Application of Reinforcement Learning in Electrical Machine Design

Yasmin SarcheshmehPour

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Helsinki 20.07.2023

Supervisor

Assoc. Prof. Alex Jung

Advisor

Dr. Victor Mukherjee

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Yasmin SarcheshmehPour

Title Application of Reinforcement Learning in Electrical Machine Design

Degree programme Computer, Communication and Information Sciences

Major Machine Learning, Data Science and Artificial Intelligence **Code of major** SCI3044

Supervisor Assoc. Prof. Alex Jung

Advisor Dr. Victor Mukherjee

Date 20.07.2023 **Number of pages** 102 **Language** English

Abstract

Electrical motors are essential components of many types of mechanical systems and are used in a wide range of applications, from appliances to industrial equipment. The utilization of electric motors and the associated systems is projected to account for approximately 43-46% of worldwide energy usage. Designing electrical machines, such as generators and motors, is a complex task that involves balancing a number of competing objectives, such as efficiency, cost, and reliability. Traditional optimization methods, such as gradient-based optimization and evolutionary algorithms, have been widely used to design electrical motors. However, these methods can be computationally expensive and may struggle to find the global optimum solution in complex design spaces.

Reinforcement learning (RL) is a rapidly growing field in machine learning, which has been proven to be an effective tool for optimizing complex systems. The approach of RL is based on an agent learning to interact with its environment to maximize the cumulative reward signal. RL has been successfully used in various fields like robotics, game playing, natural language processing and other fields.

This study examines the utilization of RL in the field of electrical motor design. A new approach based on RL is introduced to optimize the design of electrical motors, and its efficacy is demonstrated through the evaluation of various test cases. The thesis makes a noteworthy contribution by introducing a unique reward function tailored for the RL method. A comparison is made between the outcomes achieved by the proposed approach and those attained using traditional optimization methods. The potential advantages and difficulties associated with employing RL for electrical motor design are analyzed. The findings indicate that RL offers substantial enhancements in efficiency when searching for the optimal motor design within specified user requirements and manufacturing limitations, while also providing greater ease of use compared to traditional optimization techniques. Furthermore, potential directions for future research are outlined and discussed.

Keywords electrical machine design, artificial intelligence, reinforcement learning, policy gradient method, proximal policy optimization

Preface

I would like to express my sincere gratitude and appreciation to the individuals who have supported and encouraged me throughout my academic journey and the completion of this master's thesis.

I would like to begin by expressing my utmost appreciation to my thesis advisor, Victor Mukherjee, as well as my esteemed colleagues and fellow researchers, Tommi Ryyppö and Jan Westerlund. Their unwavering belief in my abilities, guidance, and valuable insights have been instrumental in shaping the direction and quality of this research. Their constant support, patience, and encouragement have been a source of inspiration throughout this work.

I would like to express my heartfelt gratitude to my thesis supervisor, Alex Jung, for his invaluable guidance, expertise, and unwavering support throughout my research journey. His mentorship and encouragement have been instrumental in shaping the development and success of this research endeavor. I am truly grateful for his dedication and commitment to my academic growth and research aspirations.

I am immensely grateful for the support and understanding of my family throughout my academic journey. Their unwavering encouragement, patience, and belief in my abilities have been a constant source of strength. I deeply appreciate their unconditional love and support, which have played a pivotal role in my accomplishments.

Helsinki, 20.07.2023

Yasmin SarcheshmehPour

Contents

Symbols and abbreviations

This thesis follows the notation mostly as in [1] and for the reader's convenience the relevant parts are repeated here.

a	action
A	set of all actions
A_t	action at t
$A(s)$	set of all possible actions in state s
$\mathcal{A}_t(s,a)$	The advantage function that is the difference between $Q_t(s,a)$ value for a given state - action pair and value function of the state $V_t(s)$
G_t	return (cumulative discounted reward) following t.
$P(s'\|s,a)$	probability of transitioning to state s' from s, a
$q_\pi(s,a)$	value of taking action a in state s under policy π
$q_*(s,a)$	value of taking action a in state s under the optimal policy
$Q_t(s,a)$	estimate (a random variable) of $q_\pi(s,a)$ or $q_*(s,a)$
R	set of possible rewards
R_t	reward at t, dependent, like S_t, on A_{t-1} and S_{t-1}
s	state
S	set of all states
S_t	state at t
t	discrete time step
T	final time step of an episode
$v_*(s)$	value of state s under the optimal policy
$v_\pi(s)$	value of state s under policy π (expected return)
$V_t(s)$	estimate (a random variable) of $v_\pi(s)$ or $v_*(s)$
π	policy, decision-making rule
$\pi(s)$	action taken in state s under deterministic policy π
$\pi(a\|s)$	probability of taking action a in state s under stochastic policy π

List of Figures

List of Tables

1 Introduction

1.1 Motivation

An electrical machine is a device that converts electrical energy into mechanical energy or vice versa. We need to design customized electrical machines to meet specific requirements and needs, such as high efficiency, low cost, reliability, and safety. Electrical machines are essential to modern life, from powering our homes and offices to driving our cars and trains. Effective design of electrical machines increases their performance, reduces energy consumption, and improves their overall longevity. The design and manufacturing process of electrical machines requires specialized knowledge and skills in electrical engineering, manufacturing processes, materials science, and computer-aided design.

The process of designing an electrical machine involves several steps, beginning with identifying the necessary requirements such as the type of machine, operating conditions, power output, and efficiency. The next stage involves choosing the appropriate electrical and mechanical components such as rotor, stator, bearings, and cooling systems. Afterward, the electrical and magnetic characteristics are calculated, including the number of turns, magnetic core size, conductors, and winding configuration. The mechanical structure is then developed while considering the size, shape, and used materials. The final stage involves refining the design by optimizing it based on testing to increase efficiency and performance. The field of electrical machines is constantly evolving with new technologies and innovations being developed every day.

Designing electrical machines is a challenging task as it requires balancing various factors and trade-offs. One example of this is the relationship between power output and material usage, where higher power requirements may result in greater material usage and therefore increased cost. On the other hand, if the machine is designed to be smaller, it may generate more heat. Utilizing RL can help resolve these trade-offs by automating the design process. The RL algorithm can learn from trials and errors as well as from its mistakes to find the optimal design that balances all the relevant objectives. This approach is more efficient than traditional design methods, where engineers manually evaluate different design options to determine the best solution, as it saves time and resources.

RL is a type of machine learning that focuses on making decisions in a sequential manner in order to maximize a reward signal. In electrical machine design, RL is used to improve performance metrics such as efficiency, power density, and torque density. The goal of RL in this context is to find the optimal design that balances these multiple objectives through an automated optimization. The algorithm learns from its experiences and adjusts its decision-making strategy accordingly in order to achieve the highest reward. The design process can be made more efficient by minimizing the time and effort needed to manually search for the most optimal design.

Additionally, RL can also be used to optimize the design of electrical machines for specific applications. To illustrate, a machine intended for utilization in a wind turbine must prioritize achieving low weight and high torque density. On the other hand, a machine intended to be used as a power generator on a cruise ship must prioritize meeting size constraints and ensuring high-quality voltage output. Therefore, RL can be used to find the best design for each specific application. Another use case of RL in electrical machine design is for dynamic optimization, which enables the real-time optimization of machine performance by adapting to changes in different conditions. For instance, it can be used to adjust the performance of a wind turbine in response to variations of wind speed.

Furthermore, the application of RL in electrical machine design can provide numerous benefits, including: automating the design process, optimizing performance for specific applications, and improving the control of the machines.

1.2 Related work

In recent years, RL has gained increasing attention in the field of electrical machine design due to its ability to handle complex design challenges with large state and action spaces. Despite this growing interest, there is limited research in the area of RL-based electrical machine design. This chapter provides a comprehensive review of the current state of the art in machine learning-based and traditional optimization methods for the electrical machine design problem by examining the published works of the past few years. The existing literature can be divided into two main categories: genetic algorithms (GA) and RL methods.

The paper [4] presents a method for optimizing the design of three-phase induction motor in manufacturing process using a GA. The objective functions of this work are torque, efficiency, and cost. The method involves adjusting design parameters, such as the number of stator and rotor slots, the diameter of the rotor, and the thickness of the magnetic core, to achieve improved performance. The GA is used to search the design space for the optimal combination of design parameters. The method is demonstrated by applying it to an induction motor design problem, and the authors show that it results in improved performance compared to an existing motor having the same ratings.

The same as the previous work, [5] presents a design optimization of a three-phase induction motor using a genetic algorithm (GA). However, their usage is more specific and the induction motor employed in this work has an external rotor. The goal of the optimization is to find the optimal dimensions for the machine components to maximize its efficiency. The design optimization process involves adjusting design parameters, such as the number of stator and rotor slots, the shape of the slots, and the number of turns in the windings, to achieve improved performance. The results show that the GA-optimized design improved the efficiency of the machine compared to a traditional design method.

The above mentioned works [4, 5] demonstrate that genetic algorithms are a good choice for designing electrical machines. However, they are usually time-consuming and computational intensive as they require a large number of function evaluations and simulations to converge to an optimal solution [6]. Moreover, genetic algorithms have the tendency to overfit to the training data, leading to a loss of generalization ability and reduced performance in the real-world scenario [7]. Additionally, genetic algorithms do not guarantee an optimal solution and may not always find the best solution and can sometimes give unreliable results. It's hard to predict what the outcome will be, and the algorithm might stop at a suboptimal solution instead of the best one.

The only study that has employed RL in the context of electric machine design is documented in [8]. This study uses RL and evolutionary optimization to create a linear induction motor. The process starts with a random design, which is tested to see how well it performs. The results are then used to update the design using RL, which helps make it better. After that, the updated design is used for evolutionary optimization to explore the design space and find the best design. The process is repeated until a satisfactory performance is achieved or a predetermined stopping criterion is met. The authors demonstrate the effectiveness of the method by applying it to the design problem of a motor and show that it outperforms traditional optimization techniques. The proposed method is employed to create a linear induction motor, and the data from this process is saved to be used for future designs. If the requirements of a new design are similar to previous designs, this method can quickly suggest suitable designs. Even for new requirements, the method can generate more data to adapt to the new design. However, it is not a universal solution that can be applied to any induction motor design.

Additionally, there have been numerous works that explore the application of machine learning techniques in electrical machine condition monitoring. These studies aim to leverage the capabilities of machine learning algorithms to enhance the detection, diagnosis, and prediction of faults in electrical machines. AS an example, [9] presents a comprehensive study on the application of machine learning techniques, including SVM and decision tree algorithms, for fault detection and diagnosis of induction motors. The authors use motor current signals as input features and achieved accurate fault classification and diagnosis. Moreover, [10] focuses on the fault diagnosis of induction motors using machine learning techniques. The authors extract statistical features from motor current signature analysis (MCSA) data and employ a random forest algorithm for fault classification. The proposed method achieves high accuracy in diagnosing various motor faults.

In conclusion, designing electrical machines poses challenges due to nonlinear relationships between design variables and performance metrics. Multiple optimal solutions or trade-offs between conflicting objectives further complicate the search for global or near-global optima. Additionally, the design process involves numerous variables (e.g., geometries, materials, winding configurations), leading to a high-dimensional search space that slows down efficient identification of optimal designs. The evaluation of performance through computationally intensive methods like electromagnetic simulations or finite element analysis adds to the time and cost required for optimization, potentially limiting its feasibility.

RL has demonstrated its potential to tackle intricate design problems characterized by high-dimensional state and action spaces. Additionally, RL algorithms can effectively handle complex and non-linear relationships between design variables and performance metrics by learning the underlying patterns and optimizing the design based on feedback and rewards. The literature review also shows that machine learning and RL has been successfully applied to various aspects of electrical machine design and control. However, the application of RL in the design of electrical machines has received limited attention, leaving sufficient room for improvement. Therefore, further research is necessary to fully exploit the potential of RL in this field. To the best of the author's knowledge, this study represents the first investigation into the application of RL as a design automation approach for rotating electrical machines. This model is developed for sales tool, however, engineers can also use it to obtain an excellent initial machine design.

1.3 Scientific contributions

This master thesis explores the application of RL to electrical machine design. One of the key challenges in applying RL to machine design is the design of an appropriate reward function that accurately captures the desired behavior of the machine. The main contribution is a novel approach to designing reward functions, which is based on a combination of performance metrics and cost functions. This novel reward function is designed to find the proper machine design in the least amount of design evaluations.

By combining performance metrics, such as machine nominal current, torque, stator temperature rise, and airgap flux density, with cost functions, such the mass of the motor, the proposed approach optimizes the overall performance of the machine while also considering the economic feasibility of the design. This not only leads to better machine performance but also reduces manufacturing costs, making the design more accessible and practical. Furthermore, this approach can be adapted to different operating conditions and design constraints, such as varying load conditions, input voltage, and material properties. This makes it a flexible and adaptable approach that can be applied to a wide range of machine design tasks.

The literature review conducted in this thesis identifies the key challenges and opportunities in electrical machine design. In contrast, the inherent capabilities of RL, such as handling complexity, multi-objective optimization, adaptability, and automation, make it a promising approach for designing electrical machines, surpassing the limitations of traditional optimization methods. Furthermore, the significance of developing suitable reward functions within the RL framework is well understood. The proposed approach to designing reward functions is demonstrated through a series of experiments on a range of electrical machines, including different induction motors. It is shown that this approach is effective in optimizing machine performance and can be adapted to different operating conditions and design constraints. In addition, the performance of the proposed approach is evaluated in comparison to traditional optimization techniques. The results demonstrate that this approach outperforms existing algorithms in terms of speed, and convergence to optimal solutions.

The potential applications of RL in electrical machine design are also discussed, highlighting the potential for significant improvements in the computational time. Future research directions in this area are also identified, including the integration of RL with other machine learning techniques and the development of RL algorithms for more complex machine design tasks.

1.4 Thesis outline

The rest of this work is structured as follows. Section 2 presents the background including the concepts of electrical machines and electrical machine design. Also, Section 2 explains the basics of RL that is the method used in this thesis. Then, Section 3 presents the problem formulation including our electrical machine design game, the proposed reward functions. Additionally, a comprehensive description of the RL method utilized in this thesis is presented in Section 3. In Section 4, the proposed algorithm is tested on various induction machines with different stages in order to address the research questions posed. The experiments conducted provide valuable insights and findings on the effectiveness of the proposed approach. Section 5 concludes this thesis and presents potential avenues for further research in the field of RL applications in electrical machine design.

2 Background

This section provides a comprehensive overview of the two main components that form the basis of the study: electrical machines and RL.

The electrical machine subsection (section 2.1) highlights the significance of electrical machines in various industries and their crucial role in energy conversion. It covers essential topics such as different types of motors, their elements and properties, and performance evaluation metrics. This subsection establishes the foundational knowledge necessary to understand the complexities involved in electrical machine design. The RL subsection (section 2.2) introduces the field of RL. It explains the fundamental principles and concepts of RL, including markov decision process and the Bellman equation. The subsection also explores different types of RL algorithms, such as on-policy and off-policy algorithms, Monte-Carlo and temporal difference methods, that can be employed to tackle optimization problems in electrical machine design.

By combining the knowledge from the electrical machine and RL subsections, the thesis aims to explore the use of RL techniques in optimizing electrical machine design. The goal is to leverage the adaptive and learning capabilities of RL algorithms to autonomously discover optimal designs for electrical machines.

2.1 Electrical machine

An electrical machine is a device that converts electrical energy into mechanical energy or vice versa. There are two main types of electrical machines: generators and motors. Generators are devices that convert mechanical energy into electrical energy. They work by using a rotating shaft to turn a coil of wire within a magnetic field. This causes a current to flow in the wire, producing an electrical voltage. Generators are commonly used in power plants to produce electricity for distribution to homes and businesses.

Moreover, motors are devices that convert electrical energy into mechanical energy [11]. They employ an electric current to produce a magnetic field that propels a rotor consisting of a metal core. The part called the rotor spins inside a magnetic field made by the stator. This makes the rotor turn and spin the output shaft of the motor. Such rotation is useful in energizing mechanical tools like fans, pumps, and conveyor belts. Motors are commonly used in appliances and industrial equipment. Induction motors and synchronous motors are two types of electrical motors that are commonly used in various industrial and commercial applications.

A synchronous motor is a type of AC motor where the rotor rotates at the same speed as the stator's rotating magnetic field. The rotor is connected to the power source, typically through a brushless exciter. Synchronous motors are used in applications that require high power, such as in cruise ship propulsion, hoist that is used in mining. They are typically more expensive and have more complex construction than induction motors.

On the other hand, an induction motor, also known as an asynchronous motor, is a type of AC motor where the rotor rotates at a slightly slower speed than the stator's rotating magnetic field. The rotor is not connected to the power source and instead, it is induced to rotate by the stator's magnetic field. Induction motors are typically less expensive, durable and have a simpler construction than synchronous motors. They are widely used in fans, pumps, and other industrial applications. In this work, the focus will be placed on induction motors.

2.1.1 Elements of induction motor

An electrical machine, such as an electric motor or generator, typically has several key elements (as shown in Figure 1).

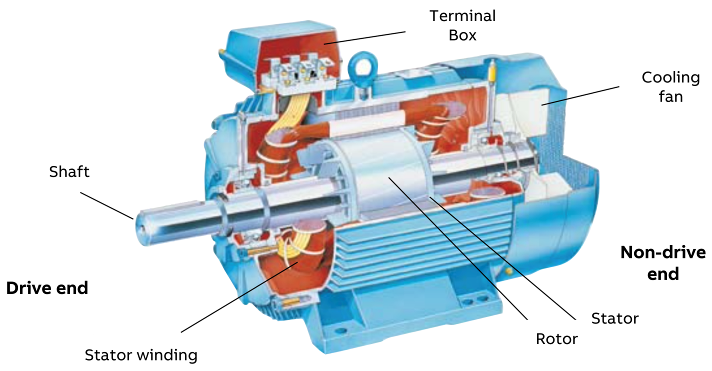


Figure 1: Elements of induction motor.

The *stator* is the stationary part of an induction motor. It typically consists of a frame and iron core that holds a set of windings, which are used to create a magnetic field. The stator windings are connected to a power source and, when energized, produce a magnetic field that penetrates the rotor. The stator is also responsible for providing support for the rotor and the bearings that hold it in place.

