actions.

In practice, the ϵ-greedy approach enables the agent to balance between exploration, where it seeks new strategies, and exploitation, where it leverages its existing knowledge to maximize rewards. A typical implementation involves decaying ϵ from an initial value like 1.0 (fully random) to a lower bound such as 0.1 or 0.01, allowing the agent to converge towards a more optimal policy over time.

.

Softmax Policy

The softmax policy utilizes a softmax function to transform outputs into a probability distribution, assigning a specific probability to each possible action.In other words actions are chosen according to a probability distribution that depends on their value estimates. Higher-valued actions are more likely to be chosen, but lower-valued actions still have a chance. The Softmax policy can be formally represented as:

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{b \in A(s)} e^{Q(s,b)/\tau}} \tag{3.3}$$

In this formula:

- $P(a|s)$ represents the probability of selecting action a in state s. This probability is proportional to the exponentiated value of $Q(s, a)$, ensuring that actions with higher values are more likely to be chosen, while still allowing for the possibility of selecting lower-valued actions.
- $Q(s, a)$ represents the action-value function, which estimates the expected reward for taking action a in state s. This value incorporates both immediate rewards and future discounted rewards, providing a comprehensive measure of an action's potential benefit.
- τ is a temperature parameter that controls the degree of exploration. The term "temperature" is borrowed from statistical mechanics, where higher temperatures lead to more randomness. In the context of reinforcement learning, a higher τ value increases exploration by making the action probabilities more uniform, thus encouraging the selection of a wider range of actions. Conversely, a lower τ value favors exploitation by making the action probabilities more peaked around the highest-valued actions, thus focusing on the best-known actions.
- The denominator, $\sum_{b \in A(s)} e^{Q(s,b)/\tau}$, is the sum of the exponentiated value estimates for all actions b available in state s. This term ensures that the probabilities $P(a|s)$ sum to 1, normalizing the action probabilities so they form a valid probability distribution.

The benefits of using the Softmax policy include:

- **Balanced Exploration and Exploitation**: The Softmax policy allows for a tunable balance between exploration and exploitation through the temperature parameter τ. This flexibility helps in finding an optimal policy by adequately exploring the action space and then exploiting the best actions found.
- **Smooth Probability Distribution**: By using the exponential function, the Softmax policy ensures that the probabilities of selecting different actions are smoothly distributed. This avoids the sharp distinctions between actions that are typical in deterministic policies, leading to more robust decision-making in uncertain environments.

- **Avoiding Overcommitment**: The policy prevents overcommitment to suboptimal actions early in the learning process by maintaining a probability distribution that gives some weight to all actions. This helps in thoroughly exploring the action space, which can be critical in complex environments.
- **Mathematical Tractability**: The exponential function is smooth and differentiable, which makes the Softmax policy amenable to gradient-based optimization techniques. This is particularly advantageous in scenarios where policy gradients are used to improve the policy iteratively.

The use of the exponential function in the Softmax policy has specific advantages:

- **Sensitivity to Differences in Values**: The exponential function magnifies differences in action values $Q(s, a)$. Small differences in $Q(s, a)$ are translated into larger differences in probabilities $P(a|s)$, making the policy more sensitive to the value estimates.
- **Normalization**: The exponential function ensures that the resulting probabilities are always positive and sum to 1 when divided by the normalization term (the sum of exponentials). This creates a valid probability distribution without requiring additional constraints.

Analytical insights into the Softmax policy include:

- **Gradient of the Softmax Function**: The gradient of the Softmax function with respect to $Q(s, a)$ can be derived and used in policy gradient algorithms. This gradient is essential for updating policies in reinforcement learning frameworks.
- **Temperature Annealing**: In practice, the temperature parameter τ can be annealed (gradually reduced) over time to shift the policy from exploration to exploitation. This annealing process helps in fine-tuning the policy towards the optimal strategy as more information is gathered.

Criteria for Selecting a Policy

- **Maximize Expected Return**: The primary goal is to maximize the expected cumulative reward (return) over time.
- **Adaptability**: The policy should adapt based on new information and learning. This ensures that the agent can improve its decisions as it gains more experience.
- **Balance Exploration and Exploitation**: A good policy maintains a balance between exploring new actions and exploiting known rewarding actions.

When High-Rated Routes Are Not Available

In certain scenarios, the value function may not provide a clear high-rated route, or the agent may encounter situations where all available actions have low or uncertain value estimates. This lack of high-rated routes necessitates a strategic approach to decision-making to ensure the agent continues to improve and adapt its policy effectively. In such cases, the policy must prioritize exploration over exploitation to discover new information and potentially better routes. The following strategies are commonly employed to handle these situations and ensure the agent can still make progress towards optimal decision-making:

Exploration: The policy might increase exploration to gather more information about the environment. This can involve taking random actions or actions that have not been tried frequently.

Using Prior Knowledge: If no high-rated routes are available, the policy might rely on prior knowledge or heuristic methods to select actions that are likely to yield good outcomes.

Multi-Armed Bandit Approaches: In scenarios where immediate rewards need to be balanced against exploration, techniques from multi-armed bandit problems can be applied to formulate policies that dynamically adjust exploration rates .

3.7.5 Value Function: The Compass of Optimal Policy

In our pursuit of an optimal policy, the value function emerges as a critical guiding light. It quantifies the long-term desirability of being in a particular state while adhering to a given policy. By understanding the value function, we gain the ability to predict future rewards and make informed decisions that maximize cumulative gains. Let's delve deeper into the mathematical formulations and the unique insights each provides.

State-Value Function: Measuring State's Potential

The state-value function, denoted as $V^\pi(s)$, captures the expected total reward an agent can anticipate when starting from state s and consistently following policy π:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{3.4}$$

Let's break down the components of this equation:

$V^\pi(s)$: The value of being in state s under policy π. This is the central quantity we want to estimate and maximize. $\mathbb{E}_\pi$: The expectation (average) under policy π. This signifies that we're averaging over all possible future trajectories the agent might take, given its actions are chosen according to π. G_t: The return (cumulative discounted reward) from time step t onwards. It

encompasses the immediate reward at time t plus the discounted sum of future rewards. $S_t = s$: The condition that the agent is in state s at time t.

In essence, the state-value function tells us how "good" it is to be in a particular state. A higher value implies a greater potential for accumulating rewards over the long run.

Action-Value Function: Evaluating Action's Impact

While the state-value function provides an overall assessment of a state, the action-value function, denoted as $Q^\pi(s, a)$, zooms in on the value of taking a specific action a in state s while following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{3.5}$$

This equation builds upon the state-value function but adds a crucial element:

$A_t = a$: The condition that the agent takes action a at time t.

The action-value function is instrumental in refining our decision-making. It quantifies the expected return not just for being in a state, but for taking a specific action within that state. By comparing action-values, we can identify the optimal action – the one that leads to the highest expected return – and thereby improve our policy.

The Interplay: Guiding Optimal Behavior

The state-value and action-value functions are deeply intertwined. We can express the state-value function in terms of the action-value function and vice-versa. This relationship forms the bedrock of many reinforcement learning algorithms.

By iteratively refining our estimates of these value functions, we gradually uncover the optimal policy – a policy that consistently chooses actions that maximize long-term rewards.

3.8 Bellman Equations: The Recursive Road to Optimal Values

After grasping the concept of value functions, we can now appreciate the power of Bellman equations. These equations offer a recursive way to decompose and compute value functions, which quantify the expected return from a given state (or state-action pair). Bellman equations form the cornerstone of many reinforcement learning algorithms, as they break down the complexity of decision-making into a series of interconnected steps.

3.8.1 Bellman Equation for the State-Value Function

The Bellman equation for the state-value function, $V^\pi(s)$, under policy π is:

$$\begin{aligned} V^\pi(s_t) &= \mathbb{E}_{a_t\sim\pi, s_{t+1}\sim P}\left[r(s_t,a_t) + \gamma V^\pi(s_{t+1})\right] \\ &= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R} P(s',r|s,a)\left[r+\gamma V^\pi(s')\right]. \end{aligned} \tag{3.6}$$

In this equation:

- $V^\pi(s_t)$: The value of being in state s_t at time t under policy π.
- $\mathbb{E}_{a_t\sim\pi, s_{t+1}\sim P}[\cdot]$: The expectation over actions a_t (chosen according to policy π) and next states s_{t+1} (determined by the environment's transition probabilities P).
- $r(s_t, a_t)$: The immediate reward received after taking action a_t in state s_t.
- γ: The discount factor, controlling the importance of future rewards.
- $V^\pi(s_{t+1})$: The value of the next state s_{t+1}, again under policy π.

The equation reveals the recursive nature of the value function: the value of being in a state now depends on the immediate reward we get and the discounted value of the states we might end up in later.

3.8.2 Bellman Equation for the Action-Value Function

The Bellman equation for the action-value function, $Q^\pi(s,a)$, is:

$$\begin{aligned} Q^\pi(s_t,a_t) &= \mathbb{E}_{s_{t+1}\sim P}\left[r(s_t,a_t) + \gamma\mathbb{E}_{a_{t+1}\sim\pi}\left[Q^\pi(s_{t+1},a_{t+1})\right]\right] \\ &= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R} P(s',r|s,a)\left[r+\gamma Q^\pi(s',a')\right]. \end{aligned} \tag{3.7}$$

Most terms are the same as in Equation 3.6, but the key difference lies in focusing on the value of both the current state s_t and the chosen action a_t. It introduces an additional expectation over the next action a_{t+1}, acknowledging that the future trajectory (and thus the total return) depends not just on where we are, but what we do there.

3.8.3 Bellman Optimality Equations

If we seek the optimal value functions (and thus the optimal policy), we use the Bellman Optimality Equations:

$$\begin{aligned} V^*(s_t) &= \max_a \mathbb{E}_{s_{t+1}\sim P}\left[r(s_t,a) + \gamma V^*(s_{t+1})\right] \\ &= \max_a \sum_{s'\in S, r\in R} P(s', r|s, a)\left[r + \gamma V^*(s')\right]. \end{aligned} \tag{3.8}$$

$$\begin{aligned} Q^*(s_t,a_t) &= \mathbb{E}_{s_{t+1}\sim P}\left[r(s_t,a_t) + \gamma \max_{a'} Q^*(s_{t+1},a')\right] \\ &= \sum_{s'\in S, r\in R} P(s', r|s, a)\left[r + \gamma \max_{a'} Q^*(s',a')\right]. \end{aligned} \tag{3.9}$$

These equations are similar to their policy-specific counterparts, but instead of averaging over actions according to a fixed policy π, we now take the maximum over all possible actions. This is because we're seeking the best possible value, achievable by always choosing the optimal action.

3.8.4 Bellman's Principle of Optimality

The foundation of these Bellman equations is Bellman's Principle of Optimality. It states that an optimal policy has the property that, regardless of the initial state and action, the subsequent decisions must also constitute an optimal policy. This means that an optimal policy is made up of optimal sub-policies. This principle enables us to break down the problem of finding an optimal policy into smaller, solvable subproblems.

3.9 The Role of Rewards and Discounting

The agent's learning process is fundamentally driven by rewards, which act as feedback signals indicating the success or failure of its actions. The reward function, $R(s, a)$, assigns a numerical value (the reward) to each action a taken in a given state s.

3.9.1 Defining Rewards

Rewards can be defined in various ways:

- As a function of the action: $R(a)$, where the reward depends solely on the action taken.

- As a function of the state-action pair: $R(s, a)$, where the reward depends on both the current state and the action taken.
- As a function of the transition: $R(s, a, s')$, where the reward depends on the current state, the action taken, and the resulting next state.

The choice of reward formulation depends on the specific problem and the information available to the agent.

3.9.2 Discounting Future Rewards

The agent's objective is to maximize the expected cumulative reward over time. However, rewards received in the future are generally considered less valuable than immediate rewards. This is where the discount factor, γ, comes into play. The discount factor, a value between 0 and 1, allows us to prioritize immediate rewards over future rewards. The discounted return, G_t, is calculated as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.10}$$

In this equation:

- G_t is the total discounted return from time step t onwards.
- R_{t+k+1} is the reward received at time step $t + k + 1$.
- γ^k is the discount factor raised to the power of k, where k represents the number of time steps into the future.

A higher value of γ (closer to 1) places more emphasis on future rewards, encouraging the agent to plan for the long term. A lower value of γ (closer to 0) makes the agent more "myopic," focusing primarily on immediate rewards.

3.9.3 The Importance of Discounting

Discounting future rewards serves several important purposes:

- **Mathematical Convenience**: It ensures that the infinite sum of rewards in Equation 3.10 converges to a finite value, making it mathematically tractable.
- **Reflecting Uncertainty**: It accounts for the inherent uncertainty in future rewards. The further into the future we look, the less certain we are about the outcomes.
- **Modeling Preference**: It allows us to model the agent's preference for immediate versus delayed gratification.

By adjusting the discount factor, we can control the agent's behavior and its balance between exploring new options and exploiting known rewards.

3.10 Advancements in Model-Free Reinforcement Learning

Model-free reinforcement learning (RL) has evolved significantly beyond the foundational approaches like Monte Carlo, SARSA, Q-learning, and TD(λ). While these methods establish the core principles of learning from interactions, they often face challenges such as inefficiency in high-dimensional spaces and instability in dynamic environments. In response, advanced techniques have been developed to overcome these limitations, focusing on improving scalability, stability, and learning efficiency. This section delves into two prominent advancements in model-free RL.

- **Dealing with Large or Continuous State Spaces**: When the state space becomes too vast to be represented by a table, we need function approximation techniques. This is where deep learning comes into play, enabling us to represent complex value functions and policies using neural networks.
- **Efficient Exploration**: Naive exploration strategies, like epsilon-greedy, can be inefficient in large or complex environments. Advanced exploration techniques, such as curiosity-driven exploration or intrinsic motivation, aim to guide the agent towards more informative and rewarding experiences.

Deep Reinforcement Learning: Combining Deep Learning and RL

Deep reinforcement learning (DRL) is a powerful combination of deep learning and reinforcement learning. It leverages the representation learning capabilities of deep neural networks to tackle high-dimensional state spaces and complex control tasks.

Deep Q-Networks (DQN): A Breakthrough in DRL

Deep Q-Networks (DQN) represent a landmark achievement in DRL. DQN extends Q-learning to high-dimensional state spaces by using a deep neural network to approximate the Q-value function. This allows DQN to learn directly from raw sensory input, such as images or video frames. DQN also incorporates experience replay and target networks for stable and efficient learning.

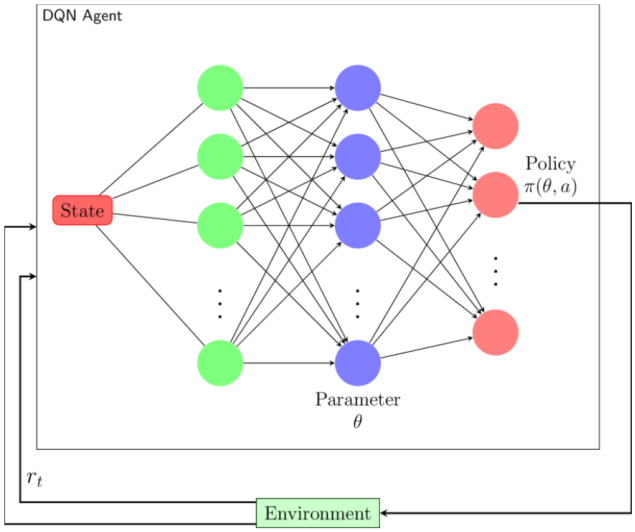


Figure 3.4: The architecture of a Deep Q-Network (DQN) showing the input layer (e.g., image), hidden layers (neural network), and output layer (Q-values for each action). The replay memory and target network are also depicted. Source: [36].

Advanced Exploration Strategies

Beyond epsilon-greedy, various advanced exploration techniques have been developed to address the challenges of efficient exploration in complex environments. These include:

- **Count-Based Exploration**: The agent favors actions that have been taken less frequently.
- **Optimism in the Face of Uncertainty**: The agent prefers actions whose outcomes are uncertain, as they offer greater potential for learning.
- **Intrinsic Motivation**: The agent is driven by curiosity or a desire to explore novel states or behaviors.

By incorporating these and other advanced exploration strategies, model-free RL agents can learn more effectively and efficiently in challenging environments.

This revised section now provides a smoother transition into the discussion of model-based methods and highlights the ongoing research and development in extending model-free RL techniques.

3.11 Model-Based Approaches: Planning with a Map

In contrast to model-free methods, model-based RL algorithms build an explicit model of the environment. This model can be used for planning, simulating different trajectories, and evaluating the potential outcomes of various actions.

3.11.1 Dynamic Programming: Planning with Perfect Information

Dynamic Programming (DP) methods are a classic example of model-based RL. They require a perfect model of the environment and use it to compute optimal policies efficiently. However, DP methods are often computationally expensive and intractable for large or continuous state and action spaces.

3.11.2 Other Model-Based Methods: Learning and Planning

In many practical scenarios, a perfect model of the environment is not available. Instead, the agent can learn a model from experience and use it for planning. This approach is called model-based reinforcement learning with learned models. Examples include Dyna-Q and model-based policy optimization algorithms.