The *rotor* is the rotating part of an induction motor. It typically consists of a core that is made up of laminated insulated steel sheets, which is used to reduce eddy currents. The rotor core has windings or conductors, which are subject to the magnetic field created by the stator. The interaction between the stator and rotor magnetic fields causes the rotor to rotate, generating mechanical power. The rotor is supported by bearings, which allow it to rotate freely inside the stator.

The *bearings* are mechanical connections between rotating and static parts that help keep the rotating part stable. They're found between the stator and the rotor, and their job is to decrease friction, limit wear and tear, and give support to the rotor. This helps the rotor turns stably by minimizing frictional losses. Bearings are critical components of an induction motor, as they play a key role in maintaining the alignment and stability of the rotor. They must be properly lubricated and maintained to ensure that the motor runs efficiently and reliably.

The rotating element that transmits power from an induction motor to the load is known as the *shaft*. This component is linked to the rotor, which is the rotating section powered by the magnetic field created by the stator, a fixed part of the motor.

Usually, the shaft extends from both sides of the motor and is supported by bearings that enable smooth and low-friction rotation. Additionally, the shaft is employed for mounting pulleys, gears, or other mechanical tools that help in transmitting power to the load.

Cooling system in an induction motor is used to dissipate the heat generated by the motor during operation. This is typically accomplished by circulating a coolant, such as air or liquid, through the motor to remove heat from the windings and other components. The type of cooling system used in an induction motor can vary depending on the application and the size of the motor. Some common types of cooling systems include forced air cooling, liquid cooling, and fan cooling.

Terminal box in an induction motor is a protective enclosure that houses the electrical connections for the motor. It typically contains terminals for the power supply and control wires. The terminal box is typically located on the side or top of the motor and is designed to be easily accessible for maintenance and repairs. It protects the internal electrical connections from dust, moisture, and other environmental factors.

2.1.2 Properties of electrical machine

The characteristics of an electric motor can be classified into various categories that are interconnected and can substantially affect the machine's effectiveness, efficiency, and longevity. these characteristics can be categorised into three main groups, requirement variables, design variables, and performance variables. Within the scope of this thesis, specific design variables and performance values have been selected for examination, as they are directly relevant to the research objectives. While there exists a multitude of design variables and performance values, this study narrows its focus to those that are specifically applicable to the context of this thesis.

Requirement variables

Power output in an induction motor is the rate at which electrical energy is converted into mechanical energy. It is measured in watts (W) or horsepower (hp). The power input to the motor (also known as the "input power") is the electrical power supplied to the motor's stator, while the power output from the motor (also known as the "output power") is the mechanical power produced by the rotor. The difference between the input power and the output power is the power lost as heat due to various factors such as friction, iron losses, and winding losses.

Voltage in an induction motor refers to the electrical potential difference between two points in the circuit, for example, two phases of stator winding. The voltage applied to the stator winding can be either single-phase or three-phase, depending on the design of the motor. The voltage level and frequency must be within a specific range in order to ensure proper operation of the motor.

Relative starting current in an induction motor refers to the ratio of the current drawn by the motor at the time of starting to its rated full-load current. It is expressed as a percentage and is used to describe the level of current motor draws from network during locked rotor condition. This current is typically several times higher than the normal running current of the motor. The value of relative starting current depends on various factors, including the size and type of the motor, the method of starting, and the power supply system. The relative starting current can impact the sizing of electrical supply and protective devices and should be taken into consideration in motor selection and application design.

Design variables

The *number of turns in the stator winding* is chosen to provide the required magnetic field strength and flux linkage for the motor's rated voltage and operating frequency. The specific number of turns in the winding will also depend on the desired operating characteristics of the motor, such as efficiency, torque, and speed.

The *core length* of an induction motor refers to the length of the iron core that forms the stator and/or rotor of the motor. The core length is an important design parameter that affects the motor's performance characteristics such as efficiency, power output, and torque. The specific core length of an induction motor can vary widely depending on the motor's design and application. Generally speaking, larger motors will have longer cores than smaller motors, but there are many other factors that can affect the core length as well. The specific core length will depend on factors such as the motor's power rating, speed, efficiency, and torque requirements, as well as the size and shape of the motor's housing or enclosure.

Rotor tooth tip is the end of the rotor teeth in a cage rotor construction. It's exposed to the air gap between the stator and rotor, and has a significant impact on the motor's starting torque, maximum torque, and efficiency. The air-gap is a critical dimension in motor design, and should be kept as small as possible without causing damage to the motor. The shape of the tooth tip also affects performance, with a rounded tip resulting in higher efficiency and a sharp tip resulting in lower efficiency. Proper design of the rotor tooth tip and air-gap is essential for optimal motor performance.

Performance values

Power factor in an induction machine refers to the phase shift between voltage and current. Real power is the power that is actually used to do work, while reactive power is generating the magnetic field in the motor. The power factor is important in induction machines because it affects the efficiency of the machine, and can also impact the operation of the electrical grid to which the machine is connected.

Torque in an induction motor refers to the turning force that causes the rotor to rotate. It is the rotational equivalent of force and is measured in Newton-meters ($\mathrm{N \cdot m}$) or pound-feet ($\mathrm{lb \cdot ft}$). It's proportional to the current in the stator winding and the magnetic field strength in the air gap. Motor design factors, like pole number, cage construction, and winding configuration also influence torque. Torque varies with speed and is described by a torque-speed curve.

Nominal current is the current at which an induction motor is designed to operate during normal conditions. It's also called rated or full-load current, and determined by the motor's design and construction, such as the winding and wire turns. Nominal current flows through the stator winding at rated power and speed. It's specified on the motor nameplate along with other information. Operating the motor beyond its rated current, voltage, or frequency can cause damage to the windings and other components and lead to motor failure.

Airgap flux density is the strength of the magnetic field between the stator and rotor in an induction motor, affecting the torque produced by the motor. It's measured in Tesla or Gauss and determined by factors like stator current and winding design. Higher flux density leads to higher torque but also higher losses and temperature. Rotor speed also affects it. Proper range of flux density is important to avoid saturation of the stator core and ensure motor operation.

Stator temperature rise in an induction motor is the increase in temperature of the stator windings and core due to losses in the motor. It's measured in Kelvins and affected by factors such as current flow, motor load, cooling efficiency, and environment. High stator temperature rise can reduce motor efficiency and power factor, degrade insulation, and cause failure. To prevent this, the motor must be designed to handle expected loads and operated within rated limits, and the cooling system must be effective.

The *Efficiency* of an induction motor is the ratio of output power to input power, and is affected by factors like motor design, load, power factor, current and temperature. Efficiency is highest at full load and nominal speed, and can be improved by using high-quality materials, optimizing motor design, and matching load and power factor. Efficiency is important for energy consumption and cost of operation, and a high efficiency motor can save energy and money.

Obtaining optimal performance values in electrical machine design is a challenging task due to the intricate interdependencies among various factors. In Figure 2, multiple examples are illustrated to showcase the fluctuations in different performance values resulting from modifications in the design variables, coil turns, tooth tip height, and length of the motor. In order to emphasize the patterns in performance values related to variations in design variables, the numerical values on the y-axis have been excluded in Figure 2.

2.1.3 Electrical machine design

For different use cases we need to design personalized induction motors to meet the specific requirements and needs. Each application may have unique operating conditions (e.g. speed, torque, power, environment), and different design features may be required to optimize performance and efficiency. By designing induction motors specifically for a particular application, we can ensure that the motor will operate efficiently and reliably, resulting in reduced energy consumption, lower maintenance costs, and increased productivity. Additionally, personalized induction motors can also be designed to meet specific safety standards, provide better performance, and improve overall system integration.

Induction motors are extensively used in various industrial and commercial applications, and a well-designed motor is efficient, durable, and meets the necessary demands of its designated purpose, whereas a poorly designed one is energy-intensive

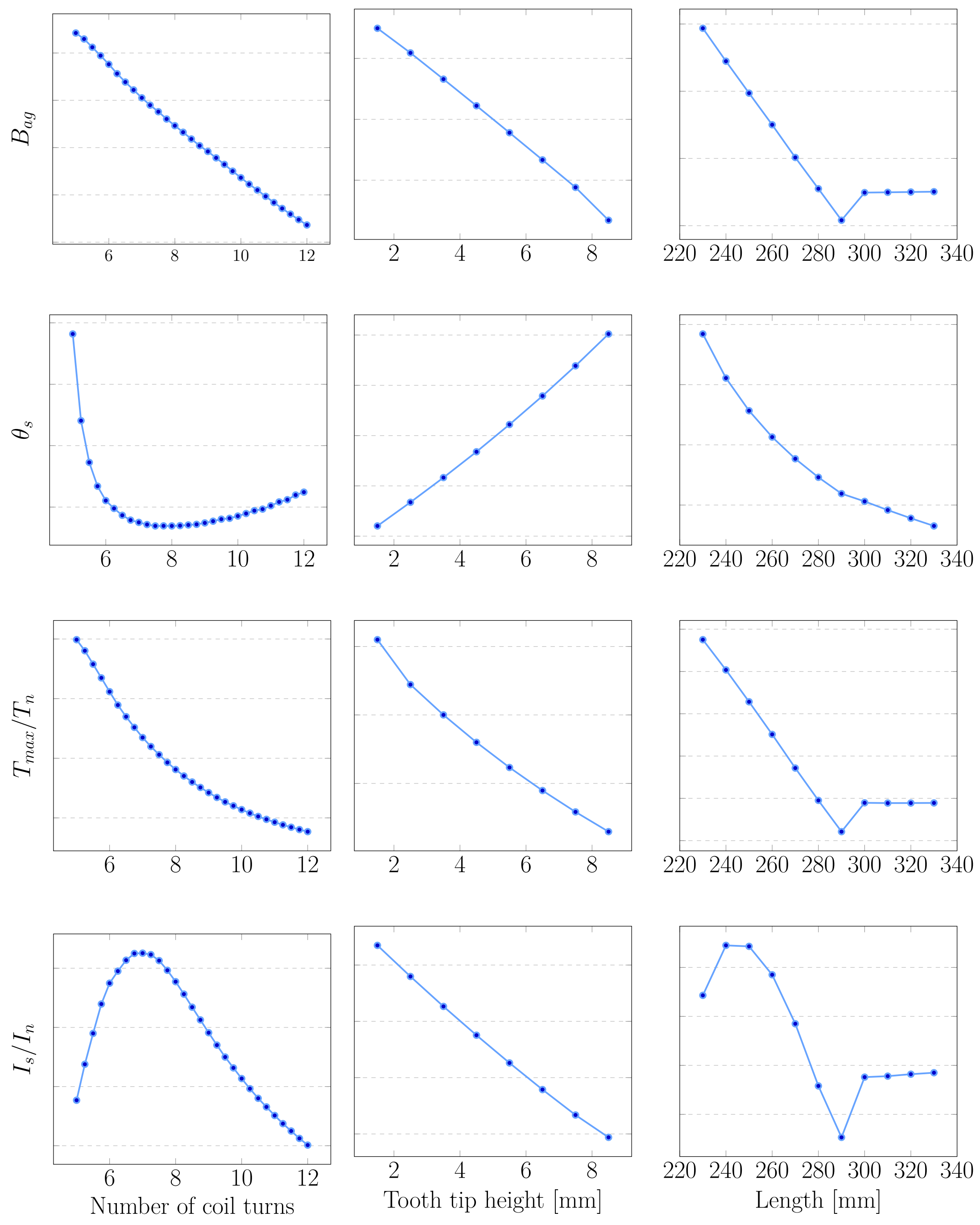


Figure 2: Variations in performance values due to changes in coil turns, tooth tip height and length design variables, where number of coil turns is actually the effective number of turns (number of turns/parallel path), I_s/I_n is the relative starting current, T_{max}/T_n is the relative breakdown torque, θ_s is stator winding temperature rise, and B_{ag} is the airgap flux density.

and prone to failure [12, 11]. Designing an induction motor involves critical factors like core construction, winding insulation, and optimizing geometries like rotor and stator windings. Moreover, parameters such as voltage, frequency, size, weight, torque, speed, efficiency, reliability, cost, and environmental conditions must be considered to achieve an effective design. This chapter aims to emphasize the significance of the design of induction motors through an introduction that will briefly discuss essential design considerations and trade-offs, and set the stage for a deeper understanding of the design process.

The design of the electrical motor is considered as a parameter optimization problem, wherein the goal is to find optimal values for the motor's parameters.

$$\begin{aligned} \text{Minimize } & F(x) \\ \text{Subject to } & G(x) = 0 \\ & \bar{G}(x) < 0 \end{aligned} \tag{1}$$

The objective function $F(x)$ is defined with respect to the design variables represented by x. Additionally, the equality and inequality constraints are denoted by $G(x)$ and $\bar{G}(x)$, respectively. In this study, the design parameters pertain to the length, coil turns, and rotor tooth tip height of the electrical machine. The primary aim of the objective function is to minimize the mass of the machine, while ensuring its feasibility through the satisfaction of equality and inequality constraints. The design process of the electrical motor is influenced by user-specific requirements, operational requirements, and manufacturing constraints, which are formulated as equality and inequality constraints within this study. Hence, to attain the design of a specific machine with predefined power and voltage values, it is necessary to satisfy the design feasibility while minimizing the objective function.

To evaluate the feasibility of the motor design, certain performance criteria have been established. This study considers five specific performance values, which include rotor tooth tip height, airgap flux density, starting current, starting torque, and stator temperature rise. These performance values serve as indicators or measures of the motor's performance and operational characteristics. Each value represents a specific aspect of the motor's behavior and functionality. By considering these specific performance values, engineers can thoroughly evaluate the feasibility and effectiveness of the motor design. The study aims to optimize the motor's parameters and meet the desired performance values while ensuring the motor's reliability and efficiency.

Designing an electrical machine requires thoughtful planning and engineering to produce a device that fits its intended purpose, while also achieving desired performance and reliability criteria. Basically, it is a trade-off between efficiency, cost, power density, and reliability, and requires a thorough understanding of the operating conditions, power supply, as well as application requirements [13]. The trade-offs in the design of an induction motor can be:

- Efficiency vs. cost: Increasing the efficiency of an induction motor often involves adding additional materials, which increases the cost of the motor.

- Torque vs. speed: An induction motor can be designed to produce high torque or high speed, but not both simultaneously. A trade-off must be made between the two depending on the requirements of the specific application.
- Size vs. power output: Increasing the power output of an induction motor often requires a larger motor, which may not be practical in some applications due to space constraints.
- Voltage vs. current: An induction motor can be designed for high voltage and low current, or low voltage and high current, but not both simultaneously. A trade-off must be made between the two depending on the requirements of the specific application and the electrical supply available.
- Efficiency vs. power factor: Improving the efficiency of an induction motor often involves reducing the power factor, which can result in higher current draw.
- Starting current vs. torque: An induction motor can be designed for high starting current and high starting torque, or low starting current and low starting torque, but not both simultaneously. A trade-off must be made between the two depending on the requirements of the specific application.

The process of designing an electrical machine involves repeating certain steps until the desired outcome is reached. Initially, an engineer must gather all the fundamental parameters such as voltage, frequency, shaft power, and rotational speed. In addition, they must consider other constraints such as maximum allowable starting current, under-voltage during startup, and thermal limitations. After gathering all the necessary information, the engineer selects a starting point for the motor design that includes a rotor of appropriate size to generate the required torque. They then set the stator winding turns to achieve the desired air gap flux density. Once the design is complete, the engineer evaluates the performance of the motor. If the motor meets all the constraints, the design is accepted. However, if the design does not meet all the requirements, the engineer adjusts the design by changing core length, number of winding turns, or other relevant parameters. The engineer then repeats the previous step until an acceptable design is obtained [14].

In the context of large corporations, particularly within the expansive industrial sector where numerous motors are manufactured daily to cater to diverse customer needs, the electric machine design process and dimension estimation necessitate rapid execution. In such cases, the prompt availability of machine dimensions based on application, efficiency, load curve, and operational conditions becomes significantly crucial. To address this demand, electric motor designers typically selects a base or seed machine, which is a machine that closely aligns with the customer's requirements. Subsequently, certain machine parameters are adjusted to achieve the desired customer specifications. Traditionally, this is accomplished through a deterministic approach, either manually or with the aid of an algorithm, where the machine designer iteratively modifies specific design parameters to assess compliance with the customer's requirements. However, this process remains time-consuming, impeding the ability to provide customers with immediate feedback on machine design and pricing during sales negotiations.

2.2 Reinforcement learning (RL)

To address the given challenge in designing electrical machines, a RL approach is used. The integration of RL techniques into the design of induction machines represents a promising avenue for advancing the field of machine design. RL is an area in machine learning that looks at the behaviour of an intelligent agent interacting with a certain environment in order to maximize its cumulative reward [15]. RL along with supervised learning, and unsupervised learning are three main paradigms of machine learning. In supervised and unsupervised learning the goal is finding patterns in the training data and learning about them. In contrast, RL focuses on creating a policy that instructs an agent which course of action to take at each stage, and therefore it is more beneficial in dynamic problems. Another difference between RL and supervised learning is that RL doesn't call for display of labelled input/output pairs, or an explicit corrections of undesired behaviours. In fact, the agent determines how to carry out the job at hand by learning from its experience after completing the task for a specific number of times.

The main elements of RL are an agent and an environment. According to [1], the agent interacts with the environment (as shown in Figure 3) that can be either fully observed or partially observed. When the agent is able to observe the complete state of the environment, it is referred as fully observed. Chess is a good example of this because both players of the chess have access to all the information; whereas, when the agent can only see a partial observation of the state, we say that the environment is partially observed [16]. Games like poker are examples of partially observed games because each player can see only his own hands, but not those of the opponents.

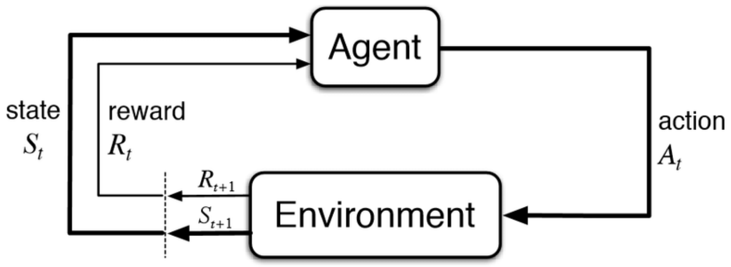


Figure 3: The reinforcement learning framework [2].

In RL, the agent (policy network) starts from an initial state S_0, then at each time step t, the agent begins in state S_t, takes an action A_t, arrives at a new state (S_{t+1}), and receives a reward (R_{t+1}) until it arrives at the final state S_T. The aim of the agent is to maximize its cumulative reward during this procedure.