3.11.3 Advantages and Limitations of Model-Based Methods

Advantages:

- Sample efficiency: Model-based methods typically require fewer interactions with the environment to learn, as they can leverage the model for planning and simulating trajectories.
- Ability to plan and reason: The explicit model allows for better planning and reasoning about the consequences of actions, potentially leading to more sophisticated and optimal policies.

Limitations:

- Model accuracy: The effectiveness of model-based methods heavily relies on the accuracy of the learned environment model. Inaccurate models can lead to suboptimal or even harmful policies.
- Computational complexity: Building and using an environment model can be computationally demanding, particularly for complex environments with large state and action spaces.

3.12 Choosing the Right Approach

The choice between model-based and model-free methods depends on the specific problem and the available resources.

- If a good model of the environment is available or can be easily learned, model-based methods can be very effective.
- If the environment is complex or poorly understood, model-free methods may be a better choice.
- Hybrid approaches that combine elements of both model-based and model-free learning can sometimes offer the best of both worlds.

In this thesis, we focus on model-free approaches due to the complexity and challenges of modeling the specific environment we are dealing with. However, we acknowledge the potential benefits of model-based methods and recognize their growing importance in the field of reinforcement learning.

3.13 Policy-Based vs. Value-Based Methods: Two Paths to Optimal Decision-Making

In the realm of reinforcement learning, algorithms can be broadly classified into two distinct categories: policy-based and value-based methods. These approaches offer different strategies for tackling the challenge of learning optimal policies in complex environments.

3.13.1 Policy-Based Methods: Directly Shaping the Strategy

Policy-based methods directly optimize the policy itself, aiming to find the best mapping from states to actions that maximizes the expected return. They do this by adjusting the parameters of the policy function (often represented by a neural network) to make high-reward actions more likely.

The Policy Gradient Theorem

The foundation of many policy-based algorithms is the policy gradient theorem. This theorem tells us how to adjust the policy's parameters to increase the expected return. The policy gradient is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) Q_{\pi_\theta}(s, a) \right] \tag{3.11}$$

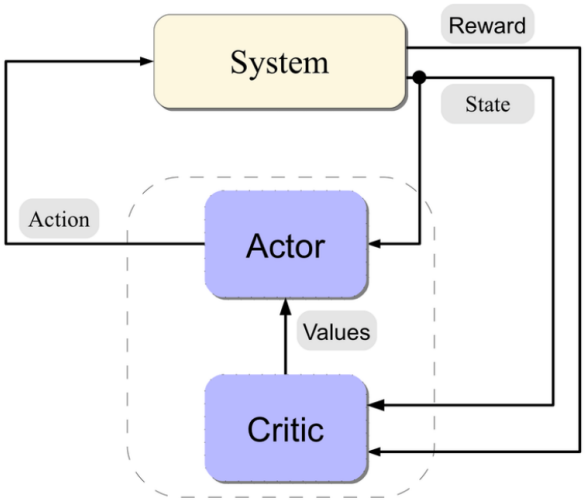


Figure 3.5: A block diagram of the actor-critic architecture used in Deep Deterministic Policy Gradient (DDPG). The actor network generates actions, while the critic network estimates action values. Source: [35].

In this equation:

- $\nabla_\theta J(\theta)$ is the gradient of the expected return $J(\theta)$ with respect to the policy parameters θ.
- $\mathbb{E}_{\pi_\theta}$ denotes the expectation under the current policy π_θ.
- $\nabla_\theta \log \pi_\theta(a \mid s)$ is the gradient of the log probability of taking action a in state s under the current policy.
- $Q_{\pi_\theta}(s, a)$ is the action-value function, representing the expected return after taking action a in state s and following the current policy thereafter.

Advantages and Limitations of Policy-Based Methods

Advantages:

- **Effective for Continuous Action Spaces**: Policy-based methods are well-suited for problems with continuous action spaces, where directly parameterizing the policy can be more natural and effective than learning a value function.
- **Stochastic Policies**: They can learn stochastic policies, which can be beneficial in environments with inherent randomness or where exploration is important.
- **Convergence Properties**: Often demonstrate better convergence properties than value-based methods.

Limitations:

- **High Variance**: The gradient estimates in policy-based methods can exhibit high variance, which can make learning unstable.
- **Sample Inefficiency**: They may require a large number of samples to learn good policies, as they often rely on Monte Carlo estimates of the policy gradient.

3.13.2 Value-Based Methods: Estimating Value to Guide Actions

Value-based methods take a different approach. They focus on learning the value function, which estimates the expected return for each state or state-action pair. The policy is then derived implicitly from the value function: the agent chooses the action with the highest estimated value in a given state.

Bellman Equation and Value-Based Methods

As discussed in Section 3.3, the Bellman equation is a fundamental tool for learning value functions. It provides a recursive relationship between the value of a state and the values of its successor states.

Advantages and Limitations of Value-Based Methods

Advantages:

- **Simpler Implementation**: Value-based methods are often easier to implement than policy-based methods, as they don't require direct differentiation of the policy.
- **Efficient in Discrete Action Spaces**: They are particularly well-suited for problems with discrete action spaces, where the agent can easily compare the estimated values of different actions.

Limitations:

- **Deterministic Policies**: Value-based methods naturally lead to deterministic policies, which can limit exploration in some environments.
- **Challenges with Continuous Actions**: They are generally not as effective for problems with continuous action spaces, as finding the action that maximizes the value function can be computationally expensive.

3.14 The Landscape of RL Algorithms: Matching Tools to Tasks

The landscape of reinforcement learning (RL) algorithms is vast and diverse, encompassing a wide array of approaches designed to address the unique challenges presented by different environments and tasks. From foundational algorithms like Q-learning to advanced techniques like Deep Q-Networks (DQN) and Policy Gradient methods, RL offers a rich toolbox for solving a wide spectrum of problems.

3.14.1 Key Considerations for Algorithm Selection

Choosing the right RL algorithm is not a one-size-fits-all endeavor. It hinges on a deep understanding of the environment's characteristics, including:

- **State and Action Spaces**: Are they discrete (finite number of distinct values) or continuous (infinite range of values) ?
- **Dynamics**: Is the environment deterministic (outcomes are fully predictable given the state and action) or stochastic (outcomes have an element of randomness) ?
- **Available Information**: Does the agent have access to a model of the environment, or does it need to learn from experience ?

By carefully considering these factors, we can match the appropriate RL algorithm to the task at hand. For example, DQN and its variants excel in environments with discrete actions, while policy gradient methods are well-suited for continuous action spaces.

In this chapter, we'll explore the foundational concepts and methodologies that underpin many RL algorithms. These include:

- **Action-value methods**: Estimating the value of taking specific actions in given states.
- **Markov Decision Processes (MDPs)**: A mathematical framework for modeling sequential decision-making problems.
- **Agent-environment interaction**: Understanding how agents interact with and learn from their environment.
- **Policy and value functions**: Representing strategies for choosing actions and estimating the expected return.
- **Bellman equations**: Recursive equations for calculating value functions.
- **Experience replay**: A technique for storing and reusing past experiences to improve learning.

By delving into these fundamental building blocks, we'll lay the groundwork for understanding and applying a wide range of RL algorithms to solve real-world challenges.

3.15 Reward Function and Loss Function: Guiding Learning and Measuring Progress

Two key components in the RL learning process are the reward function and the loss function.

3.15.1 Reward Function: Shaping Agent Behavior

The reward function, often denoted as $R(s, a)$ or $R(s, a, s')$, serves as the compass guiding the agent towards desired behavior. It assigns a numerical value (the reward) to each action the agent takes in a given state. This reward signal provides feedback, reinforcing successful actions and discouraging less favorable ones.

Designing Effective Reward Functions

Crafting an effective reward function is crucial, as it directly influences the agent's learning process and ultimate performance. Here are some key considerations:

- **Alignment with Task Goals**: The reward function should accurately reflect the goals of the task the agent is trying to accomplish.
- **Balancing Multiple Objectives**: Many tasks involve multiple objectives, such as safety, efficiency, and performance. The reward function must balance these potentially competing objectives.
- **Sparsity**: In some environments, rewards may be sparse, meaning they are only received infrequently. Techniques like reward shaping can help address this by providing intermediate rewards to guide the agent towards long-term goals.

Example: Autonomous Driving

In the context of autonomous driving, the reward function might include components for:

- **Safety**: Rewards for avoiding collisions, maintaining safe distances, and obeying traffic laws.
- **Efficiency**: Rewards for reaching destinations quickly while minimizing fuel consumption.
- **Passenger Comfort**: Rewards for smooth driving and avoiding harsh maneuvers.

3.15.2 Loss Function: Measuring and Minimizing Errors

The loss function plays a crucial role in training RL agents, particularly those using neural networks as function approximators (e.g., in Deep Q-Networks). It quantifies the difference between the predicted values (e.g., Q-values or policy probabilities) and the target values (e.g., estimated returns or expert demonstrations).

Common Loss Functions in RL

Common loss functions used in RL include:

- **Mean Squared Error (MSE)**: Measures the average squared difference between predicted and target values.
- **Huber Loss**: A robust loss function that is less sensitive to outliers than MSE.
- **Kullback-Leibler (KL) Divergence**: Measures the difference between two probability distributions, often used in policy gradient methods.

Role of the Loss Function

The loss function serves as a guide for the optimization algorithm (e.g., gradient descent) to update the agent's parameters. By minimizing the loss, the agent's predictions become more accurate over time, leading to improved decision-making and performance.

3.16 Experience Replay: Learning from the Past

Experience replay is a fundamental technique in reinforcement learning that allows agents to learn more efficiently from their past experiences. It addresses some of the limitations of traditional online learning methods, where updates are made based only on the most recent interaction with the environment.

3.16.1 The Experience Replay Mechanism

In experience replay, the agent stores its experiences in a replay memory, also known as a replay buffer. Each experience typically consists of a tuple:

$$(s_t, a_t, r_{t+1}, s_{t+1})$$

where:

- s_t is the state at time t.
- a_t is the action taken at time t.
- r_{t+1} is the reward received after taking action a_t.
- s_{t+1} is the next state the agent transitions to.

The replay memory accumulates these experiences over multiple episodes. During training, the agent randomly samples a batch of experiences (a mini-batch) from the replay memory and uses them to update its value function or policy.

3.16.2 Algorithm: Experience Replay

Algorithm 1 Experience Replay

```
Initialize replay buffer D
for episode = 1 to M do
    Initialize state s_1
    for t = 1 to T do
        Select action a_t from s_t using policy π (e.g., ϵ-greedy)
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_{t+1}, s_{t+1}) in D
        Sample random minibatch of transitions (s_i, a_i, r_{i+1}, s_{i+1}) from D
        Update value function or policy using the minibatch
    end for
end for
```

3.16.3 Benefits of Experience Replay

Experience replay offers several advantages:

- **Data Efficiency**: Each stored experience can be reused multiple times for learning, making more efficient use of the data collected from the environment.
- **Reduced Variance**: Sampling experiences randomly from the replay buffer decorrelates updates, reducing the variance of the updates compared to standard online learning.
- **Breaking Correlations**: By using experiences from different time steps and episodes, experience replay helps to break the temporal correlations that can exist in the sequence of experiences, leading to more stable and robust learning.
- **Off-Policy Learning**: Experience replay enables off-policy learning, where the agent can learn from experiences generated by a different policy (e.g., a previous version of its own policy).

3.16.4 Prioritized Experience Replay

A variant of experience replay called prioritized experience replay further improves efficiency by prioritizing experiences based on their importance for learning. This is often done using a criterion like the TD error, which measures the surprise or unexpectedness of an experience. By sampling experiences with higher TD errors more frequently, the agent can focus its learning on the most informative transitions.

3.17 Reinforcement Learning Algorithms: A Taxonomy

With a solid understanding of the fundamental principles of reinforcement learning (RL), we now turn our attention to the diverse landscape of RL algorithms. These algorithms are designed to solve a wide range of sequential decision-making problems, from game playing and robotics to autonomous driving and resource allocation.

3.17.1 Classification of RL Algorithms

We can classify RL algorithms based on the characteristics of the environment they are designed for:

- **Environments with Limited States and Discrete Actions**: These environments have a small, finite number of states and actions. They are relatively simple and often serve as a good starting point for learning RL.
- **Environments with Unlimited States and Discrete Actions**: These environments have a large or even infinite number of states, but the actions are still discrete. This category includes many real-world problems where the state space is too large to enumerate explicitly.
- **Environments with Unlimited States and Continuous Actions**: These are the most challenging environments, with both large state spaces and continuous action spaces. They require specialized algorithms that can handle the complexities of continuous control.

3.17.2 Algorithms for Limited States and Discrete Actions

Q-Learning: Learning Action Values

Q-learning is a model-free, value-based algorithm that learns the optimal action-value function, $Q(s, a)$, which represents the expected return for taking action a in state s and following the optimal policy thereafter. The Q-value update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- $Q(s,a)$: The Q-value of taking action a in state s.
- α: Learning rate, controlling the step size of the update.
- r: Immediate reward received after taking action a in state s.
- γ: Discount factor, balancing the importance of immediate and future rewards.
- $\max_{a'} Q(s',a')$: The maximum Q-value among all possible actions in the next state s'.

Algorithm 2 Q-Learning

```
Initialize Q(s, a) arbitrarily for all state-action pairs
for each episode do
    Initialize s
    repeatfor each step of episode
        Choose a from s using policy derived from Q (e.g., ϵ-greedy)
        Take action a, observe reward r and next state s′
        Q(s, a) ← Q(s, a) + α [r + γ max_a′ Q(s′, a′) − Q(s, a)]
        s ← s′
    until s is terminal
end for
```

Example Application: Q-learning can be applied to simple games like Tic-Tac-Toe or grid-world navigation.

3.17.3 Algorithms for Unlimited States and Discrete Actions

Deep Q-Networks (DQN): Scaling Q-Learning with Neural Networks

DQN extends Q-learning to high-dimensional state spaces by using a deep neural network to approximate the Q-value function. This allows DQN to learn from raw sensory input, such as images or video frames, making it applicable to a wider range of problems.

Key DQN Components:

- **Experience Replay**: Stores past experiences in a buffer and samples them randomly for training, improving data efficiency and reducing correlations.
- **Target Network**: A separate network with the same architecture as the main Q-network, used to provide stable targets for learning.

Algorithm 3 Deep Q-Network (DQN)

Initialize replay memory D
Initialize Q-network with random weights θ
Initialize target network with weights $\theta^- = \theta$
for each episode **do**
    Initialize state s_1
    for $t = 1$ to T **do**
        Select action a_t from s_t using ϵ-greedy policy based on $Q(s_t, \cdot; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta^-)$ if episode terminates at step $j + 1$, otherwise $y_j = r_j$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
        if Every C steps **then**
            Reset $\theta^- = \theta$
        end if
    end for
end for

Example Application: DQN achieved breakthrough performance in playing Atari games from raw pixel input.

3.17.4 Algorithms for Unlimited States and Continuous Actions

Deep Deterministic Policy Gradient (DDPG): Combining Policy and Value Learning

DDPG is an actor-critic algorithm that combines policy-based and value-based approaches. It uses an actor network to learn a deterministic policy and a critic network to estimate the action-value function. This enables DDPG to handle continuous action spaces effectively.

Key DDPG Components:

- **Actor Network**: Directly outputs deterministic actions for a given state.
- **Critic Network**: Estimates the action-value function $Q(s, a)$ to evaluate the actions taken by the actor.
- **Replay Buffer**: Stores experiences and samples them for training.
- **Target Networks**: Used to stabilize the learning process.

Algorithm 4 Deep Deterministic Policy Gradient (DDPG)

Initialize actor network $\mu(s|\theta^\mu)$ and critic network $Q(s, a|\theta^Q)$ with random weights
Initialize target networks $\mu'(s|\theta^{\mu'})$ and $Q'(s, a|\theta^{Q'})$ with weights $\theta^{\mu'} \leftarrow \theta^\mu$, $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer R
for each episode **do**
    Initialize a random process N for action exploration
    Receive initial observation state s_1
    for $t = 1$ to T **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + N_t$
        Execute action a_t and observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient: $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
        - $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        - $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
end for

Example Application: DDPG has been successfully applied to robotics tasks involving continuous control, such as manipulating objects or controlling robotic arms.

Conclusion. Now that we have thoroughly discussed the foundational concepts of machine learning, including its paradigms such as supervised, unsupervised, and reinforcement learning, as well as the building blocks and methodologies underpinning reinforcement learning, we are well-prepared to delve into its practical applications. With a solid understanding of agent-environment interactions, policies, value functions, and the balance between exploration and exploitation, we can now transition to the next chapter.

In the following chapter, we will examine the implementation details of the chosen reinforcement learning model, the criteria for its selection, and the step-by-step process of adapting it to address the specific challenge of mitigating brake squeal while maintaining system stability. This will include discussions on the simulation environment, training setup, and evaluation metrics employed to assess its performance. Let us now move forward to explore how these theoretical principles translate into practical solutions.