The emphasis of RL is to find a balance between the exploration and the exploitation [17]. Exploration is more of a long-term concept because it allows the agent to better understand every action that can be advantageous in the long run. While exploitation, on the other hand, effectively exploits the agent's current estimated value and chooses the greedy method to maximize its reward. So it may be possible that the agent won't receive the highest reward since it is being greedy with the estimated value rather than the actual value [18]. This exploration-exploitation trade-off is a result of an agent's partial awareness of the state, actions, and rewards.

Due to the generality of RL, it has several applications [19]. (As presented in Figure 4), from autonomous driving [20], medicine [21], and healthcare [22], to control theory [23], multi-agent systems [24], and statistics. RL can also be used to understand how equilibrium might develop under constrained rationality in economics [25] and game theory [26]. Other applications of RL are robotics [27], choosing where to put ads on a website [28], recommender systems [29] and so on. Although, RL has many applications, it is more difficult to apply RL to real-world business scenarios because it needs simulated data and an environment.

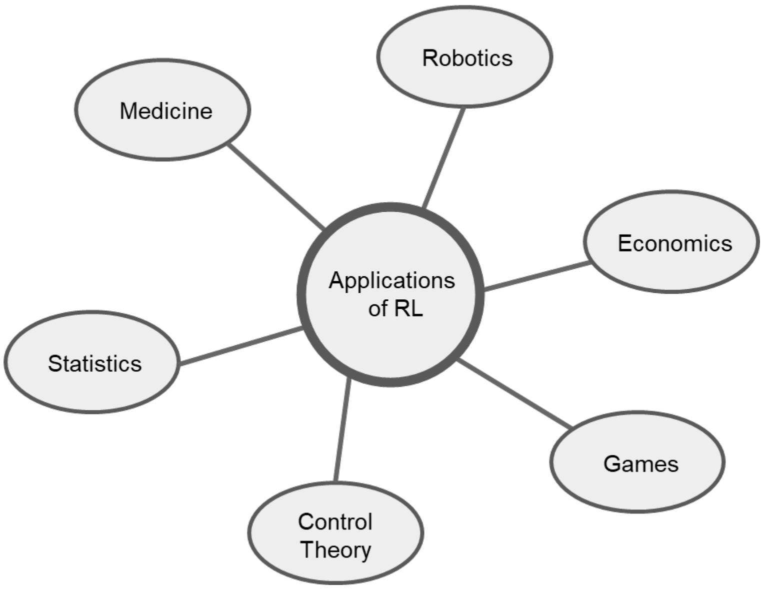


Figure 4: Some applications of RL.

2.2.1 Markov decision process

Markov decision processes (MDP) can be considered as the environment in which an agent operates and thus are used to model the basics of the RL. In mathematics, a discrete-time stochastic control process is known as a MDP [30]. It provides a mathematical formulation for modeling choices in circumstances where results are influenced by chance and a decision-maker. Essentially, the phrase "Markov Property" applies to a stochastic process that has memoryless characteristic, which means that the probability of the next event only depends on the state obtained in the current step, and not any other previous steps [31]

$$P(X_{n+1} = x_{n+1}|X_n = x_n, X_{n-1} = x_{n-1}, \cdots, X_0 = x_0) = P(X_{n+1} = x_{n+1}|X_n = x_n) \tag{2}$$

Main elements of MDPs are:

1. States (S): The set of possible states in the environment.

   A state, denoted by s, is a comprehensive description of the current situation in the world that the agent has access to. On the other hand, an observation, denoted by o, is a limited description of a state, which may not include all information about the current situation. While observations and states may be the same in some cases, this is not always the case.

   In RL, we almost always represent states and observations by a real-valued vector, matrix, or higher-order tensor.

2. Actions (A): The set of all possible actions that may be taken by the agent in the given environment.

   Different environments allow different kinds of actions. The action spaces in some environments, such as Atari and Go, are discrete, meaning that the agent can only make a limited number of moves. On the other hand, other environments, such as controlling a robot in the physical world, have continuous action spaces. This difference in action spaces has significant implications for deep RL methods, as some algorithms can only be used in one type of action space and would require significant modification to work in the other.

   In the context of electrical machine design, the action space can include altering the motor's length, adjusting the number of coil turns in the winding, and modifying the tooth tip height of the rotor slot.

3. State transition ($P(s'|s, a)$): Given a state (s) and an action (a), this is the probability distribution over next states.

   In RL, the state transition function most generically defines the probability of an agent transitioning from one state to another. The state transition function can be either deterministic or non-deterministic.

4. Reward (R_t): The immediate reward (or expected immediate reward) for taking an action (a) in a state (s) at the time step t.

MDPs with finite state and action spaces with an explicitly provided transition probabilities and reward functions can be used to research optimization issues based on dynamic programming [32, 33]. They work in a variety of industries, including manufacturing, economics, automatic control, and robotics. In contrast, MDPs in which the probabilities or rewards are uncertain, are used in RL and the problem in such MDPs is to find a "policy" for the decision-maker. Therefore, the aim of RL for the agent would be to learn an optimal (nearly-optimal) policy that maximizes the cumulative reward for MDPs. Mostly, to compute the optimal policy for RL algorithms, two different functions indexed by state are needed: Value function $V_t(s) \in R$ that is the expected cumulative rewards starting from state s, and the policy $\pi(s) \in A$ that models the agent's behaviour.

For a RL agent, in order to maximizes its cumulative rewards (not only the immediate reward from one decision), the agent must concurrently gather present rewards and also consider the future ones. Since the future rewards do not worth the same as the current rewards, we need to use a mechanism to discount the importance of future rewards. This discounted Reward can be expressed by the term return G_t

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

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate. if γ is smaller than 1, therefore future occurrences are given a lower weight than present rewards [34].

2.2.2 Elements of RL

RL is a strategy where an agent acts and observes the rewards to explain how to operate in a certain setting. In RL, learning happens as a result of an agent's interactions with the outside world. The algorithm presents a state where a user can commend or criticize the algorithm for taking a certain action based on the input data. The reward/punishment cycle is repeated as the RL algorithms learn from the experience and make adjustments (as presented in Figure 3).

A RL model consists of four fundamental parts in addition to the agent and the environment: a policy, a reward, a value function, and the model of the environment.

Policy

An agent's behavior at a particular time is controlled by a set of rules that is called policy [35]. Policy connects states of the environment to the actions made in response to the perceived states. In a more formal sense, it's a mapping between environmental conditions and the activities that should be conducted in those conditions. The policy may be a lookup table, a straightforward function, or it may need complex calculation like a search procedure. Additionally, the behavior of the agent might be determined only by the policy. Since only a policy can specify how an agent will behave, therefore, it is the fundamental component of RL.

Policies are used to determine the behaviour of the agent. However, the fact that policies only take into account the current state, rather than additional information like time or previous states, is another crucial aspect of policies. In a Markov decision process, the current state effectively defines all the data that is utilized to choose the current action. In deep RL, we work with parameterized policies, which are rules whose results are calculable functions that depend on a set of parameters that we can modify using an optimization technique to control the behavior. A policy could be either deterministic or stochastic:

- Deterministic policy ($a = \pi(s)$): A deterministic policy refers to a mapping from states to actions that always selects the same action for a given state. In other words, given a particular state, a deterministic policy will always choose a specific action without any randomness or probability involved. Deterministic optimal policies can be found using state-value or action-value functions.

As an example, deterministic policies are commonly employed in industrial automation settings, where consistent and predictable actions are required. Manufacturing processes, assembly lines, and robotic systems often rely on deterministic policies to ensure precise and reliable operations.

- Stochastic policy ($\pi(a|s) = P[A_t = a|S_t = s]$): A stochastic policy is a probabilistic mapping from states to actions. Unlike a deterministic policy that always selects the same action for a given state, a stochastic policy incorporates randomness or probability distributions in decision-making.

  Stochastic policy has the following properties:

  - The sum of total probabilities for each state must be 1: $\sum_a \pi(a|s) = 1$.
  - The probability of each action should be non-negative: $\pi(a|s) \geq 0$.

  Stochastic policies can be useful in situations where there is uncertainty or noise in the environment, or where it is beneficial to explore different actions in order to gather more information. Stochastic policies can also be more flexible than deterministic policies, as they allow the agent to adapt to changing circumstances. As an example, stochastic policies can be applied in RL algorithms for financial trading, where the market conditions are inherently uncertain. By incorporating randomness, the policy can adapt to changing market dynamics and explore different trading strategies to optimize returns while managing risk.

It's typical to say "the policy is seeking to maximize reward" instead of "the agent," as the policy is essentially the agent's brain.

Reward signal

The purpose of RL is defined by the reward signal. The environment sends a signal known as a reward signal to the learning agent at each state, immediately. Rewards are considered as short-term signals that are given as feedback after the agent takes an action and transits to a new state. Summing all future rewards and discounting them would lead to the cumulative discounted reward G_t (3) that counts as a long-term signal.

The reward signal is an important aspect that can change the policy. For instance, the policy might be changed to choose other actions in the upcoming episodes if the agent's behavior results in a small reward. These rewards are given based on the agent's successful and unsuccessful actions. The main objective of the agent is to increase the cumulative reward by acting appropriately. In RL, when a multi-object constraint problem is solved, it is important to consider the objects and constraints of the problem in the reward function. In this study, the reward setting is examined to ensure that a reliable regulation is produced for the chosen RL method.

Value function

The value function informs an agent of the state's and action's qualities as well as the potential reward. A value function defines the future reward that an agent can expect to accumulate by taking an action in a particular state [36]. It is important to keep in mind that a policy π in RL is a function that maps a state s and an action a to the probability or likelihood of selecting that action given the state. In simpler terms, the expected return starting in a state s and then following a policy π that is indicated by $v_\pi(s)$, is the value of the state s under the policy π. For MDPs, we can define $v_\pi(s)$ formally as:

$$v_\pi(s) = \mathbf{E}_\pi[G_t | S_t = s] \tag{4}$$

where where $\mathbf{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π, and the return value G_t (3) is the cumulative discounted reward at timestep t. Note that the value of the terminal state, if any, is always zero.

We call the function $v_\pi(.)$ the state-value function for policy π. The value function indicates the good states and the actions for the future. Whereas, a reward indicates the immediate signal for each good and bad action. The reward is a necessary component of the value function because value cannot exist without it, and the main goal for estimating values is to eventually obtain better rewards. As a result, you can consider rewards to be the main objects and values to be the secondary objects (they are just used as predictions of future rewards). However, values are what we consider while making and assessing decisions. Therefore, instead of focusing on acts that will provide the biggest rewards, we aim to achieve states of highest value since these actions will ultimately yield the highest cumulative rewards.

Following the value function, even though an agent may constantly receive a low immediate reward from a state, if that state has a high value, it indicates that additional states with higher rewards will frequently follow it (therefore, it would be better for the agent to choose that state). The environment immediately provides rewards, but values need to be assessed and re-estimated based on the observations and rewards an agent gathers over the course of its training phase. This is so that the value of a state can change according to what the agent understands about the potential outcomes from that state in the future. Additionally, the agent will find more choices as it continues to explore.

Model of the environment

The model, which imitates the behavior of the environment, is the final component in RL. Model is the agent's representation of the environment, which includes the transition function and the reward function [36]. A model, for instance, can forecast the subsequent state and reward if a state and an action are provided. Basically, the model is used for planning. Planning entails developing a "plan" that you will utilize to accomplish a "goal." In RL, planning typically refers to the use of an environment model to identify a strategy that will, ideally, assist the agent in behaving optimally. Which means that the model offers a mechanism to choose a course of action by taking into account all potential outcomes before those outcomes actually occur.

The environment model is used to interact with an agent for training a policy, as demonstrated in Figure 5. Depending on the RL application, the environment may be a black box model (such as neural networks), a true system, a mathematical model, etc. In other words, the environment is formulated as a partially observable Markov decision process (POMDP) [1] and serves as a model of the problem under study. The agent can also interact with the actual environment when researching study targets like robots or video games and since everything is taking place entirely on a computer, the wining or losing of the agent is generally neither dangerous nor expensive. However, some real world environments are more likely to lack any prior knowledge of environment dynamics and model-free RL methods come handy in such cases.

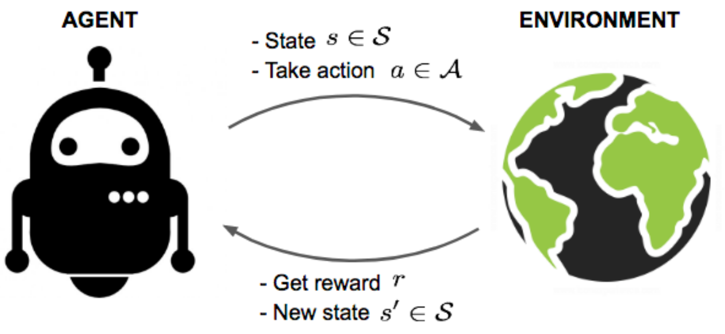

AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 5: A simple reinforcement learning illustration.

RL algorithms can be broadly categorized as model-free and model-based. The term "model-based approach" refers to methods for tackling RL problems with the use of models. In contrast, a model-free strategy is one that doesn't employ a model. Model-free algorithms do not create an explicit model of the MDP or, to be more precise, the environment. They are more similar to trial-and-error algorithms, which test hypotheses about the environment using actions and immediately determine the best course of action. The model-free algorithms can be value-based or policy-based. In the value based algorithms such as Proximal Policy Optimization (PPO) [3], the optimal policy follows directly from precisely calculating the value function of each state.

In RL, the agent chooses an action from a set of actions, moving from one state to another, and is rewarded for its decisions. This process proceeds in a sequential manner, which results in the creation of a trajectory that depicts the chain or sequence of states, actions, and rewards. Trajectory can also be thought of as the complete history of the agent's actions and the resulting rewards.

$$\tau = (S_0, A_0, R_1, S_1, A_1, R_2, \cdots) \tag{5}$$

where $S_t \in S$ is the state at time step t, $A_t \in A$ is the action taken by the agent at time step t, and R_{t+1} is the reward received by the agent after taking action A_t in state S_t.

The trajectory can be used to learn from past experiences and to make better decisions in the future. In a Markov decision process (MDP), the trajectory follows the Markov property, which means that the future state depends only on the current state and the action taken, and not on the history of previous states and actions.

$$P(S_{t+1}|S_t, , A_t, S_{t-1}, A_{t-1}, \cdots , S_0, A_0) = P(S_{t+1}|S_t, , A_t) \tag{6}$$

Moreover, the Bellman equation is a tool that helps the agent interact with its environment and keep track of the trajectories and the rewards it receives. It does this by creating a connection between the value of a state $v_\pi(s)$ or state-action pair $q_\pi(s, a)$, the value of the next possible states $v_\pi(s')$ or state-action pairs $q_\pi(s', a')$, and the immediate reward received, R. Once the equation is solved, the resulting values can be used to guide the agent towards making better decisions and getting higher rewards. The Bellman equation plays a crucial role in numerous RL algorithms, making it a fundamental element of the field.

2.2.3 The Bellman equation

In RL, the goal of the agent is to get the maximum total reward. To do this, the agent needs to know the value of each state. This allows the agent to compare the value of the current state with the value of future states without having to wait for all the rewards to come in. The Bellman equation (7) formalizes this connection between the value of a state and its potential future states.

$$\begin{aligned} v_\pi(s) &= \mathbf{E}_\pi[G_t|S_t = s] \\ &= \mathbf{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} P(s'|s, a) \left[R_{t+1} + \gamma \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_{t+1} = s'] \right] \\ &= \sum_a \pi(a|s) \sum_{s'} P(s'|s, a) \left[R_{t+1} + \gamma v_\pi(s') \right] \end{aligned} \tag{7}$$

where $a \in A$, $s' \in S$ and $R_{t+1} \in R$. The Bellman Equation states that the long-term reward for a given action is equal to the sum of the rewards from the current action R_{t+1}, and the anticipated rewards from the subsequent acts, $\gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}$. Essentially, we can state that the immediate reward R_{t+1}, and the discounted future values, $\gamma v_\pi(s')$, are the two components that the Bellman equation divides the value function into.

In RL usually when the agent is in state s, it could choose from a variety of action options. Each of these could result in the environment responding with one of several subsequent states (s'), as well as a reward (R_{t+1}). The Bellman equation (7) takes an average of every possibility, weighting each one according to its likelihood of happening. According to Bellman equation, the start state's value must be equal to the (discounted) value of the anticipated next state plus any rewards that are anticipated along the way. Note that this state value function has a unique answer to the Bellman equation.

However, the agent does not know what to do in each state from the state value function alone. Moreover, Bellman offers another formula to calculate the value of state-action pairs called the state value function $q_\pi(s,a)$ that defines the value of taking an action a in the state s under a policy π:

$$q_\pi(s,a) = \mathbf{E}_\pi[G_t|S_t = s, A_t = a] = \mathbf{E}_\pi[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{8}$$

As a result of the state value function, the agent can make a decision on which action to take easily. It will choose the action with the highest state value function, as this is the action that is anticipated to produce the greatest total reward over time.

In this context, the agent's experience can be used to estimate the values of $v_\pi(s)$ and $q_\pi(s,a)$. For instance, if an agent follows the policy and keeps track of the average of actual returns, the average will eventually converge to the value of the state, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. The averages will similarly converge to the action values, $q_\pi(s,a)$, if separate averages are recorded for each action conducted in a state s. Because these estimating techniques entail averaging over numerous random samples of actual returns, we refer to them as Monte Carlo methods (section 2.2.6).

Of course, keeping individual averages for each state may not be feasible if there are a lot of states. As opposed to this, the agent would need to keep $v_\pi(.)$ and $q_\pi(.,.)$ as parameterized functions and change the parameters to be a better fit to the observed returns. Although, the expected return much depends on the characteristics of the parameterized function approximator, this can also result in reliable estimations.