4 Implementation: Training a DQN Agent to Mitigate Brake Squeal

4.1 Implementation Overview

The foundation of this thesis is the Pin and Disk Experiment conducted by the University of Lille, which recorded a dataset capturing various scenarios of the experiment. Additionally, pretrained models developed by the M14 department at Hamburg University of Technology are employed, consisting of two key models: the **Simulation Model**, which predicts state channels at the next timestep, and the **Squeal Model**, which predicts the presence of brake squealing. These models are integrated with a reinforcement learning-based controller to be developed in this thesis. This controller receives feedback from the pretrained models in the form of 8 state channels and decides the optimal input channels of normal force and environment channels of motor speed required to maintain system stability and minimize brake squeal. The controller is implemented using a model-based Q-table reinforcement learning algorithm.

By managing the system states, as described in this chapter, and constraining the controller's actions, the system achieves indirect control to ensure the desired stability. The control problem is formulated as a Markov Decision Problem (MDP), which is effectively solved using a Q-table reinforcement learning algorithm, as outlined in [23]. The Q-table algorithm demonstrates the ability to manage nonlinear system dynamics by directly controlling the normal force applied to the brake pad and the motor speed, using a single controller to indirectly maintain the trajectory of system states as they are generated by the pretrained models.

While the Q-table approach is well-suited for limited state spaces, scaling to larger state spaces introduces challenges. Expanding the number of discrete states necessitates larger Q-tables, which align with the needs of this project but can lead to memory constraints and significant time and data requirements for accurate population. Addressing arbitrarily large state spaces in this context becomes impractical.

The dataset originates from the pin-on-disk setup, capturing extensive measurements to simulate real-world friction dynamics. These datasets encompass system inputs, environmental conditions, state variables, and squeal occurrence indicators, stored in a hierarchical data format (H5). Accurate and thorough examination of this data ensures the reliability of the research findings and shapes the research direction, hypotheses, and conclusions. A focused overview of data characteristics and processing methodologies highlights their critical role in maintaining research integrity and applicability.

Given the sequential nature of the data, the implementation incorporates the Long Short-Term Memory (LSTM) layer in the reinforcement learning model. LSTMs, with their ability

to model temporal dependencies and address vanishing gradient issues, are integral for handling time-dependent state transitions and sequential decision-making in partially observable environments.

The validation process identifies the optimal normal force and motor speed values to maintain stability. Through integration, the system initializes with data snippets and uses the simulation and squeal models, alongside motor speed and normal force adjustments, to predict state trajectories and squeal outcomes until the defined end time of the cycle. Once the development and training of the RL controller is completed, the same validation process is used again to retrieve the stored controller weights and replay memory saved during the training process. The controller then replaces the normal force and motor speed with actions it generates, which dynamically update the force and speed. Through feedback from the 8 system states and squeal prediction results, the controller provides the force and speed required to maintain system stability and minimize squeal.

4.2 Dataset Overview

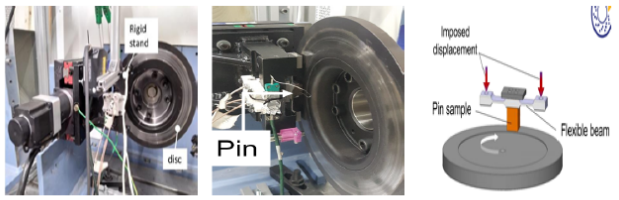
The research utilizes a time-series dataset to explore the behavior of Non-Asbestos Organic-Based (NAOB) friction materials under varying rotational dynamics. NAOB materials are widely used in braking systems for their environmentally friendly composition and their ability to maintain stable frictional performance across diverse conditions.

Recorded at a 90Hz resolution, the dataset consists of 706 braking tests, capturing critical parameters such as forces, temperatures, displacements, and frequencies during braking events. These data points provide a comprehensive understanding of the dynamic behavior of braking systems under controlled experimental conditions.

The experimental setup features a 0.36m diameter steel disc interacting with a NAOB pin. Controlled rotational speeds and contact times are applied to simulate real-world braking conditions, ensuring the dataset accurately represents the performance of the friction material. This extensive dataset serves as a robust foundation for analyzing the system's dynamics and developing effective control strategies.

4.3 Data Structure Overview

The dataset, integral to this research, is meticulously organized into two primary categories: `derivative_data` and `test_conditions`. Each category is further divided into several subcategories, which are structured to facilitate easy access and detailed analysis. The organization of these categories is vital for accurately interpreting the data and ensuring the robustness of the research findings. The data is organized in the following format:



(a) Experimental setup. (b) NAOB pin and steel disc. (c) Animated Model of NAOB Pin-Disc Contact.

Figure 4.1: Experiment setup. Source: [30].

Keys beneath `derivative_data` and `test_conditions`:

- `axis0`
- `axis1`
- `block0_items`
- `block0_values`

4.3.1 Derivative Data

This category encompasses the core measurements and observations from the experimental data. It is crucial for understanding the dynamic behavior of the NAOB friction materials under test conditions.

- `axis0`: Contains parameter names encoded as byte strings, reflecting the types of data captured, e.g., `[b'FZ', b'FCT_BI_S', b'FCT_BI_E', b'FCT_BE_E', b'TC_BE_S']`. These parameters represent key variables such as forces, temperatures, and displacements.
- `axis1`: Lists indices corresponding to the parameters in `axis0`, providing a numerical reference for data alignment and retrieval, e.g., `[0, 1, 2, 3, 4]`.
- `block0_items`: Mirrors `axis0` by listing the same parameters, ensuring consistency across different data structures, e.g., `[b'FZ', b'FCT_BI_S', b'FCT_BI_E', b'FCT_BE_E', b'TC_BE_S']`.
- `block0_values`: Comprises a nested list of values for each parameter, forming the dataset's backbone. These values are critical for subsequent analyses, e.g., `[-0.25311189623761, -5.421400819912945e-05, ...]`.

4.3.2 Test Conditions

This category details the specific conditions under which each experiment was conducted. Understanding these conditions is essential for interpreting the data correctly and for replicating the experiments, if necessary.

- `axis0`: Lists parameters related to test conditions as byte strings, indicating the settings under which data was collected, e.g., `[b'VitesseMoteur', b'IncrementAXMO', b'T_OFF', b'T_ON', b'Trigger1']`.
- `axis1`: Provides indices for the test condition parameters, aligning them with their respective data points, e.g., `[0, 1, 2, 3, 4]`.
- `block0_items`: Similar to `axis0`, it enumerates the test condition parameters for easy reference and consistency, e.g., `[b'VitesseMoteur', b'Increment AXMO', b'T_OFF', b'T_ON', b'Trigger1']`.
- `block0_values`: Contains a nested list of specific values for each test condition parameter. These values are instrumental in analyzing how different test conditions affect the behavior of the materials, e.g., `[200.0, 50000.0, 30.0, 30.0, -0.500795367486276, -1.0000164363695627]`.

This structured approach to data organization not only enables efficient access but also ensures a detailed and accurate analysis of the experimental results. It is particularly crucial for studying the behavior of NAOB pins under various testing conditions, providing a foundation for the research's analytical rigor and reliability.

The following algorithm outlines the process of transforming the recorded data into a structured NumPy format, enabling seamless application of desired preprocessing techniques.

Key Features and Description

The focus is narrowed to the following key features: `FZ`, `VitesseMoteur`, `TC_BE_S`, `TC_BE_M`, `TC_BI_S`, `TC_BI_M`, `TC_BI_E`, `TC_BE_10`, `DEP_D`, `FY_Smoothed`, and `isSquealing`. These variables are critical for understanding the dynamic behavior of the NAOB friction materials under test conditions and are described in the following table.

Categorization of Features

- **System Input:**
  - ▷ Normal force (`FZ`): Represents the applied load on the pin, influencing the brake system dynamics.

Algorithm 5 Processing H5 Files and Saving Combined Dataset

Input: Folder path containing H5 files, $folder\_path$
Output: Combined dataset saved as `combined_dataset_raw.npy`
Step 1: Process Individual Files
for each file in $folder\_path$ **do**
    Open the H5 file and check for required datasets
    Extract features: FZ, $VitesseMoteur$, TC_BE_S, TC_BE_M, TC_BI_S, TC_BI_M, TC_BI_E, TC_BE_10, DEP_D, $FY\_Smoothed$, and $isSquealing$
    Combine extracted features into a 2D array
    Append valid arrays to the processed data list
end for
Step 2: Combine Extracted Data
if processed data exists **then**
    Stack all 2D arrays into a single combined dataset
    Save the dataset as `combined_dataset_raw.npy`
else
    Print "No valid data found for processing"
end if

Table 4.1: Description of Variables and Units

Name	Unit	Description
`FZ`	N	Normal force applied on the pin
`VitesseMoteur`	tr/min	Rotational velocity of the disc
`TC_BE_S`	°C	Temperature at the exterior edge surface
`TC_BE_M`	°C	Temperature 2mm under the exterior edge
`TC_BI_S`	°C	Temperature at the interior edge surface
`TC_BI_M`	°C	Temperature 2mm under the interior edge
`TC_BI_E`	°C	Temperature at the entry point of the interior edge
`TC_BE_10`	°C	Temperature at 10mm below the exterior edge
`DEP_D`	V	Normal displacement of the disc
`FY_Smoothed`	N	Tangential force exerted on the pin (smoothed)
`isSquealing`	Binary	Indicates the presence of squeal noise

- **Environmental Condition:**
  - ▷ Rotational speed (`VitesseMoteur`): Provides the context for system behavior under varying dynamics.
- **State Variables:**
  - ▷ Temperature readings at key locations (`TC_BE_S`, `TC_BE_M`, `TC_BI_S`, `TC_BI_M`, `TC_BI_E`, `TC_BE_10`),
  - ▷ Displacement of the disc (`DEP_D`),

▷ Tangential force exerted on the pin (`FY_Smoothed`).

- **Squeal Occurrence:**

  ▷ Binary indicator (`isSquealing`): Signals the presence or absence of brake squeal noise, derived using an 80 dB threshold.

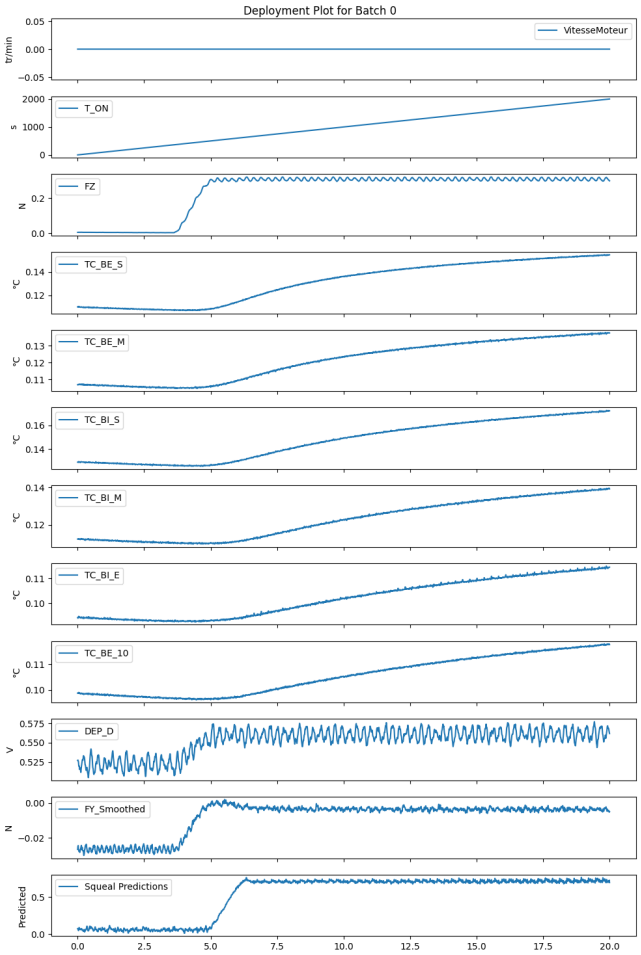


Figure 4.2: Sample recorded data structure showing variable organization.

4.4 Brake Analysis

To furtherly understand the nature of the scenarios used for creating those mentioned H5 files , we need first to understand that the dataset can be divided into groups based on braking event durations. Each group represents a distinct range of time steps or in simple words the longevity of which the pedal is pressed, aiding in targeted analysis of braking behavior.

The duration of the sample used to verify the controller's performance was chosen as 2000 time steps, which corresponds to approximately 20 seconds at a sampling frequency of 90Hz.

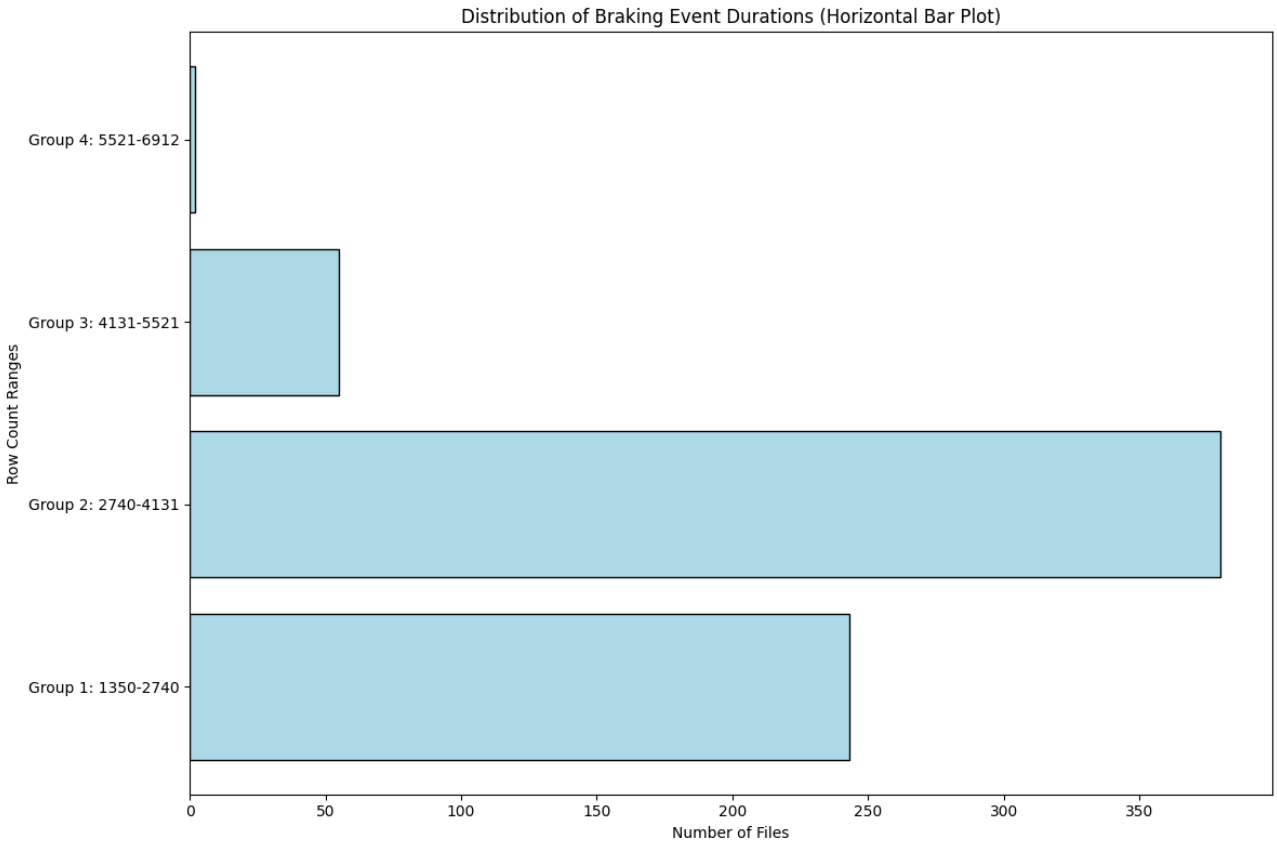


Figure 4.3: Braking analysis based on brake duration

This duration was selected primarily for simplification purposes, allowing for consistent and manageable analysis across all experiments.

While this value does not encompass the entire range of durations in the dataset, it closely aligns with the mean duration of 1511.4 time steps observed in Group 1 and provides a practical balance given the broad range of time steps in the data. By focusing on 2000 time steps, the analysis remains representative of the majority of the dataset while ensuring computational efficiency and simplicity.

Table 4.2: Group Statistics Based on Time Step Durations

Group	Range (Time Steps)	Count	Mean	Std Dev	Min-Max
1	1350-2740	243	1511.4	191.4	1350-2079
2	2740-4131	380	3806.2	140.9	3348-4059
3	4131-5521	55	4708.0	149.8	4563-4950
4	5521-6912	2	6484.5	427.5	6057-6912

4.4.1 Continuing with Data Exploration

As the final goal is to maintain system stability while reducing squeal occurrence according to the difference longevity of the presses on the brakes which is grouped into four groups according to the the following graph , however first,it is essential to first conduct a deeper statistical analysis of the Pin-and-Disk experiment. This includes identifying the correlation between the selected features to understand which of them has the most influence on squeal occurrence. Such statistical understanding of the correlations and data distribution is crucial. Additionally, it involves checking for any missing data or outliers that might require filtering to ensure the reliability and accuracy of the analysis.

In order to have this deeper understanding of the data and to achieve this target , a combination of boxplots, and heatmaps provides complementary insights:

- **Boxplots** The feature-wise Boxplots provide a comprehensive understanding of the central tendency, spread, and presence of outliers. Such information is instrumental for tailoring preprocessing techniques like normalization, scaling, and outlier removal, which we will be relying on here using quartiles and the interquartile range (IQR). The IQR, defined as the range between the first quartile (Q1) and third quartile (Q3), is particularly useful for identifying and handling outliers, as it highlights data points that deviate significantly from the central distribution.
- **Heat Maps** offer a visual representation of feature correlations, revealing strong associations or dependencies that might impact system behavior or squeal occurrence.

By combining these methods, a holistic perspective on the data is achieved, allowing for informed decisions regarding data preparation, filtering, and further analysis.

4.4.2 Outlier Analysis and Handling

Outliers play a critical role in understanding the data distribution and ensuring robust model performance. However, deciding whether to retain or remove outliers requires a nuanced approach, particularly when working with operational datasets like the Pin-and-Disk experiment.

Identifying Outliers: From the boxplot analysis (Figure 4.4), Outliers were removed using the Interquartile Range (IQR) except for T_ON, and isSquealing, as these represent fixed test conditions or binary values without numeric outliers. (Table 4.3).

Impact on Controller Design: Removing outliers ensures the controller focuses on core operational regions while avoiding destabilization caused by edge cases. However, retaining valid outliers within operational limits ensures that the controller can handle realistic variability in system behavior.

Implications for Preprocessing: The non-normal distributions observed in most features necessitate the use of non-parametric methods for subsequent analysis. Skewness and tails in

Feature	Min Value	Max Value
FZ	-6.8	547.6
VitesseMoteur	200.0	800.0
TC_BE_S	26.2	154.8
TC_BE_M	26.2	159.5
TC_BI_S	26.0	116.8
TC_BI_M	26.0	143.6
TC_BI_E	26.0	155.6
TC_BE_10	26.2	166.7
DEP_D	6.4	8.7
FY_Smoothed	-7.5	19.5
isSquealing	22.5	96.8

Table 4.3: Minimum and maximum values per feature.

features like **FZ** and **FY_Smoothed** highlight the need for normalization to improve model convergence and performance.

4.4.3 Correlation Analysis

Understanding the relationships between features is vital for identifying dependencies and redundancies in the dataset. For this purpose, Spearman's rank correlation coefficient, a non-parametric measure, was chosen due to the non-normal distribution observed in the dataset. The heatmap in Figure 4.5 visualizes the correlations among all features.

Key Observations:

- **FZ and FY_Smoothed**: The high positive correlation ($r = 0.97$) suggests that the normal force significantly influences the tangential force. This relationship is critical for understanding the system's frictional behavior.
- **FY_Smoothed and isSquealing**: The moderate correlation ($r = 0.64$) indicates that higher tangential forces are associated with increased squeal occurrence, aligning with domain expectations.
- **VitesseMoteur and FY_Smoothed**: A negligible correlation ($r = 0.09$) suggests that rotational velocity has little direct impact on tangential force. This feature may be less significant for modeling squeal occurrence. **The weak correlation ($r = 0.09$) highlights that rotational velocity indirectly influences tangential force through dynamic interactions.**

Now, as we have gained a deeper understanding of the data and are about to create the training, testing, and validation datasets, we have succeeded in understanding the data and removing outliers that might hinder the controller's ability to learn the data patterns and

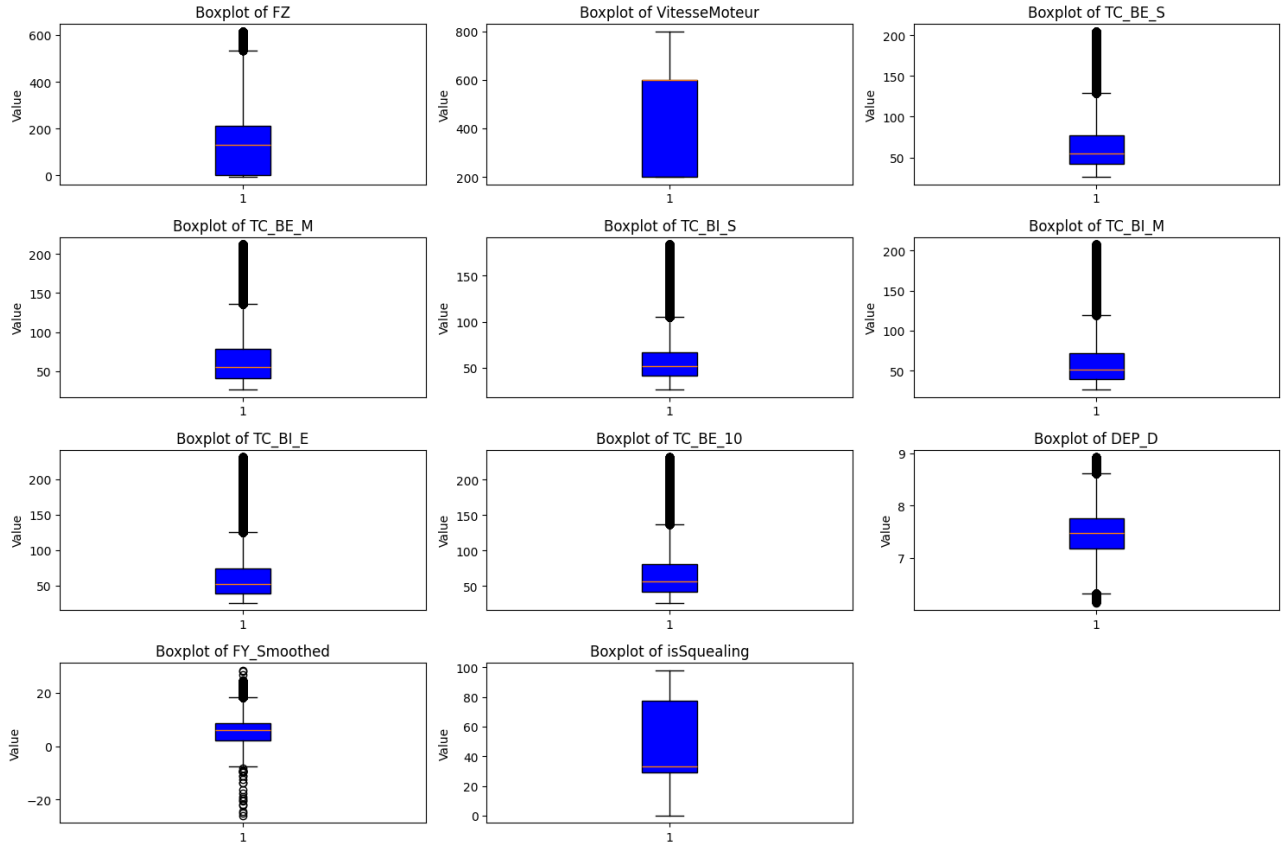


Figure 4.4: Feature-wise boxplot representation to identify central tendency, spread, and outliers in the dataset.

trends. Additionally, since we have specified the min and max values of the data, the logical next step is to normalize this data for the mentioned purposes. From the following box plot, the data that will be split later for training, validation, and testing is shown in Figure 4.6.

4.5 Continue Implementation: building a system to understand the accuracy of the system understandability

4.6 train test split

Now that we have completed extracting the recorded data from the H5 format into a more readable format using NumPy and Pandas, our tasks have been significantly facilitated. This process has allowed us to gain a deeper understanding of the braking scenarios used, grouping them, and preparing the data for further analysis. We have also set the min and max values 4.3 suitable for RL training purposes and, prior to that, for visualization using the validation system shown in the following algorithm. However, before proceeding, we need to split our

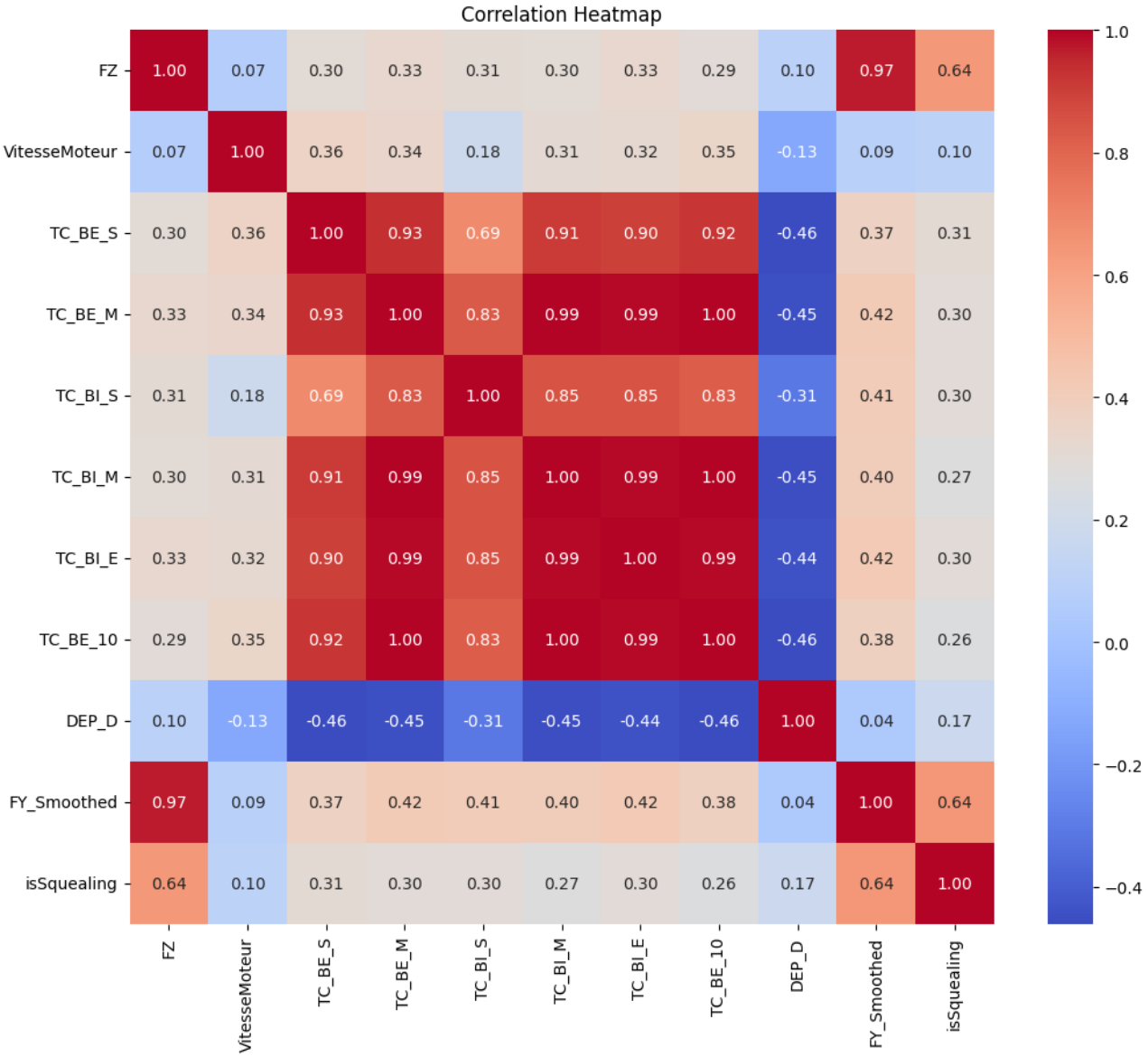


Figure 4.5: Correlation Analysis

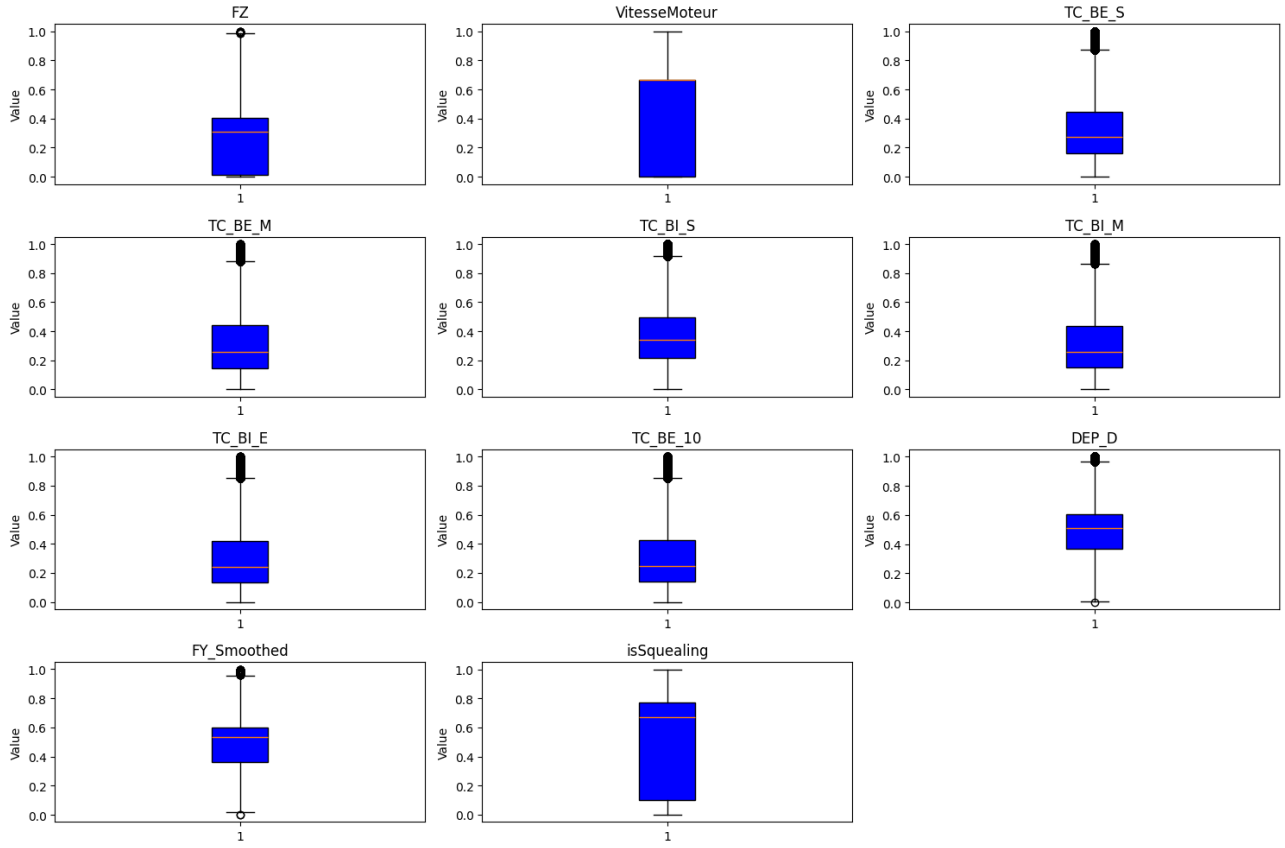


Figure 4.6: normalized boxplot

refined and normalized data—after removing outliers and normalizing—into datasets as shown in this plot 4.6 for 3 different datasets training, validation, and testing with the following shapes .

Table 4.4: Dataset Shapes After Normalization and Splitting

Dataset	**Shape (Samples, Features)**
Combined Dataset	(2,085,525, 11)
Training Data	(1,252,579, 11)
Validation Data	(268,410, 11)
Testing Data	(268,410, 11)

4.6.1 Virtual Environment for Experiment Simulation

A virtual environment was developed to simulate the experimental setup of the pin-on-disk system, integrating two machine learning models trained on the dataset provided by the University of Lille. These models are responsible for the following tasks:

- **Simulation Model:** Predicts state variables (`X`) for the next time step using 0.2s of input (`FZ`), environmental (`VitesseMoteur`), and current state data.
- **Squeal Model:** Determines the occurrence of squeal over the last 0.1s using environmental and state variables.

Usage of the Environment. The virtual environment is encapsulated in the `EnvLile` class. It enables multistep predictions by integrating simulation and squeal models. Key functionalities include:

- Initializing the environment with model paths and parameters.
- Generating predictions for state variables at the next time step and squeal occurrence over the last 0.1s using the `predict` method or `predict_tf`.
- Supporting normalization and de-normalization of data either manually (using provided normalization values) or automatically by enabling the `perform_normalization` parameter.

Input and Output Channels. The virtual environment processes the following data channels in a specific order:

- **Input (`U`):** `FZ`.
- **Environmental Condition (`E`):** `VitesseMoteur`.
- **State Variables (`X`):**
  - ▷ Pin temperatures: `TC_BE_S`, `TC_BE_M`, `TC_BE_10`, `TC_BI_S`, `TC_BI_M`, `TC_BI_E`,
  - ▷ Disc displacement: `DEP_D`,
  - ▷ Tangential force: `FY_Smoothed`.
- **Output:** State predictions and a binary indicator (`isSquealing`) for squeal occurrence.

4.7 Simulation Mechanism

To gain a deeper understanding of the functioning of this simulation model, let us examine the algorithm outlined in 6.