We can also define the optimal value function that is the maximum value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s) \tag{9}$$

for all $s \in S$. Similarly, the optimal policy that shares the same optimal value function can be defined as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{10}$$

for all $s \in S$ and $a \in A(s)$. For each state–action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write $q_*(s,a)$ in terms of $v_*(s)$ as follows:

$$q_*(s,a) = \mathbf{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{11}$$

On the other hand, we know that the value of a state under the optimal policy must be equal to its best action that is called Bellman optimality equation for $v_*(s)$:

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_*(s,a) \\ &= \max_{a \in A(s)} \mathbf{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_{a \in A(s)} \sum_{s'} P(s'|s,a) \left[R_{t+1} + \gamma v_*(s')\right] \end{aligned} \tag{12}$$

And thus the Bellman optimality equation for $q_*(s,a)$ is (see [1]):

$$\begin{aligned} q_*(s,a) &= \mathbf{E}[R_{t+1} + \gamma \max_{a' \in A(s)} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s'} P(s'|s,a) \left[R_{t+1} + \gamma \max_{a' \in A(s)} q_*(S_{t+1}, a')\right] \end{aligned} \tag{13}$$

In some cases in RL, it is not necessary to describe how good an action is in an absolute sense, but rather how much better it is compared to other actions on average. This is known as the relative advantage of an action.

Advantage Function

The concept of the advantage function can be utilized to measure the difference between the value of a particular action and the average value of all feasible actions in a given state. This difference is evaluated with respect to the expected cumulative reward, and can be expressed using the advantage function defined by equation (14). For instance, policy gradient methods rely heavily on the advantage function.

$$\mathcal{A}_\pi(s,a) = Q_\pi(s,a) - V_\pi(S_t) \tag{14}$$

The advantage function $\mathcal{A}_\pi(s,a)$ for a policy π describes how much better it is to take a specific action a in state s, compared to randomly selecting an action according to $\pi(.|s)$, assuming that the policy π is followed afterwards.

The Bellman optimality equations, which can be used to find the optimal policy (13) and value function (12), can be solved iteratively if the reward functions and transition probabilities are known. This process is called dynamic programming. The Bellman equations provide the theoretical foundation for model-based RL algorithms by allowing us to compute the optimal policy using a model of the environment. Basically, in RL there are two main types of algorithms: model-based 2.2.4 and model-free methods 2.2.4. Model-based algorithms either assume that the transition probabilities are known or estimate them online. In contrast, model-free methods do not assume any prior knowledge of these probabilities and instead estimate the policy and value function through repeated trajectories or state-action sequences.

2.2.4 Model-free vs model-based algorithms

RL algorithms can be either model-based or model-free in terms of interacting with the environment. This refers specifically to whether the agent employs predictions of the environment's response while learning or behaving. Model-free RL directly learns a value function or a policy by interacting with the environment, whereas, model-based RL builds a model of the world through interactions with it [37].

Both model-based and model-free methods can be useful tools in RL, depending on the specific problem being addressed. It is generally acknowledged that model-based RL has the potential to be much more sample efficient than model-free RL. However, model-free RL is widely applicable to various tasks since it doesn't need a model of the environment. It's preferred when environment dynamics are unknown, as it's flexible and adaptable to unpredictable changes. It handles large state spaces and allows faster decision-making without extensive planning.

Model-Based

Model-based RL is a method that enables an agent to learn by predicting the outcomes of its actions without directly interacting with the environment. This involves learning a model of the environment and using it to make decisions, combining the model and global solution in a single algorithm [38]. This approach offers numerous benefits, including improved sample efficiency, which has led to its popularity in robotic applications. Model-based RL can be implemented using two methods: constructing an environment model from the scratch and utilizing it to train the policy or value function, or conducting experiments on the system to develop an environment model that expedites the learning process. Effective strategies for model-based learning aim to enhance the accuracy of the learned policy by acquiring a model that facilitates accelerated learning.

Model-Free

Model-free RL methods do not require a model of the environment and instead rely on trial-and-error learning from experience to update their policy or value function. They adjust their behavior only in response to the results of their decisions and can learn from any RL task, making them more widely applicable to real-world problems. They are preferred over model-based when environment dynamics are not completely known, as they are more flexible, adaptive, and better suited for non-stationary environments. Model-free RL methods are the primary focus of attention in this work.

In their pioneering work on RL [1], authors Barto and Sutton used a rat in a maze to explain model-free RL by memorizing action values for state-action pairs the rat encounters during learning. The rat learns from experience which actions lead to better outcomes and assigns higher values to those actions. With this information, the rat can navigate the maze by choosing the action with the highest value for the current state, without considering future states or the underlying dynamics of the maze. This approach is model-free since it does not require a model of the environment and learns directly from experience.

Model-free algorithms most frequently utilized include Monte Carlo (section 2.2.6), temporal difference (section 2.2.6), and policy search techniques [39]. The most of the model-free approaches either explicitly search in the space of the policy parameters to discover an optimal policy (section 2.2.7), or try to learn a value function and infer an optimal policy from it (section 2.2.7). There are several popular model-free RL algorithms such as Q-learning [40], SARSA [1], and actor-critic methods like PPO [3]. These algorithms vary in the way they estimate the value function or policy, and in their suitability for different types of problems.

2.2.5 Off-policy RL vs on-policy RL

Model-free strategies can be categorized as either on-policy or off-policy. In RL, off-policy and on-policy methods refer to the way in which an agent learns and improves its policy. In off-policy RL, two policies are used - one for exploration (the exploratory policy) and one for optimization (the target policy). During training, the exploratory policy is used to collect data, and the target policy is updated based on this data to improve performance. The exploratory policy is not intended to be the final policy and is only used for data collection. In contrast, on-policy RL only uses one policy, which is the same policy used to select actions during training and to update itself based on the received rewards. In other words, the policy being improved is the one currently in use, and there is no separate exploratory policy.

Both on-policy and off-policy methods can be useful tools in RL, depending on the specific problem being addressed. On-policy methods are generally more sample efficient and do not require the storage of the experience, but may be sensitive to the learning rate and can struggle to learn when the current policy is poor. Off-policy methods can learn from experience collected by other policies, but require the storage of experience and may be less sample efficient.

Off-policy

Off-Policy learning algorithms evaluate and improve a policy that is distinct from the policy that is used to choose the course of action. This means that the agent can learn from the actions of a policy that is different from the one it is currently following. In other words, in Off-Policy methods the behavior policy is the policy used to generate the data (or experience) that the agent learns from, while the target policy is the policy that the agent is trying to learn and is used for function estimate and improvement [Target Policy!= Behavior Policy].

This process works because the target policy that obtains a "balanced" understanding of the environment, is able to draw lessons from potential errors made by behavioral policy, and continues to monitor successful acts while looking for ways to improve them. An important consideration in Off-policy learning is the difference between the distributions being sampled (behavior policy) and the ones being estimated (target policy). This mismatch can be fixed by a method known as importance sampling [41]. Off-policy methods are advantageous in the following aspects:

1. An agent can be employed for ongoing exploration while learning the best policy as it learns alternative policies (continuous exploration).

2. Agent can pick up knowledge from the example (Demonstration based Learning).

3. An agent can also learn from observation of other agents that accelerates convergence and allowing for quick learning (Parallel Learning).

However, they require the storage of experience from previous interactions with the environment, which can be costly in terms of memory and computational resources.

On-policy

On-policy learning algorithms are those that analyze and improve the policy that is being used to make decisions. This means that the algorithm focuses on improving the current action selection policy used by the agent, rather than a different policy [Target Policy == Behavior Policy].

On-policy approaches resolve the exploration vs. exploitation trade-off by introducing randomness in the form of a soft policy. This means that non-greedy actions are chosen with some probability. These methods are mostly known as ϵ-*greedy* policies because during the training they choose random actions with a ϵ probability and the optimal actions with a $1 - \epsilon$ probability. Since, choosing at random from the action space has an ϵ probability, choosing any specific non-optimal action has an $\frac{\epsilon}{|A(s)|}$ probability. However, as there is also a $1 - \epsilon$ probability of directly selecting the optimal action and a probability of $\frac{\epsilon}{|A(s)|}$ of selecting it by sampling from the action space, the chances of selecting the optimal action are always slightly higher.

On-policy methods can also be more sample efficient, since they do not need to collect experience from multiple behavior policies. However, they can be more sensitive to the choice of learning rate and may require careful tuning to achieve good performance. They can also struggle to learn when the current policy is poor, since the agent will continue to follow the current policy even if it is not achieving good results.

2.2.6 Monte-Carlo policy evaluation vs temporal difference

Both Monte Carlo policy evaluation (MC) and temporal difference (TD) learning are model-free methods used to evaluate and improve the performance of the policy, which specifies the decision-making procedure for selecting an action in a given situation. However, they differ in their methodologies and the type of problems they are best suited for.

Typically, MC is a favorable option for problems characterized by a limited number of states and a stable environment, such as episodic tasks with long episodes where the value function can only be evaluated at the end of each episode. On the other hand, TD methods are better suited for problems that involve a considerable number of states or a dynamic environment, as well as continuing tasks with short episodes where the value function can be estimated at every step. Both approaches can be employed to improve the policy performance, but the decision between MC and TD ultimately relies on the specific problem at hand and available resources.

Monte-Carlo policy evaluation (MC)

We've learned that a Markov Decision Process (MDP) (section 2.2.1), which makes choices using the four values, state (s), probability function ($P(s'|s,a)$), action (a), reward (R), can be used to solve a RL problems. Knowing all four elements makes it simple to determine the best course of action to take in order to maximize the reward. In the actual world, we hardly ever have access to all of this information at once. For instance, it is challenging to determine the state transition probability ($P(s'|s,a)$), and without it, we are unable to compute $v_\pi(s)$ and $q_\pi(s,a)$ values using the Bellman Equations (12), (13).

Compared to dynamic programming, MC methods do not require a detailed understanding of the environment or an environment model. MC methods are model-free off-policy approaches that use experience in the form of sampled states, actions, and rewards from actual or virtual interactions with the environment to achieve optimal behavior without prior knowledge of its dynamics. When the state transition probability or environment model is unknown, setting up trials for the agent to complete can help discover them. Each attempt, called an episode, involves gathering samples, getting rewards, and evaluating the value function. Contrary to what happens in the Bellman optimality equation (12), values for each state are only modified based on the final reward G_t (3) and it learns from the whole episodes. Therefore, MCs are referred as episodic MDP.

In each episode, the agent follows a given policy and interacts with the environment, collecting a sequence of state-action pairs and the resulting rewards. At the end of each episode, the return G_t (3) is computed and associated with each state visited during that episode. After running multiple episodes, the MC approach averages the returns obtained from each episode to estimate the state-value function, $V(s)$, for each state. The idea is that by averaging the returns over many episodes, any bias or variance in the estimates will eventually cancel out, resulting in a more accurate estimate of the true value function, $v(s)$.

As more episodes are generated, the MC approach iteratively gets closer to the actual value function. This is because the sample-based nature of the approach that allows it to adapt to a wide variety of environments without requiring any knowledge of the underlying dynamics. However, the MC approach can be computationally expensive and may require a large number of episodes to converge to an accurate estimate of the true value function. Additionally, it can only be used for episodic tasks where episodes eventually terminate, making it unsuitable for continuous control problems.

In mathematical terms, MC methods involve the averaging of sampled returns to address problems in RL. The MC method modifies the state value formula as shown below:

$$V(S_t) = V(S_t) - \alpha(G_t - V(S_t)) \tag{15}$$

in which $V(S_t)$ is the estimated value function for state S_t in time step t, and α is a parameter like learning rate which can affect the convergence.

Temporal difference (TD)

The MC RL algorithm can overcome the challenge of estimating the value function when the model is unknown. However, the value function may only be altered at the end of the episode, which is a drawback. In other words, the MDP learning task structure is not fully utilized by the MC approach. Fortunately, the more effective temporal difference (TD) approach enters the picture. The TD approach merely needs to wait until the subsequent time step, and thus are considered as on-policy methods. The benefit of TD approaches is that they can be used for online learning and continuous RL tasks. The TD technique constructs a TD target at time $t+1$ by adding the observed reward R_{t+1} and updating $V(s)$ with the TD error:

$$V(S_t) = V(S_t) - \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{16}$$

in which $R_{t+1} + \gamma V(S_{t+1})$ is the TD target.

TD estimates the value by applying the Bellman optimality Equation and updating the estimate using a target value, whereas MC (15) uses the accurate return G_t at a given time to update the value function. Unlike TD, MC is not biased as it employs real samples, but due to its high variance, it requires more samples than TD to achieve the same level of learning. While MC may be slower than TD, it is conceptually simple, reliable, and easy to implement. As a result, TD and MC approaches have different strengths and weaknesses, and the choice between them depends on the specific learning task at hand.

2.2.7 Different RL algorithms

Besides MC and TD learning techniques, various forms of RL algorithms have emerged in recent years. These include value-based, policy-based, and actor-critic methods. In Section 2.2.7, value-based methods are used to determine the optimal policy by estimating the ideal value function. Conversely, Section 2.2.7 focuses on policy-based methods which manipulate the policy directly to identify the optimal policy. Actor-critic methods (section 2.2.7) integrate both value-based and policy-based approaches to achieve optimal results.

Value-based algorithms

Value-based methods in RL can be considered as an expansion of the TD methods. The agent's goal is to learn a policy that maximizes the expected cumulative reward over time. Value-based RL algorithms use a value function to estimate the expected cumulative reward an agent can expect to receive by following a particular policy π. Mathematically, the value function for a policy π is defined as the expected cumulative discounted reward when starting in a state s and following policy π thereafter. Let $V_\pi(s)$ be the value function for policy π at state s. Then we have:

$$V_\pi(s) = \mathbf{E}_\pi[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s] \tag{17}$$

where $\mathbf{E}_\pi[.]$ denotes the expected value under policy π and S_0, S_1, S_2, $\cdots$ are the states visited by the agent. The agent can then use this value function to select actions that maximize the expected cumulative reward.

A tabular format is the simplest approach to save the values of the value function for various stages. However, it becomes impossible to keep all the values in a tabular format if the problem's state space is very big, thus the amount of memory needed to hold all the data becomes huge. Given the objective of this thesis, which aims to propose a comprehensive method applicable to various types of electrical machines, the design space being considered is quite extensive. Furthermore, conducting a complete scan of a table to search for a specific condition can be computationally expensive, which makes a tabular format impractical when dealing with continuous state spaces. As a solution, function approximators are employed to store a value function and address this challenge.

Numerous function approximators, including neural networks, radial basis functions, and tile coding, have been employed in literature. Neural networks are a popular tool for this in modern times. The benefit of employing neural networks over tile coding or radial basis functions is that they may represent value functions in a more complicated manner with fewer parameters than other techniques [42].

In such value-based methods, we begin with the prediction problem of estimating the state-value function $V_\pi(s)$ from experience generated using policy π. We will write $\hat{V}_\pi(s,\theta) \approx V_\pi(s)$ for the approximated value of state s given weight vector θ [1]. The goal is to reduce the error between this approximated value function and the true value function on the observed cases. Gradient-descent algorithms do this by making a tiny adjustment to the parameter vector in the direction that would most effectively minimize the error on each example after it has been run:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \frac{1}{2}\alpha\nabla_\theta[V_\pi(s) - \hat{V}_\pi(s,\theta)]^2 \\ &= \theta_t + \alpha(V_\pi(s) - \hat{V}_\pi(s,\theta))\nabla_\theta\hat{V}_\pi(s,\theta) \end{aligned} \tag{18}$$

where α is the learning rate. Keep in mind that we are just looking for an approximation that balances the errors in different states and we are not expecting to find a value function that has zero error on all states.

Examples of a value-based RL algorithm are Q-learning [43] and SARSA (State-Action-Reward-State-Action) [1]. In Q-learning, the value function is known as the Q-function and is represented as a table or matrix that maps states to actions and estimates the expected return for each combination. The Q-function is updated using the Bellman equation, which takes into account the immediate reward received for taking an action and the estimated future rewards that the agent can expect to receive.

Policy-based algorithms

Policy-based RL is a type of RL that involves learning a policy directly, instead of determining the value of each potential state and then determining the best policy. Policy, $\pi(a|s,\theta)$, is a function that maps states of the environment to actions that the agent should take. In this method, the agent seeks to implement a policy in a way that each action serves to maximize the reward in the future. As an example, the policy can be represented as a neural network, and the agent updates the policy by adjusting the weights of the network, θ, based on the observed rewards.

The policy function is typically represented as a probability distribution over the set of actions. This allows the agent to not only determine the best action for a given state but also to express a level of uncertainty or exploration in its decision-making process. For continuous action spaces, a common way to represent the policy is with a Gaussian distribution

$$\pi(a|s, \theta) = \mathcal{N}(a; \mu(s, \theta), \sigma(s, \theta)^2) \tag{19}$$

which is parameterized by a mean $\mu(s, \theta)$ and standard deviation $\sigma(s, \theta)$. The mean represents the center of the distribution, or the most likely action to be taken, while the standard deviation represents the spread or variability of the distribution, or the level of exploration the agent is willing to take. Whereas, for discrete action spaces, a common way to represent the policy is with a multinomial distribution

$$\pi(a|s, \theta) = p(a|s, \theta) \tag{20}$$

which is parameterized by a set of probabilities [44]. Each probability corresponds to a different action, and the sum of all probabilities must be equal to 1.

$$\sum_a p(a|s, \theta) = 1 \tag{21}$$

The agent selects an action by sampling from this distribution, with actions that have higher probabilities being more likely to be selected.

Policy gradient-based methods typically evaluate the policy by generating rollouts using the existing policy and computing the reward based on the resulting trajectories [42]. The policy's parameters are then adjusted using gradient descent in order to increase predicted return. Using the expected return, the policy's parameters can be updated according to the following rule

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha \nabla_\theta \ \mathbf{E}_\pi[G_t] \\ &= \theta_t + \alpha \nabla_\theta \ \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}] \end{aligned} \tag{22}$$

where α is the learning rate.

There are several benefits to using policy-based methods. One advantage is that policy-based methods can learn continuous actions directly, without the need for discretization. This can be useful in environments where the action space is continuous or high-dimensional. However, policy-based methods can be more difficult to optimize than value-based methods, and may require more data to learn a good policy. They can also suffer from high variance in the estimates of the policy gradient, which can make them less stable and harder to converge. One reason for the high variance in policy-based methods is that the rewards obtained in a single episode can be highly variable. Another reason is that they depend on the state distribution under the current policy, which can also be highly variable. Overall, policy-based RL methods can be a powerful approach for learning to interact with complex environments, and have been applied to a wide range of problems, including robotics, control, and game playing.

The policy-based approach can be categorized into two types of policies: deterministic policy and stochastic policy. A deterministic policy involves the agent following a fixed policy to select its actions based on the current state of the environment, which means that the agent will always take the same action for a given state. In contrast, a stochastic policy involves the agent selecting actions based on probabilities rather than deterministically. This means that for a given state, the agent will have a probability distribution over the set of possible actions it could take.