The algorithm describes a simulation mechanism that combines real data and predictions to iteratively simulate the system's behavior over a specified time range. Here's an explanation of the mechanism:

1. **Initialization (Steps 1–3):** The process begins by initializing the environment with:

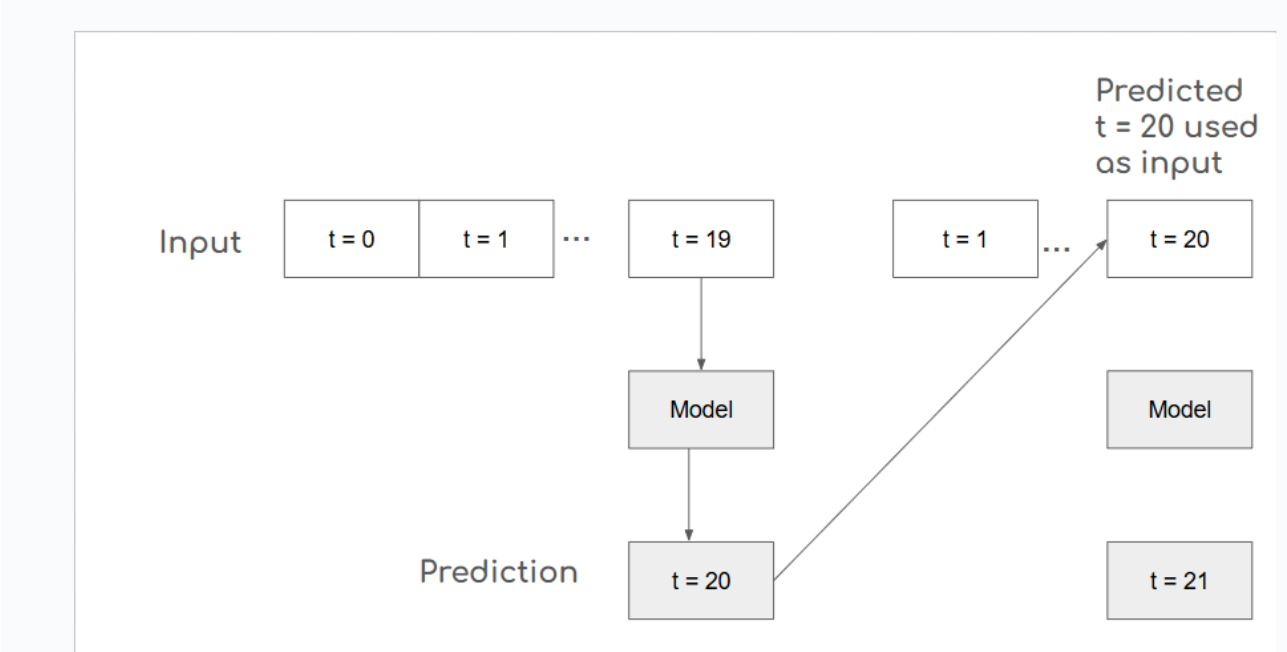


Figure 4.7: Model illustration. Adapted from [30].

- Paths to the squeal prediction model and the simulation model.
- Setting the `perform_normalization` parameter to `True`.

The algorithm then extracts key experimental data, including:

- Input channel (`FZ`).
- Environmental channel (`VitesseMoteur`).
- State channels (`TC_BE_S`, `TC_BE_M`, `DEP_D`, etc.).
- Binary squeal indicator (`isSquealing`).

Placeholders are initialized to store simulated predictions, including `full_U_steps_simulated`, `full_E_steps_simulated`, `full_X_steps_simulated`, and `full_squeal_pred`.

2. **Initialization with Real Data (Step 4):** The simulation starts by using the first 20 time steps of real recorded data as the initial input. This includes input channel (`FZ`), environmental channel (`VitesseMoteur`). and **state channels** (e.g., `TC_BE_S TC_BE_M`, `DEP_D`, etc.). However, the **binary squeal indicator** (`isSquealing`) is not included as input for simulation; instead, it is dynamically generated by the prediction model (`predict_tf`) with the updated **state channels**.

   For the simulation during the first 20 time steps, the squeal indicator channels are initialized with `NaN` values. The simulation will replace these `NaN` values starting from the 21st time step, where the first prediction from `predict_tf` provides their initial values.

3. **Iterative Prediction Loop (Steps 5–10):** For every subsequent time step, starting from time step 21:
   - **Extract Recent Data (Step 6):** The algorithm extracts the last 20 time steps of input channels (`FZ`), environmental channels (`VitesseMoteur`), and previously predicted state variables (`X(t)`), which serve as the input window for the current iteration. This sliding window ensures the predictions are always based on the most recent data.
   - **Concatenate Inputs and Predict (Step 7):** The extracted inputs are concatenated and fed into the prediction model (`predict_tf`) to simulate the next time step. This model predicts:
     ▷ Updated state variables (`X(t+1)`), representing the system's simulated state at the next time step.
     ▷ Squeal prediction (`isSquealing`), indicating whether squeal is likely to occur at this time step.
   - **Split the Output (Step 8):** The prediction results are split into two components:
     ▷ Updated state variables (`X(t+1)`).
     ▷ Squeal prediction (`isSquealing`).
   - **Update Simulation Placeholders (Step 9):** The newly predicted data is appended to placeholders (`full_U_steps_simulated`, `full_E_steps_simulated`, `full_X_steps_simulated`, and `full_squeal_pred`). These placeholders store the cumulative simulation data over all time steps for later analysis.
4. **Iterative Behavior:** This process repeats until the simulation reaches the final time step. At each iteration:
   - Inputs for the prediction model include recent `FZ` and `VitesseMoteur`, alongside the previously predicted state variables (`X(t)`).
   - The model outputs the updated state variables (`X(t+1)`) and squeal prediction, which are used as feedback for subsequent predictions.
5. **Outputs and Analysis (Step 11):** After completing all iterations, the accumulated simulation data is compared with real data to validate the model. Plots are generated to visualize the input channel (`U`), environmental channel (`E`), state variables (`X(t)` and `X(t+1)`), and squeal predictions (`isSquealing`), ensuring the simulated trends align with the real-world observations.

To understand the structure and complexities of these pretrained models, we first present a summary of both models, detailing their expected inputs and generated outputs. Additionally, we ensure that the system frequency is consistently maintained at 90 Hz.

Algorithm 6 Integration Validation for Pretrained State Initializer and Squeal Predictor

Initialize the environment with:
- Path to the squeal prediction model.
- Path to the simulation model.
- `perform_normalization` set to `True`.

Extract key experimental data:
- Input channel (`FZ`).
- Environmental channel (`VitesseMoteur`).
- State channels (`TC_BE_S`, `TC_BE_M`, `DEP_D`, etc.).
- Binary squeal indicator (`isSquealing`).

Initialize placeholders for simulated predictions:
- `full_U_steps_simulated`.
- `full_E_steps_simulated`.
- `full_X_steps_simulated`.
- `full_squeal_pred`.

Set the first 20 time steps of real data as the initial input for the simulation.
for $j = 21$ to `total_time_steps` **do**
    Extract the last 20 time steps (`stride window`) for `U`, `E`, and `X`.
    Concatenate the inputs and pass them to `predict_tf` for the next prediction.
    Split the output of `predict_tf` into:
- Updated state variables (`X`).
- Squeal prediction (`isSquealing`).

    Update the simulation placeholders:
- `full_U_steps_simulated`.
- `full_E_steps_simulated`.
- `full_X_steps_simulated`.
- `full_squeal_pred`.

end for
Generate plots for comparative analysis of:
- Input channel (`U`).
- Environmental channel (`E`).
- State variables (`X`).
- Squeal predictions (`isSquealing`).

Confirm integration by validating alignment between real and simulated data trends.

4.8 Comparison Between predict and predict_tf

The choice between predict and predict_tf hinges on their performance differences, particularly in speed and efficiency. After testing both, we observed a significant advantage in using predict_tf, primarily due to TensorFlow's optimized graph execution.

4.8.1 Key Differences

1. Computational Efficiency: predict_tf utilizes TensorFlow's graph execution, which compiles and optimizes operations into an efficient computational graph. This reduces Python overhead and enables accelerated execution, especially on GPUs or other specialized hardware. In contrast, predict operates using NumPy's eager execution, processing each operation step-by-step without batch processing or GPU acceleration, resulting in slower performance for large-scale tensor operations.

2. Parallelism and GPU Utilization: TensorFlow's graph execution allows for parallelism and hardware acceleration. Operations such as tf.concat and tf.where are executed more efficiently within a graph. predict, relying on NumPy, is limited to CPU-bound operations, restricting its ability to leverage GPU resources for faster computations.

3. Model Inference Optimization: When using predict_tf, TensorFlow's computational graph integrates the model's operations seamlessly, optimizing the execution path and minimizing processing overhead. Conversely, predict involves Python's execution overhead and cannot utilize TensorFlow's internal optimizations, leading to less efficient model inference.

4. Data Manipulation Overhead: In predict, operations like np.concatenate and slicing are handled by NumPy, introducing additional memory usage and processing time. predict_tf, on the other hand, performs these tasks with TensorFlow tensors, which are optimized for such operations within the computational graph.

4.8.2 De-Normalization Considerations

An additional difference lies in how each method handles data manipulations, such as de-normalization. TensorFlow's graph execution ensures that transformations are performed as part of the optimized pipeline, reducing overhead. In predict, similar tasks involve standalone operations in NumPy, increasing processing time.

Conclusion: The optimized execution, GPU utilization, and reduced overhead make predict_tf the preferred choice for scenarios requiring high-performance inference, particularly when dealing with large-scale data or when GPU acceleration is available. However, predict may still be sufficient for smaller-scale tasks where computational efficiency is less critical.

4.9 Model Input and Output Shapes

4.9.1 Squeal Model

The squeal model takes into account 2000 time steps from the data, with the last 9 time steps specifically used as input for squeal prediction. These time steps include motor speed and 8 state variables (temperatures). The structure shown in 4.8 and input-output dimensions were deduced from running the model and reviewing the relevant documentation.

Input Shape: $(\text{None}, 9, 9)$

- The first dimension (`None`) represents the batch size.
- The second dimension (`9`) corresponds to the number of time steps included for prediction.
- The third dimension (`9`) represents the features used, including motor speed (1 feature) and state channels (8 features, primarily temperature-related).

Output Shape: $(\text{None}, 1)$

- The output is a binary value indicating the presence or absence of squeal.

4.9.2 Simulation Model

The simulation model shown in 4.9, processes 2000 time steps of data, with input structured into windows of 20 time steps. Each window includes 10 features, which are categorized into input channels (e.g., `FZ`), environmental channels (e.g., `VitesseMoteur`), and state variables (e.g., temperatures and displacements).

Input Shape: $(\text{None}, 20, 10)$

- The first dimension (`None`) represents the batch size.
- The second dimension (`20`) corresponds to the time steps in each input window.
- The third dimension (`10`) includes features such as input, environment, and state channels.

Output Shape: $(\text{None}, 8)$

- The output consists of 8 updated state variables predicted by the simulation model.

```
Extracted files and directories: ['EnvLille']
Model: "CNN_binary_class"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_35 (Conv1D)          (None, 5, 256)            11776

 conv1d_36 (Conv1D)          (None, 5, 256)            327936

 leaky_re_lu_35 (LeakyReLU)  (None, 5, 256)            0

 max_pooling1d_23 (MaxPoolin  (None, 2, 256)           0
 g1D)

 flatten_12 (Flatten)        (None, 512)               0

 dense_24 (Dense)            (None, 9)                 4617

 dense_25 (Dense)            (None, 1)                 10

=================================================================
Total params: 344,339
Trainable params: 344,339
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.8: Squeal model structure showcasing input and output shapes.

4.9.3 Summary of Model Structures

The key input and output shapes for the models are summarized below:

- **Squeal Model:**
  - ▷ Input Shape: $(\text{None}, 9, 9)$ - 9 time steps with 9 features each.
  - ▷ Output Shape: $(\text{None}, 1)$ - Binary squeal prediction.
- **Simulation Model:**
  - ▷ Input Shape: $(\text{None}, 20, 10)$ - 20 time steps with 10 features each.
  - ▷ Output Shape: $(\text{None}, 8)$ - 8 updated state variables.

```
Model: "FNN_class"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_8 (Flatten)         (None, 200)               0

 dense_32 (Dense)            (None, 120)               24120

 dense_33 (Dense)            (None, 120)               14520

 dense_34 (Dense)            (None, 120)               14520

 dense_35 (Dense)            (None, 8)                 968

=================================================================
Total params: 54,128
Trainable params: 54,128
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.9: Simulation model structure showcasing input and output shapes.

Note: The dimensions are carefully chosen to align with the architecture of the models and ensure compatibility with the corresponding preprocessing pipeline. These shapes form the foundation for both training and deployment phases.

The following graph 4.10 illustrates the results of running the setup created for one of the lab-recorded samples, alongside its simulated counterpart. Both were initialized using the same initial conditions as the lab-recorded sample to evaluate the simulation's capabilities before delving deeper into tailored testing scenarios.

4.10 Analysis of Increment Values for Motor Speed and Normal Force

To define suitable values for increasing or decreasing normal force and motor speed, which will later be used in creating the controller's action space using the DQN method, we evaluated various combinations of increments. Based on the generated heat map 4.11 and summary table 4.5, the analysis focused on identifying values that ensure smooth transitions, minimize system squeal, and avoid spikes that could destabilize the system. This process aimed to maintain system stability and achieve the desired control objectives.

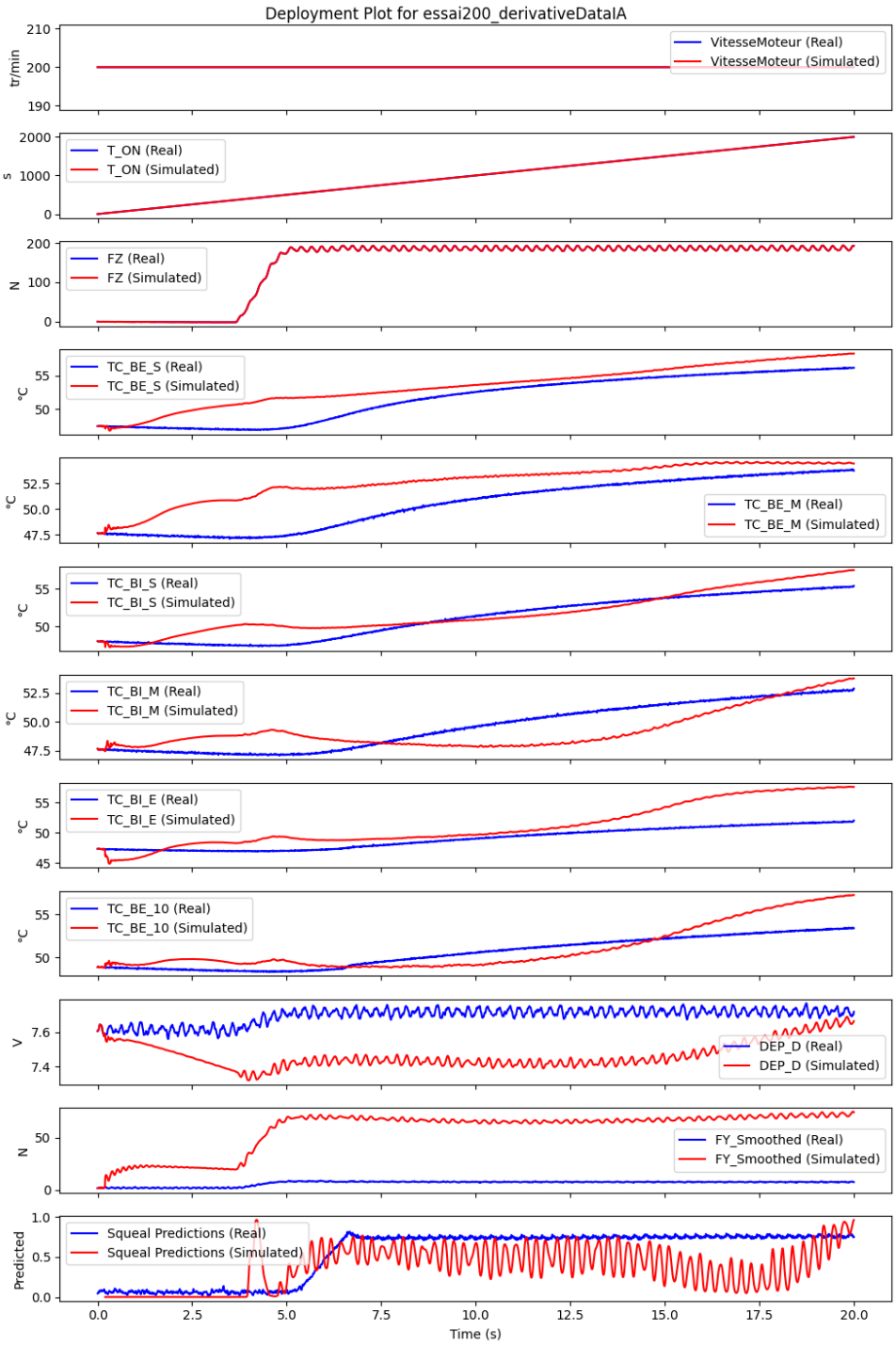


Figure 4.10: Real vs simulated model behavior

4.10.1 Preferred Values

The analysis revealed that increments of **5** and **10** provide the smoothest transitions with minimal squeal occurrences, making them highly suitable for achieving fine-grained control. Additionally, increments of **10** and **60** were evaluated for their ability to achieve faster adjustments while still maintaining stability, enabling a comparison of results.

4.10.2 Rationale for Selection

- Increments of **5** and **10** demonstrated the lowest squeal occurrences, indicating minimal disturbance to the system.
- Larger increments like **60** allowed for faster adjustments but introduced higher squeal occurrences compared to smaller increments, making them less favorable for precise control but useful for scenarios requiring rapid responses.
- A comparative analysis will be performed using **5, 10,** and **10, 60** to evaluate their impact on squeal minimization and system stability.

4.10.3 Summary of Results

Table 4.5 summarizes the average squeal values for various combinations of FZ increments and motor speed increments. The data supports the selection of **5, 10** and **10, 60** for comparative analysis.

Table 4.5: Summary of Squeal Results for Increment Combinations

FZ Increment	Motor Speed Increment	Squeal (Sum)
5	5	735.21
5	10	928.47
10	10	1373.35
10	60	821.27
60	10	1587.41
60	60	1850.54

4.10.4 Conclusion

The selected values of **5** and **10** (for finer adjustments) ensure system stability with minimal squeal occurrences, making them suitable for precise control. Larger increments such as **10** and **60** will be used for comparison, as they provide faster adjustments but come with higher squeal occurrences. The heat map visually supports these findings and highlights the impact of various combinations.

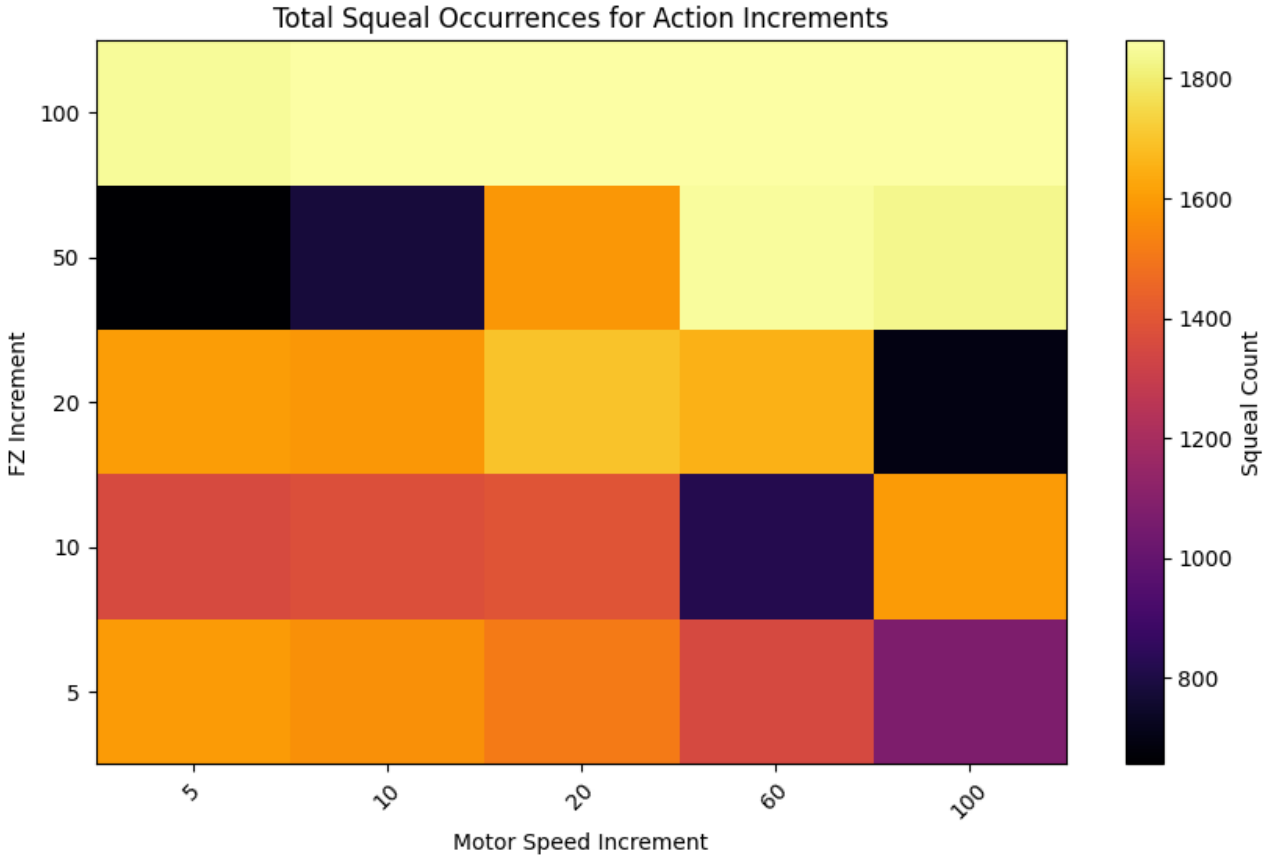


Figure 4.11: Heat map of squeal occurrences for different increment combinations

4.11 Continue Implementation: Controller Building

4.12 Deep Q-Network Architecture

The RL agent utilizes a Deep Q-Network (DQN) architecture designed to process a sliding window of 20 time steps. This is consistent with the validation step described earlier, which ensured that the simulation dynamics of the system could produce results similar to those recorded in the lab. In the following sections, we delve deeper into the architecture of the employed DQN model.

4.12.1 Model Structure

The DQN model used in this work has been carefully designed to learn optimal actions from sequential input data. Below, we provide a detailed explanation of its architecture and the reasoning behind each component:

Input Layer

- The model starts with an input layer that accepts data with a shape of `(20, 11)`. This corresponds to a sequence of 20 time steps, each consisting of 11 features. These features as discussed before, include critical variables such as motor speed, normal force, and system state variables that influence the system's dynamics.
- Using 20 time steps as the input ensures that the model captures temporal dependencies effectively. This sliding window approach allows the model to predict the next action based on a rolling history of the past data.

LSTM Layer

- The model incorporates a Long Short-Term Memory (LSTM) layer with 32 units. LSTMs are specifically designed to handle sequence data and learn temporal patterns, making them ideal for this application.
- The argument `return_sequences=False` ensures that the LSTM outputs only the final hidden state for the sequence, which is passed to the subsequent layers for action prediction.
- Increasing the number of LSTM units enhances the model's capacity to capture complex temporal dependencies, potentially improving its ability to predict optimal actions.

Dense Layers

- Following the LSTM layer, a fully connected (dense) layer with 32 units and a ReLU activation function is added. This layer transforms the LSTM output into a higher-level representation, enabling the model to learn non-linear relationships between input features and actions.
- The ReLU activation function introduces non-linearity, ensuring the model can learn complex patterns in the data while avoiding problems such as vanishing gradients which is a problem occurs when gradients become exceedingly small during backpropagation, making it difficult for the model to update its weights effectively. This issue is particularly common in deep networks and sequence models, where long-term dependencies need to be captured, but ReLU activation mitigates this by maintaining gradients in the positive range, thus supporting efficient learning."

Output Layer

- The final dense layer has a number of units equal to the `action_size`, representing the number of possible actions. This layer uses a linear activation function to output Q-values for each action.
- The Q-values represent the expected rewards for taking a specific action in a given state. Using a linear activation function allows the network to output unbounded values, which is standard for Q-value predictions in DQN models.

Model Compilation

- The model is compiled using the Mean Squared Error (MSE) loss function. This loss function measures the difference between the predicted Q-values and the target Q-values. Minimizing this loss helps the model align its predictions with the desired outcomes.
- Adaptive Moment Estimation (Adam) optimizer, with a specified learning rate, is used to ensure efficient gradient updates during training. Adam is chosen for its adaptive learning rate capabilities, which improve convergence speed and stability.

4.12.2 Hyperparameters and Their Role in Training the RL Model

The reinforcement learning model utilizes carefully tuned hyperparameters to guide the agent's learning process. These hyperparameters are tailored to the dataset, which consists of 2000 time steps and 11 features: `['FZ', 'VitesseMoteur', 'TC_BE_S', 'TC_BE_M', 'TC_BI_S', 'TC_BI_M', 'TC_BI_E', 'TC_BE_10', 'DEP_D', 'FY_Smoothed', 'isSquealing']`. The aim is to optimize the agent's ability to reduce squealing (`isSquealing`) while maintaining system stability in terms of thermal (`TC_BE_S`, `TC_BI_S`) and mechanical behaviors (`DEP_D`, `FY_Smoothed`).

Discount Factor (γ)

The **discount factor (γ)**, set to 0.99, ensures the agent considers long-term rewards, which is critical for:

- Balancing immediate actions like reducing `isSquealing` with the need to prevent long-term instability, such as overheating or excessive wear.
- Anticipating delayed impacts of adjusting `FZ` (normal force) or `VitesseMoteur` (motor speed) on the system's dynamics, such as thermal states (`TC_BE_S`) and tangential force (`FY_Smoothed`).

Learning Rate

The **learning rate**, set to 0.0005, controls how quickly the model updates its weights during training. A lower learning rate provides stability and prevents overfitting or overreaction to:

- Noisy squealing patterns.
- Sudden shifts in relationships between `FZ`, `FY_Smoothed`, and thermal states (`TC_BE_S`, `TC_BI_S`).

Exploration-Exploitation Strategy

The model employs an ϵ**-greedy exploration strategy** with:

- An initial ϵ of 1.0, encouraging broad exploration of the impact of `FZ` and `VitesseMoteur` across various scenarios.
- Gradual decay of ϵ to a minimum value of 0.01, allowing the agent to increasingly exploit learned policies for optimal performance.

This strategy is vital for:

- Identifying thresholds for squealing as `FZ` or `VitesseMoteur` varies.
- Optimizing long-term rewards by striking a balance between experimentation and leveraging learned behaviors.

Replay Buffer and Batch Size

- The **replay buffer**, with a capacity of 500 transitions, stores past experiences to ensure:
  - ▷ The model revisits critical high-squeal scenarios during training.
  - ▷ Generalization across a diverse range of transitions.
- The **batch size**, set to 20, determines how many transitions the agent learns from at each training step. This ensures the agent generalizes its learning across multiple scenarios rather than focusing on recent experiences.

Impact of Hyperparameters

These hyperparameters collectively ensure:

- The agent learns to minimize squealing (`isSquealing`) while maintaining system stability across thermal (`TC_BE_S`, `TC_BI_S`) and mechanical (`FY_Smoothed`, `DEP_D`) domains.
- Efficient exploration of `FZ` and `VitesseMoteur` settings to uncover the optimal balance for reducing squealing and ensuring stability.
- Robust policy development, enabling the agent to adapt to the intricate dynamics of the dataset.

This configuration allows the RL model to achieve the desired performance while maintaining stability across diverse operating scenarios.

4.12.3 Reward Function Design and Objectives

The reward function is designed to guide the reinforcement learning (RL) agent in minimizing squeal occurrence while ensuring system stability. This is achieved by directly regulating the normal force (`FZ`) and motor speed (`VitesseMoteur`), which are controlled by the RL controller. By adjusting these parameters, the controller indirectly influences the tangential force (`FY_Smoothed`), a critical factor in squealing behavior that is generated by a separate pretrained model.

Key Component: Squeal Penalty

The reward function prioritizes reducing squeal occurrence. A high reward is assigned when squealing does not occur (`isSquealing = 0`), while a penalty is applied otherwise. This aligns with the primary objective of minimizing brake squeal.

Challenges and Considerations

- **Nonlinear Relationships:** The relationships between normal force, motor speed, and tangential force are nonlinear. The reward function indirectly addresses these complexities by penalizing deviations in `FY_Smoothed`, without requiring explicit modeling of the nonlinear interactions.
- **Balancing Immediate and Long-term Rewards:** The design of the reward function ensures that the agent learns to balance immediate rewards (e.g., reducing squeal) with long-term stability. This is particularly important for maintaining system performance over extended periods of operation.

- **Fine-tuning of Parameters:** The coefficients in the reward function (e.g., the penalty weight for `FY_Smoothed`) were empirically tuned to achieve the desired balance. Future work could explore automated tuning methods or inverse reinforcement learning to refine these parameters further.

The reward function is a critical component of the RL framework, capturing the intricate dynamics of squealing behavior and system stability. By combining direct penalties for squealing with indirect penalties for tangential force deviations, the function effectively guides the agent toward optimal policies. The results of this approach, including its impact on squeal reduction and system performance, are presented in the subsequent sections.

4.13 Action Space and Environment

The action space consists of five discrete actions that allow the agent to adjust the braking system parameters. Based on the analysis and results derived from the heat map 4.11, two setups for the action space are considered:

4.13.1 First Action Setup

- Increase `FZ` and `VitesseMoteur` by 60 units.
- Increase `FZ` and `VitesseMoteur` by 10 units.
- Maintain current values of `FZ` and `VitesseMoteur`.
- Decrease `FZ` and `VitesseMoteur` by 10 units.
- Decrease `FZ` and `VitesseMoteur` by 60 units.

4.13.2 Second Action Setup

- Increase `FZ` and `VitesseMoteur` by 10 units.
- Increase `FZ` and `VitesseMoteur` by 5 units.
- Maintain current values of `FZ` and `VitesseMoteur`.
- Decrease `FZ` and `VitesseMoteur` by 5 units.
- Decrease `FZ` and `VitesseMoteur` by 10 units.

The two setups allow for comparative analysis between coarse-grained adjustments (10 and 60 units) and fine-grained adjustments (5 and 10 units). The choice of action setup is informed by the results from the heat map, balancing the need for system stability and squeal minimization.

The environment receives these actions and simulates their effects on the braking system using the pretrained simulation model. The RL agent observes the system state as a sliding window of the last 20 time steps, with 11 features per time step, capturing both input and state dynamics.

Markov Decision Process (MDP)

The RL framework adheres to the principles of an MDP:

- **State:** The state is represented by a concatenated tensor of normalized values for `FZ`, `VitesseMoteur`, state variables (`X`), and `isSquealing`, encapsulating the system's temporal and spatial dynamics.
- **Action:** Actions are selected based on the Q-values predicted by the DQN, reflecting the agent's policy for optimizing the reward function.
- **Reward:** The reward function penalizes squealing and deviations in `FY_Smoothed`, guiding the agent to prioritize squeal reduction while maintaining stability.
- **Transition:** State transitions are determined by the interaction between the RL agent and the simulation environment. The agent's actions update the directly controlled parameters (`FZ` and `VitesseMoteur`), which indirectly influence the tangential force (`FY_Smoothed`). The environment processes these updates and generates the next state and corresponding reward.

Replay Memory and Batch Training

Replay memory stores transitions of the form (`state`, `action`, `reward`, `next state`, `done`). This mechanism is crucial for:

- Breaking the temporal correlation in training data by shuffling experiences before training.
- Allowing the agent to revisit and learn from rare but important transitions, such as high-squeal scenarios.
- Enhancing sample efficiency, as each stored transition can be used multiple times during training.

The agent samples mini-batches from the replay memory to update the Q-values iteratively, ensuring robust learning.

Act Function

The `act` function governs the agent's behavior:

- Initially, the agent selects actions randomly (exploration) to discover the effects of different parameter combinations on squeal and system dynamics.
- Over time, the exploration rate (`epsilon`) decays, encouraging the agent to exploit its learned policy by selecting actions that maximize the expected reward.

Graphs and Their Insights

1. Reward Trends Across Episodes:. The reward plot (Figure 4.12) demonstrates the agent's cumulative and per-episode performance over 100 episodes. The cumulative reward shows a consistent and exponential increase, indicating the agent's ability to progressively improve its policy by minimizing squeal occurrences and optimizing system dynamics. However, the per-episode rewards remain relatively stable, which suggests that the learned policy focuses on maintaining steady performance rather than drastic variations in behavior.