Policy-based algorithms are the methods that focus on maximizing parameterized policies in relation to expected return (long-term cumulative reward) using gradient-based or gradient-free optimisation [45]. Both gradient-free [46, 47] and gradient-based [48, 49] techniques have been used to successfully train neural networks that encode policies. Although gradient-free optimisation can successfully cover low-dimensional parameter spaces [47], gradient-based training is still the preferred technique for the majority of Deep RL algorithms because it is more sample-efficient when policies include a lot of parameters [44].

Policy Gradient Methods have a stable convergence property because they update the policy directly, leading to smooth improvement at each step. In contrast, value-based methods, which update the value function at each step, can result in big oscillations during training. Additionally, Policy Gradient Methods can handle infinite and continuous action spaces and can learn stochastic policies, making them suitable for uncertain environments. However, a potential drawback is that they may converge to a local maximum rather than the global optimum [1].

Numerous different policy search methodologies have been developed as a result of how the gradient is defined and assessed [45]. Some examples of policy-based RL algorithms are REINFORCE [50], a policy-based method that uses MC sampling to estimate the gradient of the expected return with respect to the policy parameters, and Deterministic Policy Gradient (DPG) [51], a policy-based method that uses the deterministic policy gradient theorem to learn the policy.

Actor-critic algorithms

Actor-critic approaches can be created by combining value functions with an explicit description of the policy, as shown in Figure 6. These techniques incorporate both policy-based and value-based approaches, where the "actor" (policy) learns from feedback provided by the "critic" (value function). Actor-critic methods compromise the variance reduction of policy gradients with the inclusion of bias from value function techniques [52]. Since actor-critic methods use both value-based and policy-based approaches, they can be considered a subset of policy gradient methods. In other words, actor-critic methods are a specific type of policy gradient method that incorporate value functions to improve performance. As a result, we will talk about actor-critic techniques as a subset of policy gradient methods in a this section.

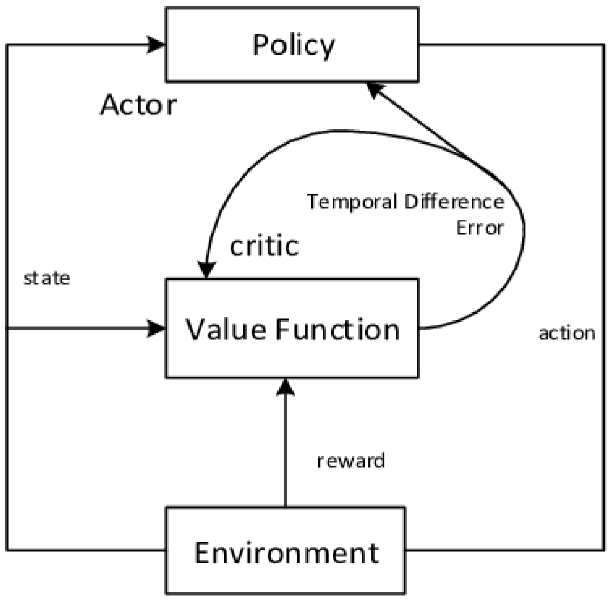


Figure 6: The actor–critic architecture. [1]. The output action is mapped from an input state by the Actor, which represents the policy and the critic is a symbol for the value function. Then the TD-error, which the critics output contributes to, can be updated for both networks. As a result, the Actor utilizes the Critic while learning.

In these methods, the actor is responsible for selecting actions based on the current policy and therefore, the actor is actually the policy function. Recall from the Bellman-equation (11) that:

$$Q_\pi(s,a) = \mathbf{E}[R_{t+1} + \gamma V(S_{t+1}) | S_t = s, A_t = a] \tag{23}$$

The optimal policy can be calculated by choosing the action with the highest expected cumulative reward:

$$\pi(s) = \max_a q_*(s,a) \tag{24}$$

where $q_*(s,a)$ is the optimal action-value function, and $\pi(s)$ is the optimal policy.

Here $V(.)$ is the state-value function that is called critic. It evaluates the quality of the chosen actions based on the expected reward. After taking an action, it looks at the new situation and decides whether things turned out better or worse than what was expected [1]. Actor-critic methods provide a TD error signal, the same as TD method (section 2.2.6), that refers to the difference between the expected reward and the actual reward received at each time step during training:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{25}$$

where R_{t+1} is the immediate reward received after taking the action at time t, γ is the discount factor, $V(S_t)$ denotes the value function that the critic used at time t, and $V(S_{t+1})$ is the estimated value of the state that follows the current state.

TD error is used in actor-critic methods to update both the actor and critic networks. The actor network uses TD error to update the policy parameters, while the critic network uses TD error to update the value function parameters. Specifically, the actor network is updated to increase the probability of selecting actions that result in higher TD error values, while the critic network is updated to minimize the TD error between the predicted and actual values of the state or action values. A positive TD error indicates that the action produced results better than expected, and therefore, similar actions should be encouraged in the future. Conversely, a negative TD error implies that the action resulted in outcomes worse than anticipated, and thus, comparable actions should be discouraged in the future.

We know that the advantage function is nothing more than the difference between the Q value for a state - action pair $Q(s, a)$, and the state's value function for a particular state $V(s)$. It is a measure of how good it is for an agent to take a certain action at a certain state. Therefore, the TD error in this case actually is the advantage function.

$$\begin{aligned}\mathcal{A}(S_t, A_t) &\overset{(14)}{=} Q(S_t, A_t) - V(S_t) \\ &\overset{(23)}{=} R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\end{aligned} \tag{26}$$

The critic component updates the value function $V(S_t)$ using the Bellman equation:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{27}$$

In which critic component uses the TD error to update the estimate of the value function. Specifically, the critic component updates the value function parameters in the direction of the TD error. This can be achieved using gradient descent, where the weights of the value function are updated in the direction of the negative gradient of the TD error with respect to the weights. The update rule for the value function parameters using gradient descent can be written as:

$$\Delta\theta = \alpha\nabla_\theta[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] = \alpha\nabla_\theta[\delta_t] \tag{28}$$

where $\Delta\theta$ is the change in the weights of the value function, θ are the parameters of the value function, α is the learning rate, and ∇_θ is the gradient of the TD error with respect to the weights of the value function By repeatedly updating the value function using the Bellman equation and TD error, the critic component learns to estimate the value function accurately, which is used to guide the actor component towards better policies.

Actor critic algorithms either use function approximators or tabular storage of the actor and critic. The policy is updated using function approximators in a manner similar to policy search methods, with the exception that the critic determines the gradient ascent's direction rather than the expected return. As a result, the variance of these kinds of algorithms is decreased.

Gradient-based actor-critic approaches have better convergence properties compared to critic-only methods, which only guarantee convergence under specific conditions. In contrast to actor-only approaches, they can converge more quickly by reducing variance. On the other hand, the case of lookup table representations of policies has served as the exclusive focus of theoretical understanding of actor-critic approaches [53].

Some examples of actor-critic RL methods include:

- Deep Deterministic Policy Gradients (DDPG) [54]: DDPG is an actor-critic method that uses deep neural networks to learn the policy and value function. It is well suited for continuous action spaces and has been used in a variety of applications, including robot control and playing video games.

- Trust Region Policy Optimization (TRPO) [55]: TRPO is an actor-critic method that uses a trust region optimization algorithm to improve the policy. It is designed to be sample efficient and stable, and has been used in a variety of applications, including robotics and natural language processing.

- Proximal Policy Optimization (PPO) [3]: PPO is an actor-critic method that uses a proximal optimization algorithm to improve the policy. It is designed to be easy to implement and has been used in a variety of applications, including playing video games and optimizing supply chain management.

Actor-critic approaches are a powerful tool for learning in continuous action spaces

3 Problem formulation

Once the essential concepts outlined in the background section have been understood, this section presents a thorough framework that can be used for the design of electrical machines using RL. It includes three key sub-sections: Electrical Machine Design Game, Reward Functions, and Method (Proximal Policy Optimization).

In the Electrical Machine Design Game sub-section (section 3.2), a novel approach is presented where the design process is transformed into a game-like environment. This game enables the application of RL techniques to optimize the design of induction machines. By formulating the design problem as a game, it becomes possible to define actions, states, and rewards, allowing for the application of RL algorithms.

The Reward Functions sub-section (section 3.3) discusses the crucial aspect of defining appropriate reward functions within the game. These reward functions act as guides to reinforce desirable design choices during the optimization process. By carefully designing reward functions, the RL agent learns to navigate the design space effectively, seeking designs that meet specific objectives and constraints.

The Method sub-section (section 3.4) focuses on the specific RL algorithm employed in the thesis, namely PPO. PPO is a state-of-the-art policy optimization method that strikes a balance between sample efficiency and stability. The section provides a detailed explanation of how PPO is utilized to train the RL agent within the electrical machine design game. It describes the training process and the optimization objectives employed.

Overall, the problem formulation section of the thesis presents an innovative framework for designing induction machines using RL. By transforming the design process into a game-like environment, defining appropriate reward functions, and employing PPO method, the thesis aims to optimize the design of electrical machines more efficiently and effectively.

3.1 Introduction

RL can be used to solve our electrical machine design problem (section 3.2) by training an agent to make decisions that optimize the mass of the machine. Here are the general steps involved in using RL to design an electrical machine game:

1. Define the environment: The first step is to define the environment in which the agent will operate, including the states that the agent can observe and the actions it can take (section 3.2). Furthermore, it is necessary to specify the reward function, which determines the rewards that the agent receives for taking certain actions in the environment (section 3.3).

2. Select a RL algorithm: Next, you will need to select a RL algorithm that will be used to train the agent. There are many different RL algorithms to choose from, each with its own strengths and weaknesses. Here, the decision was made to employ the Proximal Policy Optimization (PPO) method (section 3.4). As PPO is known for its effectiveness in optimizing RL policies, making

it a suitable choice for addressing the challenges of the electrical machine design problem. Additionally, PPO strikes a balance between exploration and exploitation, which is crucial when dealing with complex and high-dimensional design spaces.

3. Train the agent: Once the environment and RL algorithm have been defined, you can use the algorithm to train the agent by collecting experience from the environment and using this experience to update the policy. This process can be iteratively repeated until the agent's policy has converged to a satisfactory level of performance.

4. Evaluate the agent's performance: After the agent has been trained, you can evaluate its performance by running it in the environment and measuring its performance. This will help you to determine whether the agent is making good decisions and optimizing the performance of the electrical machine.

5. Fine-tune the agent's policy: If the agent's performance is not satisfactory, you can fine-tune the agent's policy by making adjustments to the environment, reward function, or RL algorithm. This can help to improve the agent's performance and optimize the performance of the electrical machine.

In this section, the focus will be placed on the chosen RL algorithm and its potential reward functions. The PPO algorithm has been selected as an effective and powerful on-policy model-free RL algorithm, utilizing an actor-critic approach (section 3.4). It can be used to design electrical machines, by training an agent to make decisions that optimize the mass of the machine. PPO needs a reward function to solve our electrical machine design game. Several different reward functions that can be utilized for PPO were proposed in section 3.3.

3.2 Our electrical machine design game

A RL game is a simulation where a RL agent interacts with an environment, with the goal of maximizing the overall reward through a series of time steps. The environment is usually modeled as a Markov Decision Process (MDP) and the agent's objective is to learn a strategy, which maps states to actions that maximizes the expected cumulative reward. The agent improves its strategy by interacting with the environment, receiving rewards and observations, and adjusting its strategy based on this experience.

Moreover, the game offers a way for the agent to assess how well it's doing. By engaging with the game environment, the agent can evaluate its current policy and use the feedback from its performance to improve and update its policy. This process of testing and updating the policy allows the agent to learn and improve over time. The game also provides an opportunity for the agent to generalize its knowledge and explore the problem space. As the agent interacts with the game environment, it can gather experiences that can be used to inform its actions in other parts of the environment. This allows the agent to apply what it has learned in one part of the environment to other parts, making it more efficient and effective in solving the problem.

A game may have a specific end point called a terminal state, and the objective for the player, or agent, is to reach it as quickly as possible. The game could be designed for one player or multiple players, and the agent may interact or compete with other players. Examples of multi agent games are robotics, traffic control, and energy management. Moreover, the game may have complex rules and obstacles that the agents must navigate and adapt to during play. The environment in which the game takes place may also be dynamic and change over time, requiring the agents to continuously update their strategies and adapt to new conditions. Therefore, the agents need to be able to perceive, adapt and react to the changes in the environment to ensure their survival and success.

3.2.1 Introduction

An electrical machine design game for RL is a simulation game in which a RL agent designs and optimizes electrical machines such as motors and generators. The agent interacts with the game environment in a sequence of time steps, with the goal of maximizing a cumulative reward. The game may have different levels, each with increasing complexity and difficulty. The player may start with a basic machine design and progress to more advanced designs as they gain experience and knowledge.

When designing the electrical machine, the player may need to take into account various factors such as the type of winding, the material and size of the rotor and stator, the number of poles, and the cooling system. They may also be required to optimize the design to achieve specific performance criteria, such as torque, power, and efficiency. In our simplified electrical design game, performance values such as airgap flux density, relative starting current, starting torque, and stator temperature rise are obtained. These performance values are directly or indirectly influenced by customer-specific requirements and manufacturing constraints.

The game also has a testing phase, where the player can evaluate their machine's performance under different operating conditions. Afterwards, based on the results of the tests, the player can make adjustments to the design and retest until they achieve the desired performance. In our game, there is a performance flag $\in \{-1, 0, 1\}$ for each of these performance values that checks one characteristic of the electrical machine. For every flag, if its relevant performance value meets the requirements, then the flag value is 0, otherwise it is either $+1$ or -1.

The feasibility of a machine design relies on all associated flags being zero, and thus, the primary objective of this work is to identify a set of design parameters where all flags are zero. Achieving this requires optimizing various performance values while striking a satisfactory trade-off among them. For instance, consider the trade-off between starting torque and starting current. Originally, if we want to lower the starting current, starting torque will be decreased at the same time. However, we want to have a low starting current with high starting torque so that we can start the motor even in a weak supply network.

Therefore, achieving optimal performance values is proved to be a complex task due to the intricate interconnections among them. Consequently, the optimization process is challenging since a single action may have a positive effect on one performance value while simultaneously having a negative impact on another. In Fig. 2, several examples are presented to demonstrate the variations in different performance values resulting from alterations in the design variables of coil turns and rotor tooth tip height. To highlight the trends in performance values associated with variations in design variables, the y-axis values have been omitted in Fig. 2.

In this game, the agent's actions are the design choices it makes. They are decreasing/increasing length of the motor, number of coil turns in the winding, and tooth tip height of the rotor slot. The states of the environment are the different configurations of the machine that result from the agent's actions. They mostly consist of the performance flags' values. The transition function defines the probability of transitioning from one state to another after taking a certain action, and the reward function defines the reward received by the agent for transitioning to a certain state.

The agent starts from an initial state S_0, which is a small base machine chosen based on the user specified requirements and the manufacturing constraints, and the final state S_T corresponds to the smallest feasible electrical machine that fits the given set of requirements and constraints. At each time step t, the agent begins in state S_t, takes an action A_t that changes the design variables, arrives at a new state (S_{t+1}), and receives a reward (R_{t+1}) that is calculated based on the performance values. Through repeated episodes (sequences of states, actions, and rewards), the policy network learns by trial and error to take the actions that maximizes the cumulative reward.

Let's take an in-depth look at the environment, states, and actions in our electrical machine design game.

3.2.2 Environment

In a RL game, the environment refers to the system or situation in which the agent (the RL algorithm) acts. It can be thought of as the "world" or "simulation" that is typically modeled as a Markov Decision Process (MDP) in which the agent makes decisions and receives rewards. The environment defines the states of the system, the actions that the agent can take, and the rules for transitioning between states and receiving rewards. The agent interacts with the environment by taking actions and observing the resulting states and rewards.

The ultimate goal for the agent is to learn a policy that maximizes the total reward over time. In some cases, the environment may also have a terminal state, which marks the end of an episode and the agent's goal is to reach that state as soon as possible. Additionally, there might also be some other states that agent should avoid and they can be defined by the reward function.

The environment was designed by constructing a black box model (state-space model) using simulation models of electrical motors. Then, both the training process and the evaluation of the RL agent are supported by this model. The environment interacts with the agent includes the reward function and state transformations. In our simplified problem, a state is represented by a vector primarily comprised of flag values and the preceding action. In some cases, the incorporation of performance values into the state space may also be considered. The action space is discrete and consists of decreasing or increasing each design variable (length of the machine, rotor tooth tip height and number of coil turns), that is in total 6 different actions.

First, an initial state S_0 is provided that is the initial seed motor calculated based on the user requirements and the manufacturing constraints. Then, the agent takes the state S_0 as an input and performs the action A_0 from the action space. The new electrical motor after applying the chosen action gives the new state S_1. Then, the reward R_1 is calculated by considering the non zero flags as well as the direction of the action with respect to the performance values. The agent continues this process until it reaches to the final state S_T where it either wins or loses the game.

3.3 Reward functions

In our electrical machine design game, the reward function is a function that maps the current state of the system and the taken actions to a scalar value that represents the reward or cost associated with those actions. The reward function is used to guide the learning process by providing a measure of the desirability of the current state and actions. The importance of a well-designed reward function cannot be overstated when it comes to the effectiveness of RL algorithms. Thus, the contribution of this thesis is design of the reward function. The reward function should correlate strongly with the desired system performance and guide the learning algorithm towards favorable actions.

In our simplified electrical machine design problem, the performance values are rotor tooth tip height, airgap flux density, relative starting current, starting torque, and stator temperature rise. These performance values are directly or indirectly related to customer specific requirements and manufacturing constraints. As mentioned in section 3.2.1, there is a performance flag $\in \{-1, 0, +1\}$ for each of these performance values. For every flag, if its relevant performance value meets the requirements, then the flag value is 0, otherwise it is either $+1$ or -1. The value of each flag indicates whether the action should decrease or increase its corresponding performance value. As an example, if *torque* flag is $+1$, the action that increases the *starting torque* performance value should give a positive reward and the action that decreases it should give a negative penalty.

In electrical machine design context, the reward function can be tied to these specific performance characteristics, aiming to optimize machine performance through the RL algorithm. Consequently, the consideration of the flags' values instead of the performance values can be taken into account when designing the reward function. RL policies need a large number of examples, often hundreds of thousands, to work effectively. This means that the reward function must be evaluated quickly, ideally within a few milliseconds.