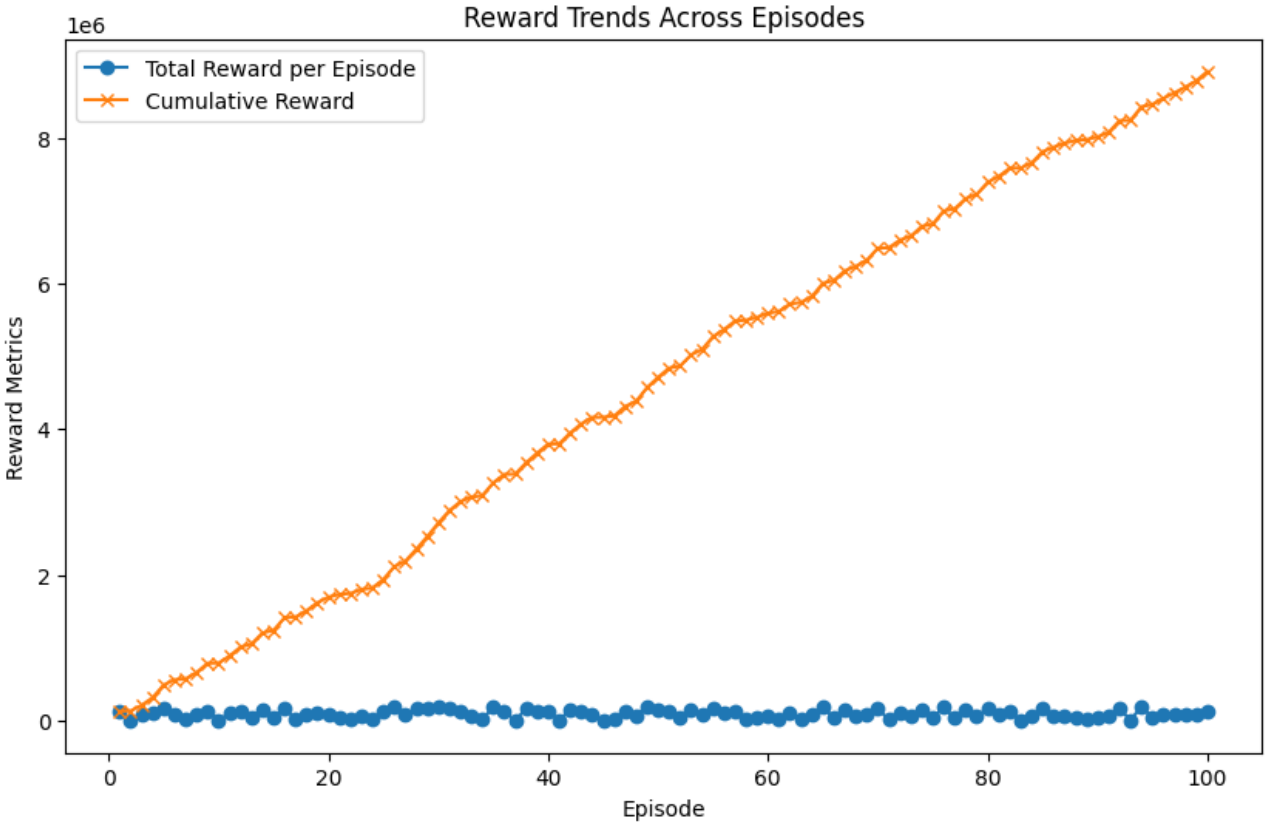


Figure 4.12: Total rewards per_episode and cumulative rewards

1: **Initialize DQN Agent:**

- Define state size $(20, 11)$ and action size.
- Initialize replay memory with a size of 500.
- Configure hyperparameters:
  - ▷ Learning rate: 0.0005
  - ▷ Discount factor: $\gamma = 0.99$
  - ▷ Exploration-exploitation balance: ϵ decaying from 1.0 to 0.01.

2: **Load Pretrained Weights and Memory:**

- Load model weights from `dqn_final_weights.h5`, if available.
- Load replay memory from `final_replay_memory.pkl`, if available.

3: **Prepare Sliding Windows for State Construction:**

- Use a sliding window approach of size 20 for the following:
  - ▷ Input channel (`FZ`).
  - ▷ Environmental channel (`VitesseMoteur`).
  - ▷ State variables (`TC_BE_S`, `DEP_D`, etc.).
  - ▷ Binary indicator (`isSquealing`).
- Concatenate the sliding windows to construct the state representation for the agent.

4: **Training Loop:**
5: **for** each episode e from 1 to $EPISODES$ **do**
6: Initialize the episode reward and action counters.
7: Randomly select a starting state from the sliding windows.
8: **for** each time step in the selected window **do**
9: **Action Selection:** Use ϵ-greedy policy to choose an action.
10: **State Transition:** Compute the next state using the selected action.
11: **Reward Calculation:** Use a custom reward function to compute rewards based on:

- Minimizing `isSquealing`.
- Reducing deviation in `FY_Smoothed`.

12: **Store Transition:** Append the state, action, reward, and next state to replay memory.
13: Update the current state to the next state.
14: **end for**
15: **Replay Training:**
16: **for** $i = 1$ to $TRAIN\_BATCHES$ **do**
17: Sample a minibatch from the replay memory.
18: Update the agent's weights using gradient descent.
19: Track the loss for each training batch.
20: **end for**
21: Append the total episode reward and cumulative rewards.
22: Save the updated weights and replay memory.
23: **end for**

2. Loss Trends Across Episodes:. The loss plot (Figure 4.13) highlights the Q-value prediction error throughout the training process. Initially, the loss increases as the agent explores the environment and adjusts its policy. Over time, the loss gradually decreases, showing convergence towards an optimal policy. The oscillations in the middle indicate instances of model updates where exploration and exploitation lead to fluctuations in learning. These oscillations stabilize in the later episodes, reflecting a balance between the Q-value predictions and the observed rewards.

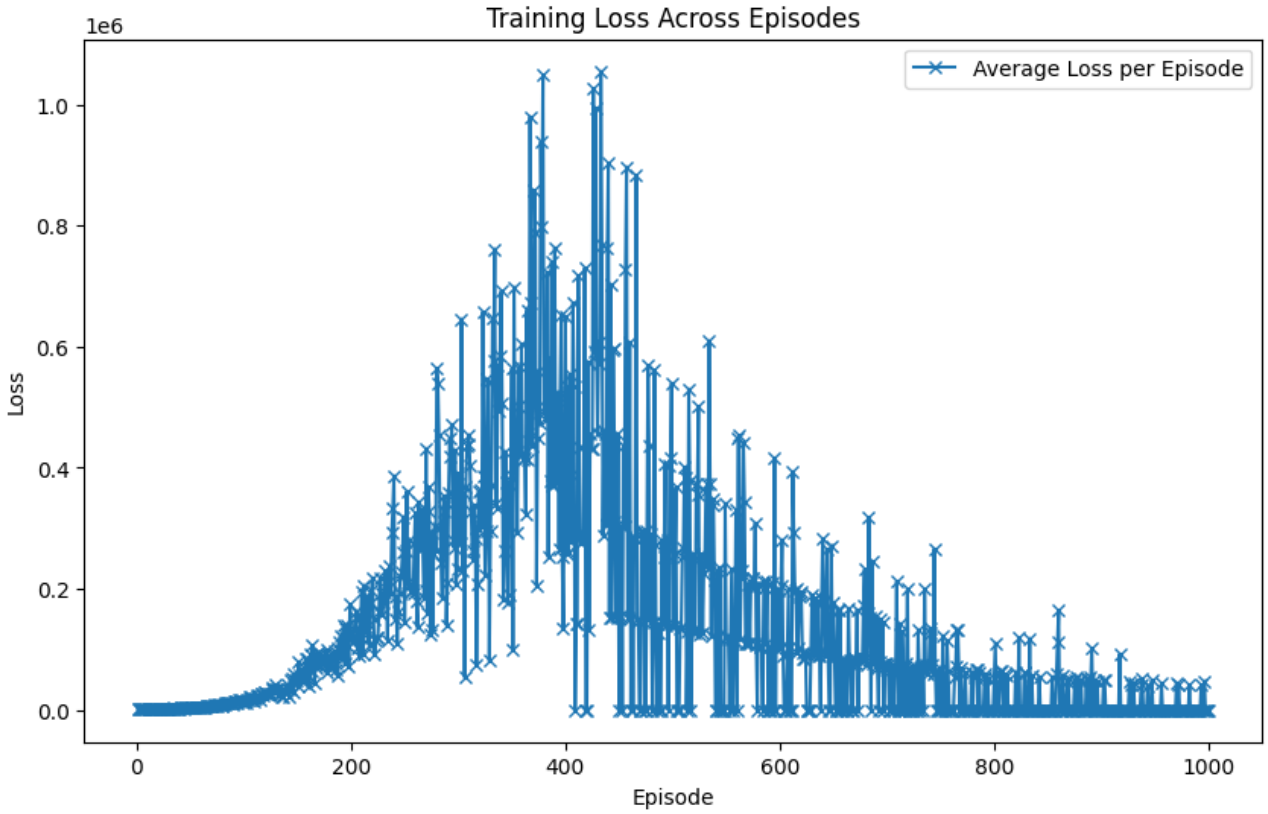


Figure 4.13: Training Loss Across Episodes: Average Q-value prediction error during training.

3. Actions Taken Per Episode:. The actions plot (Figure 4.14) illustrates the agent's decision-making patterns. The variability in actions taken during the early episodes reflects the ϵ-greedy exploration strategy, where the agent tests a wide range of actions to learn their consequences. In later episodes, the actions stabilize, indicating the agent's transition from exploration to exploitation as it converges to a well-defined policy.

Linking to Deployment

In the deployment phase, the RL agent's learned policy is applied to the braking system in real-time. The deployment code initializes with the first 20 time steps of real data as a kick-

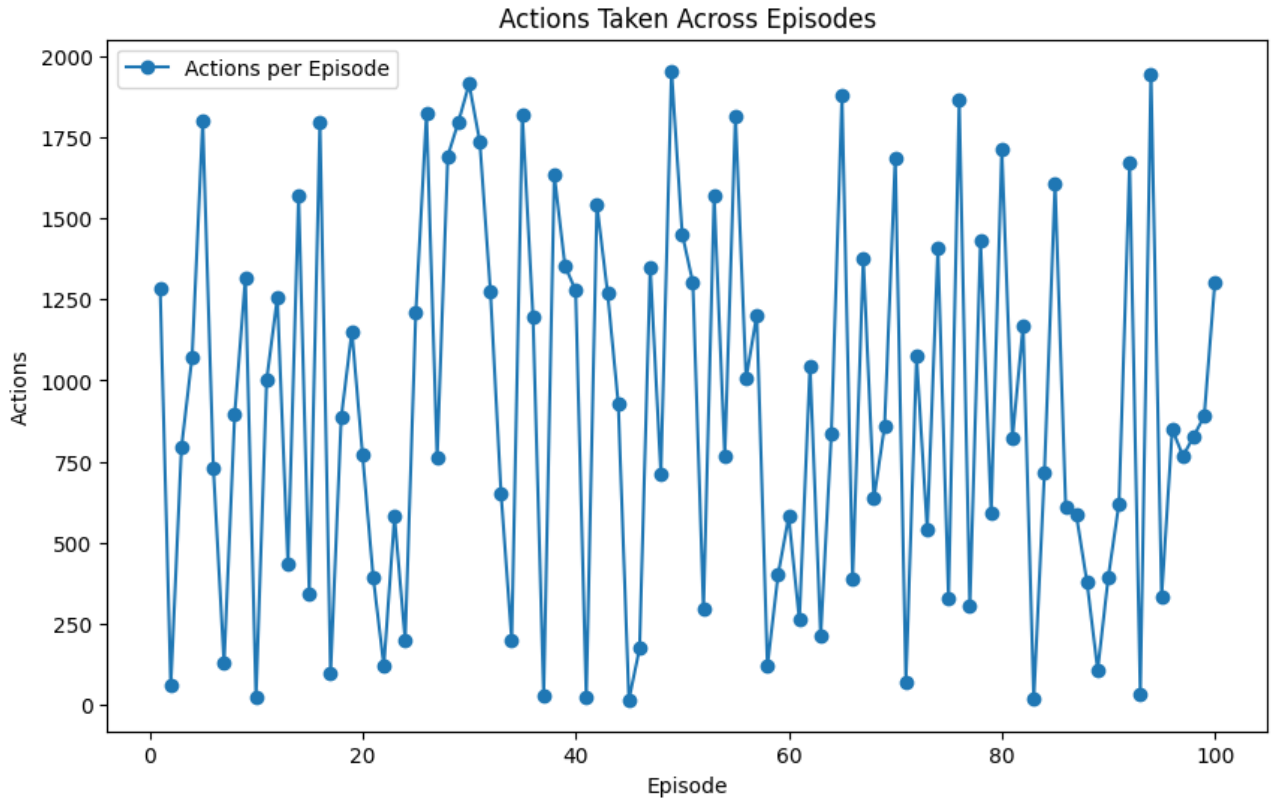


Figure 4.14: Actions Taken Per Episode: Tracking the number of actions performed by the agent per episode.

off and subsequently updates the system state using the pretrained simulation model. The agent normalizes the observed states, selects an action using its policy, and updates `FZ` and `VitesseMoteur` based on the action. This process is iteratively repeated, simulating the braking system's response to varying conditions and demonstrating the RL framework's practical applicability.

4.14 Deployment Analysis and Results

4.14.1 Model Behavior During Deployment

The deployment results, as illustrated in Figure 4.15 and

the following figures demonstrate the performance of the reinforcement learning (RL) agent in controlling the braking system dynamics using two distinct action setups.

- **Figure 4.15** represents the behavior of the RL agent under the first action setup, where increments of **10** and **60** units deduced from 4.11 are used for adjusting `FZ` and `VitesseMoteur`.

1: **Load Pretrained RL Model and Replay Memory:**

- Initialize the DQN agent with the state size and action size.
- Load the model weights from the specified path if they exist.
- Load the replay memory from the specified path if available.

2: **Initialize Data Structures:**

- Create placeholders for the simulation, including:
  - ▷ `full_U_steps_simulated_tf`: TensorFlow tensor for normal force (`FZ`).
  - ▷ `full_E_steps_simulated_tf`: TensorFlow tensor for motor speed.
  - ▷ `full_X_steps_simulated_tf`: TensorFlow tensor for state channels.
  - ▷ `full_squeal_pred_tf`: TensorFlow tensor for squeal predictions.
- Use the first 20 time steps from real data as kickoff values and update the tensors.

3: **Iterate Through Time Steps:**

4: **for** $j = 20$ to `total_time_steps` **do**

5: Extract the last 20 time steps (*stride window*) for `FZ`, motor speed, and state channels.

6: Concatenate the extracted features to form the simulation input.

7: Use the `predict_tf` function to predict:

- Updated state values for the state channels.
- Predicted squeal occurrence for the current time step.

8: **Action Selection:**

- Normalize the current state values.
- Use the DQN agent's `act` function to select an action.

9: **Update Inputs Based on Action:**

- Compute the average of the previous 20 time steps for `FZ` and motor speed.
- Modify the average values based on the action (e.g., increase, decrease, or maintain).
- Apply constraints to ensure the updated values stay within predefined limits.
- Update the tensors for `FZ` and motor speed.

10: **Update State and Squeal Predictions:**

- Update the state vector with the predicted state values.
- Flatten the squeal prediction to a scalar and update the squeal predictions tensor.

11: **Calculate Reward:**

- Use the `calculate_reward` function with the squeal prediction and smoothed tangential force.
- Append the calculated reward to the rewards list.

12: **end for**

13: **Post-Processing:**

- Store the updated tensors for further analysis.
- Generate visualizations to evaluate the model's performance.

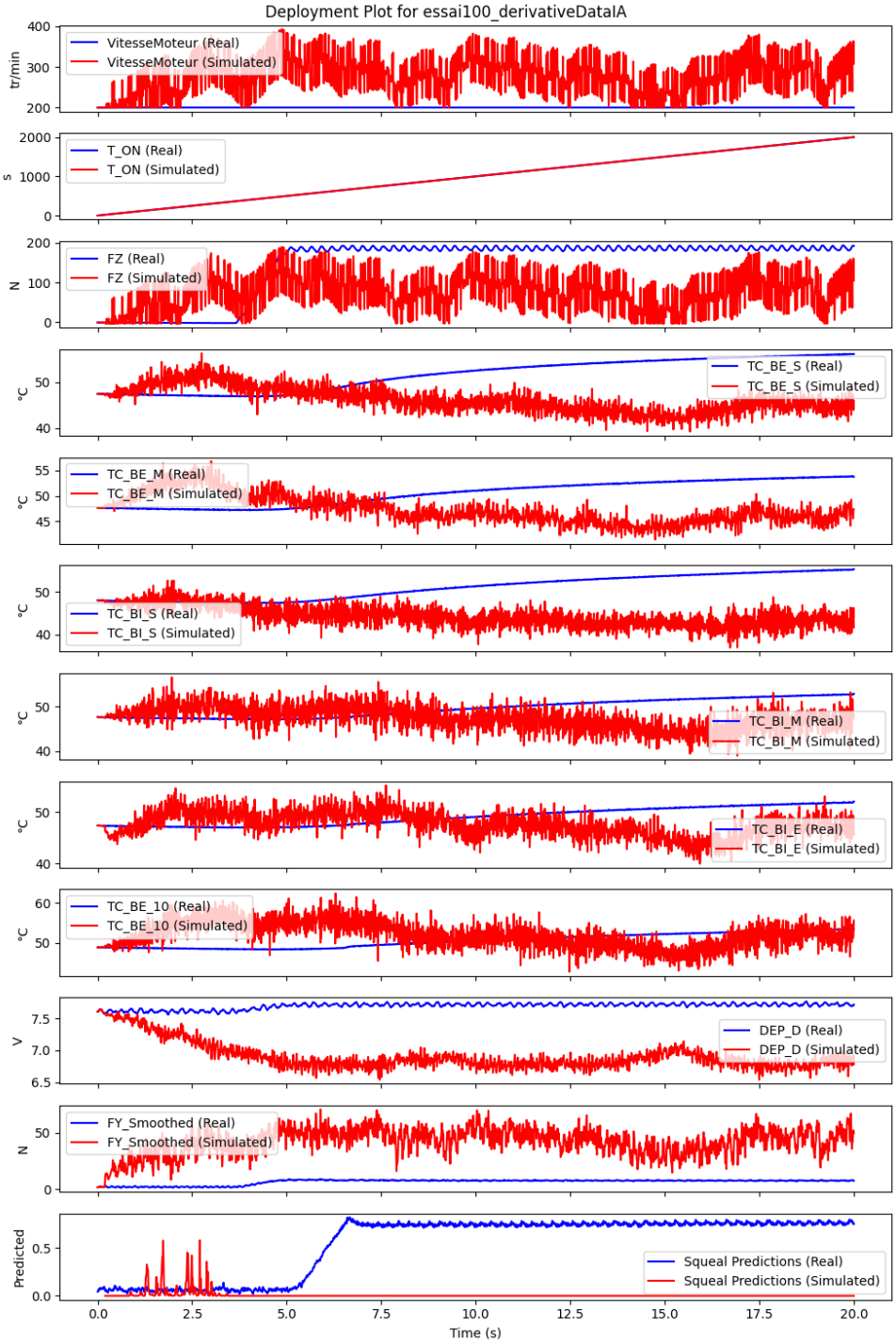


Figure 4.15: Deployment results using the 5 and 10 increment setup show smooth control and reduced squeal.

- **Figure 4.16** depicts the behavior of the RL agent under the second action setup, where finer increments of **5** and **10** units are employed for the same adjustments.

The plots show a comparison between the real-world recorded data and the simulated results produced by the RL model. By comparing the outcomes from both setups, we observe the trade-offs between coarse adjustments (10 and 60 units) and fine adjustments (5 and 10 units) in achieving system stability and minimizing squeal occurrences.

Key observations include:

- **Squeal Reduction:** The RL model effectively reduces squeal occurrences during the simulation, as evidenced by the significant decrease in the predicted squeal compared to the recorded data. This suggests that the model is achieving its primary objective of minimizing squeal within the deployment period.
- **System Stability:** Despite the reduction in squeal, the system dynamics, such as `VitesseMoteur` (motor speed), `FZ` (normal force), and thermal and mechanical state variables (`TC_BE_S`, `FY_Smoothed`, etc.), remain consistent with the behavior observed during the experimental recordings. This indicates that the RL agent maintains system stability while optimizing the braking system's performance.
- **Energy and Dynamic Behavior:** While the RL agent successfully reduces squeal, the plots reveal some signs of "violent movement" or abrupt changes in certain state variables. This could suggest that the reward function is introducing excessive energy into the system, potentially requiring refinement to balance squeal reduction and system stability more effectively.

4.14.2 Conclusion from Deployment

The deployment phase highlights the RL model's potential in effectively reducing squeal and maintaining system behavior within the experimental range. However, the observed oscillations and abrupt changes in the state variables emphasize the need for further refinement of the reward function. Addressing these issues will enhance the RL model's ability to deliver stable, energy-efficient, and reliable performance in real-world braking systems.

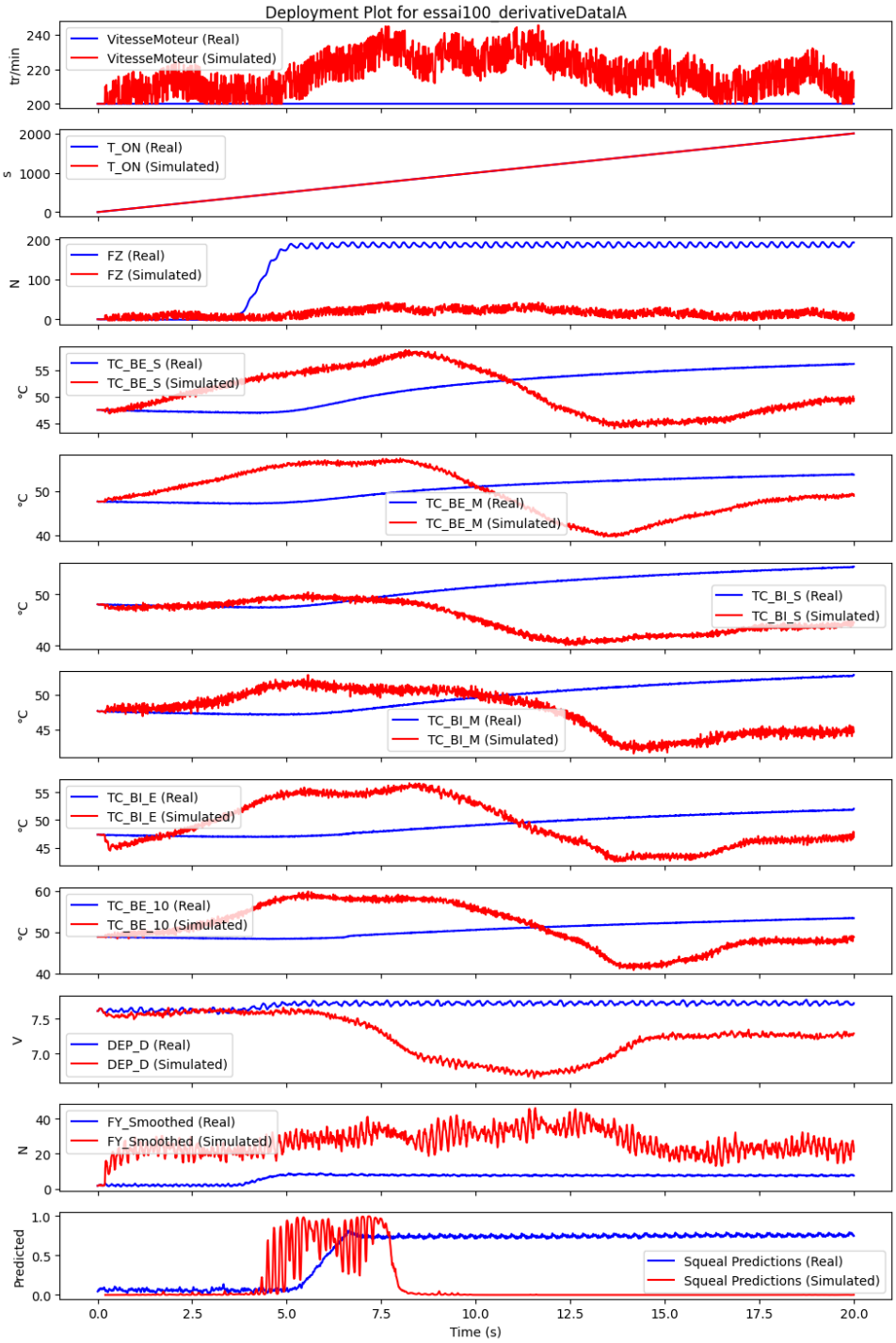


Figure 4.16: Deployment results using the 10 and 60 increment setup highlight faster adjustments with minor trade-offs.

Figure 4.17: Reward vs Time Steps During Deployment.

5 Summary of Findings and Perspectives for Future Work

5.1 Literature Review

The literature review offered a comprehensive understanding of brake squeal, its causes, and the extensive efforts made to mitigate it. It highlighted the evolution of both passive and active solutions, emphasizing the transformative role of data-driven approaches. These advancements have significantly enriched traditional methods, paving the way for dynamic solutions. This ongoing process aims to enhance control over the factors influencing brake squeal while ensuring optimal performance, safety, and comfort.

5.2 Summary of Research and Implementation

In this work, extensive research was conducted to identify and implement state-of-the-art methodologies for tackling the challenges of brake squeal control and system stability in reinforcement learning (RL) frameworks. The investigation began with a comprehensive review of machine learning and reinforcement learning techniques, focusing on their applications in dynamic control systems.

5.2.1 Reinforcement Learning and MDPs

Central to this study was the exploration of Markov Decision Processes (MDPs), which form the foundation for RL algorithms by modeling decision-making in environments with stochastic transitions and rewards. The research highlighted the evolution of RL, from foundational techniques like Q-learning and SARSA to more advanced deep reinforcement learning methods, particularly the Deep Q-Network (DQN) framework. This progression demonstrates the adaptation of neural networks to enhance the scalability and robustness of RL agents in complex environments.

5.2.2 Model Architecture and Reward Function

To ensure the implementation aligns with the latest developments, significant attention was given to selecting the most suitable RL architecture and reward function. The architecture incorporates Long Short-Term Memory (LSTM) layers to capture temporal dependencies, coupled with dense layers to enable non-linear representations. This design ensures the model

effectively learns the relationships between input states and actions, particularly for dynamic and noisy environments like braking systems.

The reward function was meticulously designed to balance competing objectives: reducing squeal occurrence and maintaining system stability. By prioritizing these goals, the reward function guides the RL agent's decisions effectively. The analysis of tangential force (`FY_Smoothed`) and its indirect control via normal force (`FZ`) and motor speed (`VitesseMoteur`) further shaped the model's development. This approach ensures decisions align with real-world operational constraints.

5.2.3 Simulation Environment and Pretrained Models

Pretrained models were integrated to enhance prediction accuracy and system understanding. providing a reliable foundation for the RL agent's training environment. The simulation environment was designed to replicate realistic conditions, ensuring the agent's learned policies are transferable to actual braking scenarios.

5.2.4 Contribution and Impact

Throughout the research, a balance was maintained between theoretical rigor and practical applicability. The chosen methodologies represent the culmination of state-of-the-art techniques, tailored to address the unique challenges of this application. By combining robust RL architectures, well-designed reward functions, and accurate simulation environments, the work demonstrates a clear commitment to advancing the field with practical, impactful solutions.

This project exemplifies the application of cutting-edge reinforcement learning techniques in a critical industrial context. It showcases how state-of-the-art methods can be adapted and optimized to solve complex, real-world problems, contributing not only to the field of RL but also to broader efforts in creating safer, more efficient, and quieter braking systems.

5.3 Recommendations for Future Work

While this work has successfully implemented a robust reinforcement learning framework for controlling brake squeal and maintaining system stability, there remain several opportunities for future research and development to further enhance the model's performance and adaptability:

5.3.1 Extended Training and Enhanced Data Utilization

Future efforts should consider extended training cycles to refine the RL agent's performance under diverse conditions. Additionally, incorporating a broader set of features that encompass the full range of data recorded during the test setup could provide the agent with richer insights, enabling it to make more informed decisions. By expanding the feature set, the model can potentially capture subtler dynamics that influence braking behavior.

5.3.2 Diverse Braking Scenarios

Introducing more variations in braking scenarios, including differences in brake pressing's longevity, force, and speed, within the operational boundaries, will improve the model's ability to generalize across real-world applications. Moreover, exploring braking scenarios that extend slightly beyond these boundaries may provide valuable insights into the system's behavior under extreme conditions, ensuring its robustness.

5.3.3 Exploration of Extended Scenarios and Reward Functions

To further validate and improve the system, future work should explore extended scenarios by varying parameters and testing alternative reward function designs. This exploration can help identify the most effective configurations and provide a more comprehensive understanding of the trade-offs between performance, stability, and squeal mitigation. Comparing results across these extended scenarios will also guide the refinement of the RL model for optimal performance.

5.3.4 Broader Context

While the current implementation demonstrates the feasibility and effectiveness of RL-based solutions for brake squeal control, these recommendations aim to build upon the existing foundation. By addressing these areas, future research can contribute to the development of more versatile and adaptive braking systems that align with evolving operational and safety standards.

6 Bibliography

Bibliography

[1] Ewins, D.J. (2000). *Modal Testing: Theory, Practice and Application.* Research Studies Press.

[2] Dufrénoy, P., et al. (2014). *Advanced Automotive Brake Systems.* Springer.

[3] Abouelazm, A., et al. (2020). *A Review of Data-Driven Approaches for Vehicle Dynamics and Control.* IEEE Transactions on Intelligent Vehicles, 5(4), 672-684.

[4] Geier, C., Hamdi, S., Chancelier, T., Dufrénoy, P., Hoffmann, N., and Stender, M. (2023). *Machine learning-based state maps for complex dynamical systems: applications to friction-excited brake system vibrations. Nonlinear Dyn,* 111:22137–22151. https://doi.org/10.1007/s11071-023-08739-6

[5] Gräbner, N., Schmid, D., and von Wagner, U. (2023). *On Drum Brake Squeal—Assessment of Damping Measures by Time Series Data Analysis of Dynamometer Tests and Complex Eigenvalue Analyses. Machines,* 11(12), 1048.

[6] Stender, M., Tiedemann, M., Spieler, D., Schoepflin, D., Hoffmann, N., and Oberst, S. (2019). *Deep learning for brake squeal: vibration detection, characterization, and prediction.* DeepAI.

[7] Fu, Y., Li, C., Yu, F.R., Luan, T.H., and Zhang, Y. (2020). *A Decision-Making Strategy for Vehicle Autonomous Braking in Emergency via Deep Reinforcement Learning.* IEEE Transactions on Vehicular Technology, 69(6), 5876-5889.

[8] Richard S. Sutton and Andrew G. Barto: *Reinforcement Learning: An Introduction. Second Edition.* MIT Press.

[9] Fadi AlMahamid and Katarina Grolinger: *Reinforcement Learning Algorithms: An Overview and Classification.* arXiv preprint arXiv:1606.01540, 2016.

[10] L.-J. Lin: *Self-improving reactive agents based on reinforcement learning, planning, and teaching.* Machine Learning, 8(3-4), 293–321, 1992.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver: *Prioritized experience replay.* arXiv:1511.05952, 2015.

[12] C. J. Watkins and P. Dayan: *Q-learning.* Machine Learning, 8(3-4), 279–292, 1992.

[13] G. A. Rummery and M. Niranjan: *On-line Q-learning using connectionist systems.* University of Cambridge, 1994, vol. 37.

[14] D. Zhao, H. Wang, K. Shao, and Y. Zhu: *Deep reinforcement learning with experience replay based on SARSA.* In IEEE Symposium Series on Computational Intelligence, 2016, pp. 1–6.

[15] H. Van Hasselt: *Double Q-learning.* In Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010, pp. 1–9, 2010.

[16] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas: *Dueling Network Architectures for Deep Reinforcement Learning.* In International Conference on Machine Learning, 4(9), 2939–2947, 2016.

[17] R. J. Williams: *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* Machine Learning, 8(3-4), 229–256, 1992.

[18] V. R. Konda and J. N. Tsitsiklis: *Actor-critic algorithms.* In Advances in Neural Information Processing Systems, 2000, pp. 1008–1014.

[19] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz: *Trust region policy optimization.* In International Conference on Machine Learning, 2015, pp. 1889–1897.

[20] R. Raileanu and R. Fergus: *Decoupling value and policy for generalization in reinforcement learning.* arXiv:2102.10330, 2021.

[21] Y. Liu, P. Ramachandran, Q. Liu, and J. Peng: *Stein variational policy gradient.* arXiv:1704.02399, 2017.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov: *Proximal policy optimization algorithms.* arXiv:1707.06347, 2017.

[23] Sutton, R. S., and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press.

[24] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). *Planning and acting in partially observable stochastic domains.* Artificial Intelligence, 101(1-2), 99-134.

[25] Bellman, R. (1957). *Dynamic Programming.* Princeton University Press.

[26] Kober, J., Bagnell, J. A., and Peters, J. (2013). *Reinforcement learning in robotics: A survey.* The International Journal of Robotics Research.

[27] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... and Hassabis, D. (2015). *Human-level control through deep reinforcement learning.* Nature, 518(7540), 529-533.

[28] Ahmed Abouelazm et al. *A Review of Reward Functions for Reinforcement Learning in the context of Autonomous Driving.*

[29] Sorg et al. *Informativeness of Reward Functions in Reinforcement Learning.*

[30] Malik, S. (2024). *Development of a Data-Driven Reinforcement Learning Environment for Reducing Brake Squeal in Braking Systems.* Dynamics Group, TUHH. Figure 4.1: Experimental setup and brake components.

[31] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... and Wierstra, D. (2016). *Continuous control with deep reinforcement learning.* arXiv preprint arXiv:1509.02971.

[32] OpenAI. (2024). ChatGPT: Language Model Assistance for Research and Writing. Retrieved from https://www.openai.com/.

[33] Charley, J. (n.d.). Spectrogram of the braking noise measurements. Retrieved from https://www.researchgate.net/figure/Spectrogram-of-the-braking-noise-measurements_fig8_258176633.

[34] Singh, A. (2019, July 18). *Introduction to Reinforcement Learning: Markov Decision Process.* Published in Towards Data Science. Retrieved from https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da.

[35] Szepesvári, C. (n.d.). *The Actor-Critic Architecture.* Retrieved from https://www.researchgate.net/figure/The-Actor-Critic-Architecture_fig2_220696313.

[36] An, Z. (n.d.). *A diagram of deep Q-network architecture.* Retrieved from https://www.researchgate.net/figure/A-diagram-of-deep-Q-network-architecture_fig1_353117089.

[37] Figure 1 - Change trend of exploration and exploitation from the beginning to the end. Available via license: Creative Commons Attribution 4.0 International. Retrieved from https://www.researchgate.net/figure/Change-trend-of-exploration-and-exploitation-from-the-beginning-to-the-end_fig1_324731872.

[38] Dunovan, K. (n.d.). *Flowchart describing agent-environment interaction.* Retrieved from https://www.researchgate.net/figure/Flowchart-describing-agent-environment-interaction_fig2_327882053.