The design of an effective reward function is a crucial and challenging aspect of achieving successful learning outcomes. This thesis examines various reward functions and highlights the most relevant ones. In all the proposed reward function, the winning and the losing criterias are the same. The agent loses the game if it reaches a fixed maximum number of total steps, M, or takes invalid actions. To ensure the feasibility of the agent's actions, any actions that violate the specified requirements and constraints are categorized as invalid. Additionally, the agent wins the game when all the flags are zero.

3.3.1 Simple reward function

Since, our objective is to design an electrical motor based on the given requirements and constraints, the true reward is initially calculated using the values of performance flags. Therefore, for the simplicity, our objective at each time step t is to increase the count of flags with a value of zero relative to the previous step. Accordingly, the reward associated with flags, denoted as $R_t(F)$, can be calculated as follows:

$$R_t(F) = \#\text{current zero flags} - \#\text{previous zero flags} \tag{29}$$

However, depending exclusively on the objective of increasing the count of flags with a zero value is deemed inadequate since it may not offer sufficient guidance for the agent to efficiently address the problem at hand. This may not facilitate meaningful learning and can potentially lead the agent to become trapped in repetitive actions. Moreover, it increases the likelihood of the agent failing to explore alternative approaches that could potentially yield viable solutions. Consequently, the agent may remain unable to achieve goals of the game. To overcome these limitations, it is necessary to incorporate additional factors or rewards that encourage the agent to explore diverse strategies, consider the global context, and discover more optimal pathways towards solving the problem. By doing so, the agent's learning process can become more comprehensive, enabling it to make informed decisions and increase the likelihood of successful outcomes.

Therefore, it becomes imperative to include supplementary considerations in order to effectively determine the agent's success or failure in the game. In the new framework, the agent's win in the game is rewarded with a substantial reward when all flags are set to zero. Conversely, the agent incurs a significant penalty, resulting in game loss, if it exceeds a predetermined maximum number of total steps (M) or performs invalid actions. By taking these additional factors into account, we can guide the agent towards a sequence of actions that will ultimately result in successful gameplay outcomes.

The corresponding reward, denoted as $R_t(G)$, can be defined in the following manner:

$$R_t(G) = \begin{cases} G_p & \text{if the agent wins the game} \\ -G_n & \text{if the agent loses the game} \end{cases} \tag{30}$$

where, the goal reward $R_t(G)$ makes a contribution with a positive constant G_p if the agent wins the game, and the losing reward would be a negative constant $-G_n$.

However, even with this reward structure, there is a risk of the agent getting stuck in repetitive loops and struggling to discover the optimal machine design. This challenge arises from the complexity of the optimization landscape, where the agent may encounter local optima or suboptimal solutions that hinder its progress towards finding the global optimum. To address this issue, additional strategies can be employed to promote exploration and diversify the agent's actions. Techniques such as regularization, stochasticity in action selection, or introducing randomness in the reward function can encourage the agent to explore different design possibilities and avoid being trapped in suboptimal solutions.

Therefore, a negative penalty is introduced, denoted as $-V_n$, that helps the agent to select better actions and explicitly aims to prevent the agent from getting trapped in repetitive action loops. This is achieved by using a caching mechanism that verifies whether the current state S_t has been previously encountered. If it has, a constant negative penalty $-V_n$ is applied to discourage the agent from revisiting the same state. Conversely, if the state is new, it is added to the cache, and the reward is calculated based on the reward sub-functions $R_t(G)$ and $R_t(F)$. Consequently, the final reward function can be defined as follows:

$$R_t = \begin{cases} -V_n & \text{if } S_t \text{ has already been seen during training} \\ R_t(G) + R_t(F) & \text{otherwise} \end{cases} \tag{31}$$

Moreover, algorithm 1 presents a comprehensive summary of the simple reward function, outlining the steps and procedures involved in the process.

Algorithm 1 Simple reward function

Input: current timestep t; current state S_t; current action A_t; previous state S_{t-1}.
Initialize: visited_states := [].
Output: R_t.

```
1:  if $S_t$ is in visited_states then
2:      return $-V_n$
3:  end if
4:  Add $S_t$ to visited_states.

5:  $R_F$ = #current zero flags - #previous zero flags

6:  if All flags are set to zero then
7:      $R_G = G_p$
8:  else if Maximum number of steps is reached then
9:      $R_G = -G_n$
10: else if $A_t$ is not valid then
11:     $R_G = -G_n$
12: end if

13: $R_t = R_F + R_G$
14: return $R_t$
```

The proposed reward function acts as a metric that reflects the agent's advancement in achieving a feasible design configuration. This straightforward reward function provides a physical framework for assessing the agent's design choices by quantifying their impacts on performance metrics. By optimizing the performance values using the flag reward function, the agent can effectively steer the design process towards achieving superior electrical machine configurations with improved mass, and other desired performance characteristics.

The reward function proposed in this section demonstrates simplicity, ease of implementation, and computational efficiency, all while remaining agnostic to specific engineering knowledge. However, its simplicity may limit its applicability to a diverse range of cases, potentially rendering it less widely suitable. Consequently, there is a need to develop an enhanced reward function that incorporates pertinent engineering knowledge, addressing the limitations of the current approach and expanding its scope to offer solutions for a wider array of scenarios.

3.3.2 Human expert knowledge based reward function

The human expert knowledge based reward function adopts a more intricate approach that evaluates each flag value in relation to its corresponding performance metric on an individual basis. This requires defining a reward function that takes into account the value of each flag, rather than solely focusing on the total number of zero flags. By considering each flag value separately, we gain a deeper understanding of its impact on the overall performance of the electrical machine. This allows us to design a reward function that reflects the relative importance of each performance metric and guides the agent towards configurations that optimize specific aspects of the machine's performance.

In our game, the performance values, along with their corresponding flags, are possessed. Each flag's value ($+1$ or -1) indicates whether the corresponding performance value should decrease or increase when an action is taken. This reward function takes into account each flag individually to provide specific guidance for the agent's design decisions. To this end, we assign rewards based on the direction in which the corresponding performance value should be modified. For a flag associated with a performance metric, such as $performance$, we assign a positive reward ($r_p > 0$) if the corresponding performance value is modified in the correct direction. Conversely, if the performance value is modified in the opposite direction, a negative reward ($-r_n < 0$) is given. For example, if the flag for *torque* has a value of $+1$, performing an action that increases the torque performance value will result in a positive reward ($r_{torque} = r_p$), while performing an action that decreases it will yield a negative reward ($r_{torque} = -r_n$). This procedure is applied to the performance metrics of rotor tooth tip height, airgap flux density, relative starting current, starting torque, and stator temperature rise.

To construct a unified performance reward function, denoted as R_P, that considers multiple objectives, we sum up the rewards for all flags. This consolidation allows the agent to navigate the design space while simultaneously considering the impact of its actions on various performance metrics, ultimately guiding it towards configurations that optimize multiple objectives simultaneously.

$$\begin{aligned} R_p &= \sum_{performance} r_{performance} \\ &= r_{tooth\_tip} + r_{flux\_density} + r_{current} + r_{torque} + r_{temprature} \end{aligned} \tag{32}$$

Moreover, a supplementary requirement that impacts the above mentioned performance reward function R_p is associated with the flag corresponding to the starting current performance value. If the current flag is not zero and the action taken is in the appropriate direction, but the torque flag has been raised (changes from 0 to either $+1$ or -1), no reward or penalty will be allocated to the current flag. Specifically, $r_{current} = 0$ in (32). By incorporating this detailed approach, the reward function aligns more closely with engineering knowledge and expertise. It enables the agent to make informed choices by comprehensively evaluating the impact of its actions on individual performance metrics. Consequently, the agent can navigate the design space more effectively, identifying configurations that excel in specific performance aspects while balancing trade-offs between different metrics.

However, it is important to note that certain priorities are defined among the performance values, and these priorities should be taken into consideration during the design process. Each performance metric may have a varying degree of importance or significance based on the specific requirements and objectives of the electrical machine design. Therefore, by explicitly incorporating the prioritization of performance values into the reward function, the agent can intelligently allocate its resources and decision-making capabilities towards achieving the desired objectives. This allows for a more efficient and effective design process, where the agent can navigate the design space with a clear understanding of the relative importance of each performance metric and the corresponding flags. By acknowledging and integrating these priorities, the agent can make informed design decisions and optimize the machine's performance in a targeted and purposeful manner.

As a result, the function $R_{priority}$ is linked to a specific priority, giving the highest priority to the flag associated with the rotor tooth tip height performance value over any other flags. As a consequence, if this flag is raised (switches from 0 to -1 or 1), the corresponding reward value, $R_{priority}$, would become negative, resulting in the reward being skipped for all other flags. Additionally, if the flag for rotor tooth tip height is not equal to zero, specific considerations denoted by R_{tooth_tip} need to be taken into account. As represented in (33), when the flag is set to 1, and the action is taken to decrease the tooth tip, the corresponding reward would become positive (r_p). Conversely, if the action does not involve decreasing the tooth tip, the reward value would become negative ($-r_n$). Similarly, when the flag is set to -1, and the action is taken to increase the tooth tip, the corresponding reward would become positive (r_p). In contrast, if the action does not involve increasing the tooth tip, the reward value would become negative ($-r_n$).

$$R_{tooth\_tip} = \begin{cases} r_p & \text{if } f_{tooth\_tip} = 1 \text{ and action decreases the tooth tip height} \\ -r_n & \text{if } f_{tooth\_tip} = 1 \text{ and action increases the tooth tip height} \\ r_p & \text{if } f_{tooth\_tip} = -1 \text{ and action increases the tooth tip height} \\ -r_n & \text{if } f_{tooth\_tip} = -1 \text{ and action decreases the tooth tip height} \end{cases} \tag{33}$$

where f_{tooth_tip} stands for rotor tooth tip height flag. If any of these situations related to the rotor tooth tip flag happens, the rewards for all other flags are also skipped.

In conclusion, the corresponding performance reward function can be defined as follows:

$$R_t(P) = \begin{cases} R_{priority} & \text{if rotor tooth tip height flag is raised} \\ R_{tooth\_tip} & \text{if rotor tooth tip height flag is not 0} \\ R_p & \text{otherwise} \end{cases} \tag{34}$$

Moreover, in order to satisfy the target relative starting current (I_s/I_n, where I_s is the starting current and I_n is the nominal current) requirement, an additional component called the I_s/I_n reward function, represented as $R_t(\mathcal{I})$, is integrated. The target I_s/I_n sets an upper limit that each state's corresponding I_s/I_n should not exceed. Therefore, when both the current and previous I_s/I_n values are lower than the target I_s/I_n, no reward or penalty is assigned. If the previous I_s/I_n was higher than the target I_s/I_n, it indicates that actions should not increase the I_s/I_n. Thus, if the current I_s/I_n is lower than the previous one, a positive reward is granted. Conversely, if the current I_s/I_n is higher than the previous one, a negative penalty is applied.

$$R_t(\mathcal{I}) = \begin{cases} 0 & \text{if } \mathcal{I}_{t-1} < \mathcal{I}_{tr} \text{ and } \mathcal{I}_t < \mathcal{I}_{tr} \\ r_p & \text{if } \mathcal{I}_{t-1} > \mathcal{I}_{tr} \text{ and } \mathcal{I}_t < \mathcal{I}_{t-1} \\ r_n & \text{otherwise} \end{cases} \tag{35}$$

where $\mathcal{I}_t$ represent the current I_s/I_n, $\mathcal{I}_{t-1}$ denote the previous I_s/I_n, and $\mathcal{I}_{tr}$ represent the target I_s/I_n.

In summary, the resulting human expert knowledge based reward function can be expressed as follows:

$$R_t = \begin{cases} -V_n & \text{if } S_t \text{ has already been seen during training} \\ R_{priority} + R_t(G) + R_t(F) + R_t(\mathcal{I}) & \text{if rotor tooth tip height flag is raised} \\ R_{tooth\_tip} + R_t(G) + R_t(F) + R_t(\mathcal{I}) & \text{if rotor tooth tip height flag is not 0} \\ R_p + R_t(G) + R_t(F) + R_t(\mathcal{I}) & \text{otherwise} \end{cases} \tag{36}$$

where, the selected constant values for the reward functions are presented in Table 1. The steps and procedures involved in the process of the human expert knowledge reward function are outlined in Algorithm 2, offering a comprehensive summary of the approach.

By expanding the reward function to consider each flag value individually, we enhance its ability to capture the intricacies of electrical machine design. This refined

Algorithm 2 Human expert knowledge based reward function

Input: current timestep t; current state S_t; current action A_t; previous state S_{t-1}; current I_s/I_n $\mathcal{I}_t$; target I_s/I_n $\mathcal{I}_{tr}$; previous I_s/I_n $\mathcal{I}_{t-1}$.
Initialize: visited_states := [].
Output: R_t.

```
 1: if $S_t$ is in visited_states then
 2:     return $-V_n$
 3: end if
 4: Add $S_t$ to visited_states.

 5: $R_F$ = #current zero flags - #previous zero flags

 6: if All flags are set to zero then
 7:     $R_G = G_p$
 8: else if Maximum number of steps is reached then
 9:     $R_G = -G_n$
10: else if $A_t$ is not valid then
11:     $R_G = -G_n$
12: end if

13: if Rotor tooth tip height flag is raised then
14:     $R_P = R_{priority}$
15: else if Rotor tooth tip height flag is not 0 then
16:     $R_P = R_{tooth\_tip}$ (33)
17: else
18:     $R_P = 0$
19:     for performance do
20:         $R_P = R_P + r_{performance}$
21:     end for
22: end if

23: if $\mathcal{I}_{t-1} < \mathcal{I}_{tr}$ and $\mathcal{I}_t < \mathcal{I}_{tr}$ then
24:     $R_{\mathcal{I}} = 0$
25: else if $\mathcal{I}_{t-1} > \mathcal{I}_{tr}$ and $\mathcal{I}_t < \mathcal{I}_{t-1}$ then
26:     $R_{\mathcal{I}} = r_p$
27: else
28:     $R_{\mathcal{I}} = -r_n$
29: end if

30: $R_t = R_F + R_G + R_P + R_{\mathcal{I}}$
31: return $R_t$
```

Table 1: Constants for reward functions.

Metric	Value
$-V_n$	-0.55
r_p	+0.3
$-r_n$	-0.3
$R_{priority}$	-0.4

reward function empowers the agent to explore a broader range of design possibilities, ultimately leading to more sophisticated and optimized machine configurations.

3.3.3 Distance reward function

In this section, a completely different reward function is employed which does not devoid of any engineering knowledge or domain-specific information. Within this reward function, we are equipped with a training set comprising various electrical machine design games, each encompassing distinct customer requirements, power and voltage specifications, alongside their corresponding final designs, referred as the goal state. Therefore, the goal state in this context represents a feasible or optimal machine configuration that meets the customer specific requirements and the manufacturing constraints. It should be noted that these goal states are exclusively accessible within the training set.

This reward function is based on the distance between the current state and the goal state. The availability of the goal state for all training machines enables the design of the distance reward function. By utilizing this new reward function, the RL agent's learning process is guided towards achieving states that are closer to the desired goal state. This approach allows the RL algorithm to effectively explore and optimize the machine design space by motivating progress towards the desired outcome.

Machine design variables, represented by $\mathcal{M}$, are the length, the number of coil turns, and the rotor tooth tip height of the machine. Therefore, at each time step $t > 0$, we possess the current state S_t along with the corresponding machine values $\mathcal{M}_t$, the preceding state S_{t-1} with the corresponding machine values $\mathcal{M}_{t-1}$, and the goal state S_G alongside the corresponding machine values $\mathcal{M}_G$. With access to this information, the machine values can be effectively tracked and compared at different time steps, allowing the progress and changes in the system's configuration to be monitored.

In this particular reward function formulation, the initial step involves calculating the distances between the current machine values $\mathcal{M}_t$ and the desired goal machine values $\mathcal{M}_G$, as well as the distance between the previous machine values $\mathcal{M}_{t-1}$ and the goal machine values $\mathcal{M}_G$. These distances serve as fundamental metrics for evaluating the alignment between the current or previous machine configurations and the desired goal configuration. By comparing these distances, we can derive the distance reward function, represented as $R_t(D)$.

To compute this distance reward function, the steps outlined in Algorithm 3 can be followed. By utilizing this algorithm, the distance reward function provides a quantitative assessment that significantly impacts the behavior of the RL agent. It encourages the agent to take actions that bring the machine values closer to the desired goal values, while discouraging actions that lead to greater deviations. Consequently, the reward function plays a vital role in shaping the learning process of the agent, directing its focus towards achieving machine configurations that are more compatible with the desired objectives. By providing a clear feedback signal, the distance reward function assists the agent in effectively navigating the search space and acquiring optimal strategies to attain the desired machine configurations.

Finally, the resulting expression of the reward function is as follows

$$R_t = \begin{cases} -V_n & \text{if } S_t \text{ has already been seen during training} \\ R_t(G) + R_t(D) & \text{otherwise} \end{cases} \tag{37}$$

where $V_n = -1.1$.

It's worth mentioning that designing and calculating this function does not require any prior engineering expertise. The process is straightforward and can be accomplished without much difficulty. It provides a measure of how close the machine values are to the goal machine values. By using this function, the assessment of the machine's performance in terms of meeting the desired goals can be determined without relying on any engineering knowledge.

3.4 Proximal Policy Optimization (PPO)

To train an agent to make optimal decisions in an electrical machine design game, PPO is a suitable RL algorithm. PPO has gained recognition as a state-of-the-art algorithm that offers several advantages for optimizing complex problems. Its ability to strike a balance between exploration and exploitation, along with its robustness in handling high-dimensional and continuous action spaces, make it particularly suitable for the challenges posed by electrical machine design. By selecting PPO, the study aims to leverage its capabilities to effectively guide the agent's learning process and achieve optimal designs for electrical machines.

In this context, the objective is to optimize the mass of the electrical machine. To design such a game using PPO, the first step, as described in Section 3.2.2, involves defining the environment in which the agent will operate. Subsequently, the reward function, as discussed in Section 3.3, needs to be established. Once these prerequisites are in place, PPO can be utilized to train the agent by gathering experience from the environment and updating the policy accordingly. This iterative process can be continued until the agent's policy has converged to a satisfactory level of performance.

PPO [3] is a RL method that falls under the model-free and actor-critic category. Its primary objective is to train agents to make optimal decisions in complex environments, regardless of whether the action space is continuous or discrete. This is an algorithm that helps an agent make the best decision based on its previous experiences and the rewards it has received so far. It is an on-policy method, meaning

Algorithm 3 Distance reward function

Input: current timestep t; current state S_t; current action A_t; current machine values $\mathcal{M}_t$; previous machine values $\mathcal{M}_{t-1}$; goal machine values $\mathcal{M}_G$.
Initialize visited_states := []; $R_D := 0; k := 2$.
Output: R_t.

```
 1: if $S_t$ is in visited_states then
 2:     return $-V_n$
 3: end if
 4: Add $S_t$ to visited_states.

 5: for $m \in \mathcal{M}$ do
 6:     $prev\_dist = |m_G - m_{t-1}|$
 7:     $current\_dist = |m_G - m_t|$
 8:     $change = |current\_dist - prev\_dist|$
 9:     if $prev\_dist = 0$ and $current\_dist = 0$ then
10:         r = 0
11:     else if $prev\_dist = 0$ and $current\_dist \neq 0$ then
12:         r = -1
13:     else if $current\_dist > prev\_dist$ then
14:         r = $-(min(\frac{change}{prev\_dist}, 1))^k$
15:     else if $current\_dist < prev\_dist$ then
16:         r = $(\frac{change}{prev\_dist})^k$
17:     else
18:         r = 0
19:     end if
20:     $R_D = R_D + r$
21: end for

22: if $R_D < 0$ then
23:     $R_D = -|R_D|^{\frac{1}{k}}$
24: else
25:     $R_D = R_D^{\frac{1}{k}}$
26: end if

27: if All flags are set to zero then
28:     $R_G = G_p$
29: else if Maximum number of steps is reached then
30:     $R_G = -G_n$
31: else if $A_t$ is not valid then
32:     $R_G = -G_n$
33: end if

34: $R_t = R_G + R_D$
35: return $R_t$
```

that it updates the agent's decision-making policy based on the actions it has taken in the past, rather than using a separate set of experiences to update the policy. Additionally, PPO uses a policy gradient method, which means that it directly optimizes the policy, instead of estimating the value function or state-action pairs. In simple terms, PPO helps the agent learn by adjusting its behavior in real-time, using the rewards it receives and the actions it takes.

3.4.1 PPO applications

PPO is a popular and effective RL algorithm that has been applied to a wide variety of tasks and environments. Some of the applications of PPO include but not limited to:

- Robotics: PPO has been used to train robots to perform a variety of tasks, such as grasping objects, navigating through environments, and following moving targets [56].
- Control systems: PPO has been used to optimize control systems in a variety of applications, including power systems, and process control [57].
- Game playing: PPO has been used to train agents to play games, such as chess and Go, at a high levels of skills [58].
- Autonomous vehicles: PPO has been used to train self-driving cars and other autonomous vehicles to make decisions in complex environments [59].
- Natural language processing: PPO has been used to train natural language processing systems to understand and generate human-like texts [60].

3.4.2 PPO basic idea

The basic idea behind PPO is to iteratively update the agent's policy in order to improve its performance. This is done by collecting experience from the environment, and using this experience to update the agent's policy.

Here is an outline of the basic steps involved in the PPO algorithm:

1. Collect experience from the environment: The agent interacts with the environment and collects experience by taking actions and observing the resulting states and rewards.
2. Estimate the value function: The value function is a tool that estimates the total reward that an agent is likely to receive for being in a certain state. This value function is used to assess the desirability of the agent's actions and to guide the optimization of the agent's policy.
3. Update the policy: The policy of an agent is a set of rules that dictate its behavior in a given environment. The PPO algorithm updates the policy based

on the agent's experience and the estimated value function, using an optimization algorithm that is guided by the clipped surrogate objective function. This ensures that the policy is improved in a controlled manner, balancing the need for significant improvements in performance with the need for stable learning.

4. Evaluate the updated policy: The updated policy is evaluated by running the agent in the environment and measuring its performance. This helps to ensure that the policy is actually improving the agent's performance and is not just overfitting to the collected experience.

5. Repeat the process: The process is repeated until the policy has converged to a satisfactory level of performance.

Overall, the PPO algorithm works by iteratively improving the policy through a process of collecting experience, estimating the value function, and updating the policy by using policy gradient method. The end result is a policy that is able to maximize the expected cumulative reward for the agent in the environment.

3.4.3 Policy gradient methods

The policy gradient method [61] is a technique to improve a policy in RL. A policy is like a set of rules that tell the agent what actions to take in certain situations. In this method, the policy is represented as a probability distribution over actions given states, $\pi(a|s)$. The policy has parameters (θ) that can be adjusted to change how the agent behaves. To optimize the policy, data is collected on the actions and rewards of the agent while it employs the policy. This data is then utilized to estimate the impact of policy parameter changes on the expected reward. By adjusting the policy parameters in a manner that increases the expected reward, the policy can be improved.

The approach of the policy gradient method in RL that updates the policy using samples obtained from the current policy, makes it an on-policy algorithm. Policy gradient algorithms come in different types, such as finite difference methods [62] that introduce small perturbations to the policy parameters and observe the changes, or likelihood ratio policy gradient methods [63] that employ the likelihood ratio trick to estimate the gradient of the expected reward with respect to the policy parameters. The focus of this section is on the likelihood ratio policy gradient methods.

In the likelihood ratio policy gradient methods, we go along a state-action trajectory $\tau = (S_0, A_0, ..., S_{T-1}, A_{T-1}, S_T)$ when making sequential decisions. A cumulative reward $R(\tau) = \sum_{t=0}^{T-1} \gamma^t R_{t+1}$ (series of rewards R_{t+1} discounted by γ) and a corresponding probability $P(\tau)$ are associated with each trajectory. Via sampling actions, the policy affects the likelihood that we observe each set of states and actions over a certain period of time.

The ultimate goal of policy gradient methods, like all RL approaches, is to discover the optimal policy that increases the expected cumulative reward in the long run. In a completely deterministic setting, the trajectory generated by each policy π_θ could be calculated effortlessly, and the one that yielded the highest overall payoff could be identified. However, most RL problems have a stochastic component and are not entirely deterministic. As a result, the policy has an impact on but does not entirely affect trajectory probabilities.

Therefore, given our policies, the likelihood of a specific reward trajectory occurring is calculated as

$$P(\tau, \theta) = \Big[\prod_{t=0}^{T-1} P(S_{t+1}|S_t, A_t).\pi_\theta(A_t|S_t) \Big] \tag{38}$$

The transition function $P(S_{t+1}|S_t, A_t)$ is a problematic element in this scenario. It is a representation of the environment which is at best challenging to work with and at worst completely unknowable. Another disadvantage of this model is that $P(\tau, \theta)$ is the result of two probabilities. If those probabilities are low, then over large time horizons the trajectory probabilities will become extremely small.

The components discussed earlier enable us to formally state our goal, which is to maximize the expected reward over time. Consequently, the probabilities of all trajectories are summed up and multiplied by their corresponding rewards to define the objective function.

$$J(\theta) = \mathbf{E}_{\tau \sim \pi_\theta}[R(\tau)] = \sum_{\tau} P(\tau, \theta) R(\tau) \tag{39}$$

where the distribution of trajectories under the current policy is formalized by $\tau \sim \pi_\theta$. Therefore the equivalent maximizing problem is represented by:

$$\begin{aligned} \max_\theta J(\theta) &= \max_\theta \sum_\tau P(\tau, \theta) R(\tau) \\ &\overset{(38)}{=} \max_\theta \sum_\tau \Big[\prod_{t=0}^{T-1} P(S_{t+1}|S_t, A_t).\pi_\theta(A_t|S_t) \Big] R(\tau) \end{aligned} \tag{40}$$

It is evident from the above maximization problem that altering θ impacts the trajectory probability. Remember that the policy π_θ has an impact on this objective term as well. Therefore, the expected return can be improved by increasing the probabilities of high reward trajectories.

Policy gradient approaches depend on gradients to determine the direction of the update that will improve the goal function (expected return). The resulting update function is

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ J(\theta) \tag{41}$$

where

$$\begin{aligned}
\nabla_\theta\ J(\theta) &= \nabla_\theta \sum_\tau P(\tau,\theta)R(\tau) \\
&= \sum_\tau \nabla_\theta P(\tau,\theta)R(\tau) \\
&= \sum_\tau \frac{P(\tau,\theta)}{P(\tau,\theta)} \nabla_\theta P(\tau,\theta)R(\tau) \\
&= \sum_\tau P(\tau,\theta)\frac{\nabla_\theta P(\tau,\theta)}{P(\tau,\theta)} R(\tau) \\
&= \sum_\tau P(\tau,\theta)\nabla_\theta \log P(\tau,\theta)R(\tau)
\end{aligned} \tag{42}$$

The empirical estimate for m sample trajectories under policy π_θ can be utilized to approximate $\nabla_\theta\ J(\theta)$ by

$$\nabla_\theta\ J(\theta) \approx \hat{g} := \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)},\theta)R(\tau^{(i)}) \tag{43}$$

This expression is entirely tractable. Even if we don't know the environment model, we can sample the trajectories and associated rewards. All we require is a clearly stated policy that can be differentiated in relation to θ. This gradient tries to increase the probability of paths with high (positive) rewards and decrease the probabilities of trajectories with low (negative) rewards. As the sum of the probabilities are 1, so pushing up some probabilities (good ones) leads to pushing down some others (bad ones).

To change the probabilities of the trajectories ($\log P(\tau^{(i)},\theta)$) we only need to change the probabilities of the actions along that trajectory. Recall that from equation (38) we have:

$$\begin{aligned}
\nabla_\theta \log P(\tau^{(i)},\theta) &= \nabla_\theta \log \Big[\prod_{t=0}^{T-1} P(S_{t+1}^{(i)}|S_t^{(i)},A_t^{(i)}).\pi_\theta(A_t^{(i)}|S_t^{(i)}) \Big] \\
&= \nabla_\theta \Big[\sum_{t=0}^{T-1} \log P(S_{t+1}^{(i)}|S_t^{(i)},A_t^{(i)}) + \sum_{t=0}^{T-1} \log \pi_\theta(A_t^{(i)}|S_t^{(i)}) \Big]
\end{aligned} \tag{44}$$

as the first part ($\log P(S_{t+1}^{(i)}|S_t^{(i)},A_t^{(i)})$) doesn't have any factors of θ, it can be eliminated:

$$\begin{aligned}
\nabla_\theta \log P(\tau^{(i)},\theta) &= \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(A_t^{(i)}|S_t^{(i)}) \\
&= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)})
\end{aligned} \tag{45}$$

Therefore, decreasing the probabilities of a trajectory, will make the actions along in that trajectory less likely to be taken. As the probabilities of the trajectory are inversely proportional to the likelihood of the actions within the trajectory, decreasing the probabilities will decrease the likelihood of those actions being taken. This will also affect the expected cumulative reward in the long run, as the trajectory with decreased probability may have a lower expected reward.

In conclusion, by considering all the above equations we will have:

$$\begin{aligned}\nabla_\theta\ J(\theta) \approx \hat{g} &\overset{(43)}{=} \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)}, \theta) R(\tau^{(i)}) \\ &\overset{(45)}{=} \frac{1}{m}\sum_{i=1}^{m} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)})\right) R(\tau^{(i)}) \\ &= \frac{1}{m}\sum_{i=1}^{m} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)})\right) \left(\sum_{t=0}^{T-1} \gamma^t R_{t+1}\right)\end{aligned} \tag{46}$$

Using the gradient formula to make a move results in an increase in the log-probabilities of each action, which is proportional to the total sum of rewards obtained, denoted by $R(\tau^{(i)})$. However, this approach is not rational since agents should only reinforce actions based on their outcomes, i.e., the rewards obtained after the action is taken. The rewards obtained before taking the action are irrelevant in determining the effectiveness of that action. Therefore, it is necessary to consider only the post-action rewards to evaluate the goodness of the action. It can be mathematically shown that this intuition is reflected in the policy gradient, which can also be expressed as:

$$\nabla_\theta\ J(\theta) \approx \hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)}) \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{47}$$

The current formulation is unbiased but highly noisy because of the empirical returns. This high level of noise stems from the inherent stochastic nature of RL algorithms, leading to fluctuations in the estimated gradients. To address this issue, a baseline function (b) will be introduced, typically serving as a control variate.

$$\nabla_\theta\ J(\theta) \approx \hat{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)}, \theta)(R(\tau^{(i)}) - b) \tag{48}$$

As long as baseline does not depend on $\log P(\tau, \theta)$, the expected value of the extra term would be 0:

$$\begin{aligned}\mathbf{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau,\theta) b] &= \sum_\tau P(\tau,\theta)\nabla_\theta \log P(\tau,\theta) b \\ &= \sum_\tau P(\tau,\theta)\frac{\nabla_\theta P(\tau,\theta)}{P(\tau,\theta)} b \\ &= \sum_\tau \nabla_\theta P(\tau,\theta) b \\ &= \nabla_\theta \left(\sum_\tau P(\tau,\theta) b\right) \\ &= b\nabla_\theta \left(\sum_\tau P(\tau,\theta)\right) \\ &= b\nabla_\theta\,(1) \\ &= b \times 0 = 0\end{aligned} \tag{49}$$

Whereas, when there is a finite number of samples, the estimate we are accumulating will have a decrease in variance and with an appropriate baseline, the variance will be further reduced. This results in a more accurate gradient estimate.

Examples of baselines include a constant baseline, a baseline that changes over time, and a baseline that depends on the current state. When using a state-dependent baseline ($b(S_t) = V_{\pi_\theta}(S_t)$), the log probability of an action is increased in proportion to how much better its returns are compared to the expected return under the current policy.

$$\begin{aligned}
\nabla_\theta\ J(\theta) \approx \hat{g} &\overset{(48)}{=} \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)},\theta)(R(\tau^{(i)}) - b) \\
&= \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)},\theta)(R(\tau^{(i)}) - V_{\pi_\theta}(S_t^{(i)})) \\
&\overset{(47)}{=} \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)})(\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} - V_{\pi_\theta}(S_t^{(i)})) \\
&= \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)})(Q_{\pi_\theta}(S_t^{(i)},A_t^{(i)}) - V_{\pi_\theta}(S_t^{(i)})) \\
&\overset{(14)}{=} \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^{(i)}|S_t^{(i)})\mathcal{A}_{\pi_\theta}(S_t^{(i)},A_t^{(i)})
\end{aligned} \tag{50}$$

Where $\mathcal{A}_{\pi_\theta}(S_t^{(i)}, A_t^{(i)})$ is the advantage function 2.2.3.

In policy gradient methods, despite having a reward signal and an update direction, we are unsure of how much the policy needs to be updated. On the other words, finding the ideal learning rate (α) is difficult. Policy gradients have also the following flaws:

- Sample inefficiency: Samples are only utilized once. The policy is then revised, and a fresh trajectory is sampled using the new policy. This may be unaffordable due to how expensive sampling is frequently. The old samples, however, are in fact no longer representative following a significant policy shift.

- Inconsistent policy updates: Policy adjustments have a tendency to overreach and miss the reward peak or to peter out too soon. Vanishing gradients is a serious issue, particularly in neural network topologies.

- High reward variance: The policy gradient method uses Monte Carlo learning and considers the entire reward trajectory (i.e., a full episode). These trajectories frequently have considerable volatility, which hinders convergence.

PPO is a variant of the policy gradient method that uses a function that is close to the current policy the agent takes actions based on, and it does not make big changes all at once. Instead, it makes smaller changes to the function by using an easier-to-optimize function. PPO also uses a technique called trust region policy optimization (TRPO), which makes sure that the new policy is not too different from the old one. This helps to make the learning process more stable and prevents the agent from making too many mistakes or getting stuck in a bad policy.

3.4.4 Trust region policy optimization (TRPO)

TRPO is a type of RL algorithm that belongs to the category of policy optimization methods. It uses a technique called trust region approach, which involves selecting the largest possible change that the algorithm will make to the policy parameters. This is done within a specific area called the trust region. Once the largest change is selected, the algorithm then searches for the best spot within that trust region to make the change. This approach helps to prevent the algorithm from making overly large changes to the policy, which can lead to instability in the learning process. You can see an illustration of this approach in Figure 7 [source].

The trust region method defines a range of policies that have a small impact on the objective function, which is measured using a mathematical concept called KL divergence [64]. This range of policies is known as the trust region and it is centered around the current policy being used. Once the trust region is defined, the algorithm restricts the search for the optimal point, such as a local minimum or maximum, within that specific area. After the best point is found, the algorithm establishes the direction for further improvement. This iterative process continues until the optimal solution is achieved.

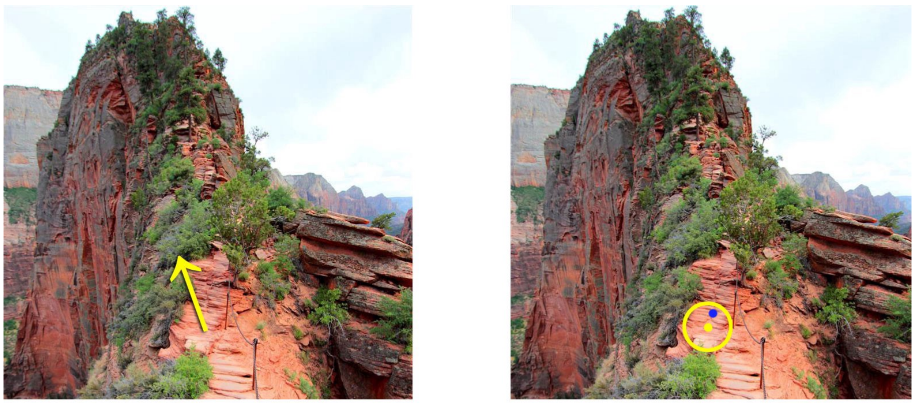


Figure 7: The parameter update in the TRPO.

For doing so, from equation (39) recall that

$$J(\theta) = \mathbf{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbf{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{\infty}\gamma^t R_{t+1}] \tag{51}$$

We can add the anticipated advantage of the new policy to the anticipated reward of the previous policy to explain the difference between the expected rewards of the two policies [55]. Therefore, the corresponding formula uses the new policy's modified action probability but the old policy's advantage function instead of having to resample the data:

$$J(\theta + \Delta\theta) = J(\theta) + \mathbf{E}_{\tau \sim \pi_{\theta+\Delta\theta}}[\sum_{t=0}^{\infty} \gamma^t \mathcal{A}_{\pi_\theta}(S_t, A_t)] \tag{52}$$

Let ρ_π be the (unnormalized) discounted visitation frequencies [55]

$$\rho_\pi(s) = P(S_0 = s) + \gamma P(S_1 = s) + \cdots \tag{53}$$

where $S_0 \sim \rho_0$ is distribution of the initial state S_0, and the actions are chosen according to π. Therefore, we can rewrite equation (52) with a sum over states instead of timesteps:

$$\begin{aligned} J(\theta + \Delta\theta) &= J(\theta) + \sum_{t=0}^{\infty} \sum_s P(S_t = s | \pi_{\theta+\Delta\theta}) \sum_a \pi_{\theta+\Delta\theta}(a|s) \ \gamma^t \mathcal{A}_{\pi_\theta}(S_t, A_t) \\ &= J(\theta) + \sum_s \sum_{t=0}^{\infty} \ \gamma^t P(S_t = s | \pi_{\theta+\Delta\theta}) \sum_a \pi_{\theta+\Delta\theta}(a|s) \mathcal{A}_{\pi_\theta}(S_t, A_t) \\ &\overset{(53)}{=} J(\theta) + \sum_s \rho_{\pi_{\theta+\Delta\theta}}(s) \sum_a \pi_{\theta+\Delta\theta}(a|s) \mathcal{A}_{\pi_\theta}(S_t, A_t) \end{aligned} \tag{54}$$

The use of $\rho_{\pi_{\theta+\Delta\theta}}(s)$ is problematic because it makes it difficult to determine the state distribution associated with the new policy without actually sampling it. To tackle this issue, the state distribution that would be obtained if the current policy were still being used is employed. However, this approach introduces an approximation error

$$J(\theta + \Delta\theta) \approx J(\theta) + \sum_s \rho_{\pi_\theta}(s) \sum_a \pi_{\theta+\Delta\theta}(a|s) \mathcal{A}_{\pi_\theta}(S_t, A_t) \tag{55}$$

To prepare for simulation, the expectation of ρ will be utilized instead, and it can be calculated through Monte Carlo simulation.

$$J(\theta + \Delta\theta) \approx J(\theta) + \mathbf{E}_{s \sim \rho_{\pi_\theta}}[\sum_a \pi_{\theta+\Delta\theta}(a|s) \mathcal{A}_{\pi_\theta}(S_t, A_t)] \tag{56}$$

To consider the probabilities of the updated policy when using the expected action of the current policy, importance sampling will be employed [65]. This involves reusing trajectories collected with previous policies in newer updated policies, instead of summing over actions.

$$J(\theta + \Delta\theta) \approx J(\theta) + \mathbf{E}_{s,a \sim \rho_{\pi_\theta}}[\frac{\pi_{\theta+\Delta\theta}(a|s)}{\pi_\theta(a|s)} \mathcal{A}_{\pi_\theta}(s, a)] \tag{57}$$

Therefore, the updated policy's relative effectiveness compared to the previous policy can be represented by the expected advantage, also known as the surrogate advantage

$$\mathcal{L}_{\pi_\theta}(\pi_{\theta+\Delta\theta}) = \mathbf{E}_{s \sim \rho_{\pi_\theta}}[\frac{\pi_{\theta+\Delta\theta}(a|s)}{\pi_\theta(a|s)} \mathcal{A}_{\pi_\theta}(s, a)] \approx J(\theta + \Delta\theta) - J(\theta) \tag{58}$$

Currently, there is an issue where accuracy decreases when the new policy deviates too far from the previous one. To address this, it is necessary to maintain a close similarity between the new and old policies

$$\begin{gathered}\max_{\theta} \mathcal{L}_{\pi_\theta}(\pi_{\theta+\Delta\theta}) = \mathbf{E}_{s\sim\rho_{\pi_\theta}}[\frac{\pi_{\theta+\Delta\theta}(a|s)}{\pi_\theta(a|s)}\mathcal{A}_{\pi_\theta}(s,a)] \\ Subject\ to : \mathbf{E}_{s\sim\rho_{\pi_\theta}}[KL(\pi_\theta||\pi_{\theta+\Delta\theta})] \leq \delta\end{gathered} \tag{59}$$

where $KL(\pi_\theta||\pi_{\theta+\Delta\theta})$ is the KL divergence [64]. Additionally, the trust region size, denoted as δ, is a hyperparameter that determines the maximum allowable KL divergence between the new policy and the old policy. Furthermore, the KL divergence can be introduced as a penalty to the surrogate advantage function $\mathcal{L}_{\pi\theta}(\pi_{\theta+\Delta\theta})$, which is then multiplied by a factor β. Therefore, if there is a significant disparity between the old and new policies, the objective function suffers severely.

$$\max_{\theta} \mathbf{E}_{s\sim\rho_{\pi_\theta}}[\frac{\pi_{\theta+\Delta\theta}(a|s)}{\pi_\theta(a|s)}\mathcal{A}_{\pi_\theta}(s,a) - \beta\ KL(\pi_\theta||\pi_{\theta+\Delta\theta})] \tag{60}$$

where β determines the severity of the punishment. If the new policy differs from the previous policy, it penalizes the goal.

The RL community has adopted TRPO widely because it solves a variety of issues related to natural policy gradients. However, finding an optimal constant β that performs consistently well throughout the entire training process has been challenging. There are still several other issues, particularly: Large parameter matrices such as deep neural networks are difficult for TRPO to manage. TRPO is less scalable to parallel environments than other policy gradient methods such as PPO [3] and A2C [66]. Since TRPO is implemented practically using constraints, computing the Fisher matrix is necessary, which considerably slows down the updating process. Additionally, we are unable to use (first-order) stochastic gradient optimizers like ADAM. TRPO is a challenging technology to understand, develop, and troubleshoot. It might be challenging to choose the best course of action when training does not produce the anticipated results. In conclusion, it is not the most user-friendly RL algorithm.

PPO is a class of first-order methods that aims to enhance the policy learning process, and it is regarded as an advancement over other on-policy algorithms, including TRPO. Nevertheless, both TRPO and PPO face a common challenge regarding how to optimize a policy with the available data without causing performance collapse unintentionally. While TRPO utilizes a complex second-order method to address this challenge, PPO uses a set of additional strategies to ensure that new policies remain close to the old ones.

PPO approaches appear to perform at least as well as TRPO experimentally, but they are substantially easier to implement. PPO is more sample efficient and stable, and it can handle high-dimensional action spaces more effectively. It is also more computationally efficient, as it uses a trust region optimization method to constrain the update of the policy, rather than requiring a second optimization step like TRPO. PPO is also more parallelizable and can be used with more complex models such as deep neural networks and it's considered more popular in recent years.

3.4.5 PPO variants

PPO comes in two main varieties: PPO-Penalty and PPO-Clip. However, the focus of this work is PPO-Clip as it has gained popularity and widespread adoption due to its simplicity and effectiveness in practice. It also is the primary variant used at OpenAI.

PPO-Penalty

Similar to TRPO (section 3.4.4), PPO-Penalty approximates a KL-constrained update by penalizing the KL-divergence in the objective function without making it a hard constraint. It involves adding a penalty term to the objective function to encourage the new policy to be close to the old policy. The objective function for PPO-penalty is defined as:

$$\mathbf{E}_{\tau\sim\pi_\theta}[\frac{\pi_{\theta+\Delta\theta}(a|s)}{\pi_\theta(a|s)}\mathcal{A}_{\pi_\theta}(s,a) - \beta_k KL(\pi_\theta||\pi_{\theta+\Delta\theta})] \tag{61}$$

where the hyperparameter β_k controls how much the training process punishes the new policy if it is too different from the old policy. This penalty is automatically adjusted during training. If the difference between the old and new policies is too big, β_k is reduced. If the difference is small, the training area expands.

PPO-Clip

The objective of PPO-Clip does not contain a KL-divergence term, and there is no constraint at all. It relies on specific clipping in the goal function to eliminate incentives for the new policy to diverge much from the previous one. This is done by limiting the maximum change in the probability of selecting an action under the new policy compared to the old policy. The objective function for PPO-Clip is defined as:

$$\mathcal{L}^{CLIP}_{\pi_\theta}(\theta) = \mathbf{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\min\left(r_t(\theta)\mathcal{A}_{\pi_\theta}(S_t,A_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\mathcal{A}_{\pi_\theta}(S_t,A_t)\right)\right] \tag{62}$$

where $r_t(\theta) = \frac{\pi_{\theta+\Delta\theta}(A_t|S_t)}{\pi_\theta(A_t|S_t)}$ and ϵ is a hyperparameter that determines the maximum allowable change in the probability. The objective function indicates that the advantage function will be clipped if the ratio of the likelihood of the new policy to the old policy is not within the range of $1-\epsilon$ to $1+\epsilon$. Essentially, this prevents significant changes in the policy if they fall outside the acceptable range.

Finally, the smallest value is selected from the clipped and unclipped objectives, resulting in a final objective that is a conservative estimate of the unclipped objective. This approach is used to ensure stability during training by limiting the change in policy update. If the change in the probability ratio between the old and new policy is beneficial to the objective, it is included in the update, but if it is detrimental, it is excluded.

3.4.6 PPO-Clip

The PPO-Clip algorithm uses two neural networks: the actor network, which is used to select actions, and the critic network, which is used to estimate the value of the current state. The agent interacts with the environment, and the experiences (observations, actions, rewards, etc.) are collected and used to update the parameters of the actor and critic networks.

3.4.6.1 Actor network

The actor network learns a mapping $\pi_\theta(a|s)$ from observed states s to corresponding actions a, by adjusting its parameters θ, based on the experiences of the agent in the environment. The ultimate goal of the actor network is to learn a policy that maximizes the expected cumulative reward $J(\theta)$ for the agent:

$$J(\theta) = \mathbf{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{63}$$

where τ is a trajectory of states and actions, and $R(\tau)$ is the cumulative reward obtained along the trajectory.

In PPO-Clip, the actor network's parameters are updated to improve the policy by making it more likely to take actions that lead to higher rewards. This is achieved by optimizing a surrogate objective function $\mathcal{L}^{CLIP}_{\pi_\theta}(\theta)$, which is a clipped version of the ratio between the new and old policies:

$$\mathcal{L}^{CLIP}_{\pi_\theta}(\theta) = \mathbf{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \min\Big(r_t(\theta)\mathcal{A}_{\pi_\theta}(S_t, A_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\mathcal{A}_{\pi_\theta}(S_t, A_t)\Big)\right] \tag{64}$$

where $r_t(\theta) = \frac{\pi_{\theta+\Delta\theta}(A_t|S_t)}{\pi_\theta(A_t|S_t)}$ is the importance sampling ratio between the new policy and the old policy, $\mathcal{A}_{\pi_\theta}(S_t, A_t)$ is the estimated advantage of taking action A_t in state S_s, and $\text{clip}(x, a, b)$ is a function that clips x between a and b. The clipped surrogate objective function encourages the actor network to take actions that have high advantage, while ensuring that the policy update is not too large.

Additionally, to incorporate the entropy of the policy distribution into the PPO algorithm, an entropy bonus is added to the objective function. The entropy coefficient c controls the weight of the entropy bonus and is used to balance the trade-off between exploration and exploitation. The modified objective function can be written as:

$$\begin{aligned}\mathcal{L}^{CLIP}_{\pi_\theta}(\theta) = \mathbf{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T-1} \min\Big(r_t(\theta)\mathcal{A}_{\pi_\theta}(S_t, A_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\mathcal{A}_{\pi_\theta}(S_t, A_t)\Big) \\ -c\mathcal{L}^{ENTROPY}_{\pi_\theta}(\theta)\Big]\end{aligned} \tag{65}$$

where

$$\mathcal{L}^{ENTROPY}_{\pi_\theta}(\theta) = \sum_a \pi_\theta(a|S_t) \times \log \pi_\theta(a|S_t) \tag{66}$$

is the entropy term that measures the level of uncertainty or randomness in the policy distribution over actions. The entropy coefficient c is a hyperparameter that determines the weight of the entropy term in the objective function. A higher value of c encourages more exploration and stochasticity in the policy, while a lower value of c encourages the policy to exploit the current best actions. The choice of the optimal value of c depends on the specific task and environment, and is often determined by trial and error.

During training, the actor network interacts with the environment by taking actions according to the current policy and receiving rewards. The agent's experiences are stored in a replay buffer, and the actor network is trained using batches of data sampled from the buffer. The parameters θ of the actor network are updated by minimizing the surrogate objective function over multiple epochs of minibatch updates. This iterative process of updating the actor network's parameters leads to the convergence of the learned policy towards an optimal policy that maximizes the expected cumulative reward.

An example of an actor network in PPO would be a neural network that takes in the current state of a robotic arm and outputs a probability distribution over the possible actions (e.g. move left, move right, stay still). The actor network would be trained to choose actions that are likely to lead to higher rewards, such as successfully grasping an object or avoiding obstacles. Another example of an actor network in PPO would be a neural network that takes in the current state of a video game and outputs a probability distribution over the possible actions (e.g. move left, move right, jump). The actor network would be trained to choose actions that are likely to lead to higher rewards, such as collecting coins or defeating enemies.

3.4.6.2 Critic network

The critic network, or the value network, estimates the current state's value and predicts the agent's expected future cumulative reward. Its aim is to make accurate predictions by adjusting its parameters based on the agent's experiences. In PPO-Clip, the critic network's parameters are optimized to improve the value function. The value function is a mathematical function that predicts the expected cumulative reward an agent can receive by following a policy in a specific environment.

More specifically, the critic network's parameters ϕ, are updated using a variant of stochastic gradient descent called the mean squared error loss. This loss function measures the difference between the predicted value function and the true value function, which is calculated using the discounted future rewards obtained by the agent. The aim of this update is to minimize the mean squared error loss and make the predicted value function more accurate in estimating the expected cumulative reward.

$$\mathcal{L}^{VALUE} = \mathbf{E}_{\tau \sim \pi_\theta}\left[(V_\phi(s) - V(s))^2\right] \tag{67}$$

where $V_\phi(s)$ is the predicted value function for state s using the critic network parameters ϕ, and $V(s)$ is the true value function, calculated using the discounted future rewards obtained by the agent while following policy π_θ.

The optimization of the critic network's parameters in PPO-Clip is crucial as it helps the agent to learn a better value function, which in turn can guide it to select better actions and improve its overall performance in the environment. This optimization is performed by minimizing the mean squared error loss function between the predicted value function and the true value function. The true value function is calculated using the discounted future rewards obtained by the agent. By minimizing the mean squared error loss, the critic network's parameters are adjusted to improve its accuracy in estimating the expected cumulative reward.

An example of a critic network in PPO would be a neural network that takes in the current state of a robotic arm and outputs a single value representing the estimated future cumulative reward for the agent if it continues to follow the current policy. The critic network would be trained to predict the expected cumulative reward based on the current state of the robotic arm and the actions taken by the agent. Another example of a critic network in PPO would be a neural network that takes in the current state of a video game and outputs a single value representing the estimated future cumulative reward for the agent if it continues to follow the current policy. The critic network would be trained to predict the expected cumulative reward based on the current state of the video game and the actions taken by the agent.

In PPO-Clip, the actor network and the critic network are trained together. The critic network is used to compute the advantage function, which is used to update the actor network's parameters in a way that maximizes the expected cumulative reward. In this way, the actor network is updated to improve the policy by making it more probable to take actions that lead to higher rewards, while the critic network is updated to improve the value function by making it more accurate in estimating the expected cumulative reward.

The update process in PPO-Clip involves comparing the "old" policy, represented by the actor network's current parameters, with the "new" policy, represented by the actor network's updated parameters. The ratio of the probability of the new policy taking a certain action over the old policy taking the same action is computed and if this ratio is greater than 1, it is clipped to be $1 + \epsilon$. This is done to prevent the agent from making large, sudden changes in the policy that can destabilize the training process [3]. This algorithm uses this "clipped" ratio to update the parameters of the actor network, along with the critic network, which helps to improve the agent's ability to estimate the value of the current state. The algorithm continues to repeat this process until the agent's performance in the environment reaches an acceptable level.

Algorithm 4 summarizes the PPO-Clip steps that are:

1. The old policy ($\pi_{\theta_k}(a|s)$), represented by the current parameters of the actor network, is used to select actions for the agent.

2. The critic network is used to estimate the value of the current state. Let's denote this value as $V_{\phi_k}(S_t)$ where S_t is the current state.

3. The estimated advantage function that is denoted as $\hat{\mathcal{A}}_t(S_t, A_t)$ where S_t is the current state, A_t is the action taken by the agent, can be defined as:

Algorithm 4 PPO-Clip

Input: number of epochs N; initial policy parameters θ_0; initial value function parameters, ϕ_0; clip rate ϵ; entropy coefficient c.
Output: final policy π_{θ_N}.

```
 1: for $k \leftarrow 1$ to $N$ do
        Collect set of trajectories $D_k = \{\tau_i\}$ using the current policy $\pi_{\theta_k}$
 2:     for $t \leftarrow 1$ to $T$ do
 3:         Observe state $S_t$.
 4:         Sample action $A_t \sim \pi_{\theta_k}(A_t|S_t)$.
 5:         Execute action $A_t$, observe reward $R_t$ and next state $S_{t+1}$.
 6:         Store $(S_t, A_t, R_t, S_{t+1})$ in the trajectory buffer.
 7:     end for
 8:     Compute advantage estimates $\hat{\mathcal{A}}_1(S_1, A_1), \cdots, \hat{\mathcal{A}}_T(S_T, A_T)$ based on the current value function.
        $\hat{\mathcal{A}}_t(S_t, A_t) = -V_{\phi_k}(S_t) + R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t+1} R_{T-1} + \gamma^{T-t} V_{\phi_k}(S_t)$
 9:     Update the policy by maximizing the PPO-Clip objective:
        $\theta_{k+1} = \max_\theta \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} min\left(r_t(\theta_k)\hat{\mathcal{A}}_t(S_t, A_t), clip(r_t(\theta_k), 1-\epsilon, 1+\epsilon)\hat{\mathcal{A}}_t(S_t, A_t)\right) - c\, \mathcal{L}_{\pi_\theta}^{ENTROPY}(\theta)$
    typically via stochastic gradient ascent with Adam,
        where the probability ratio is $r_t(\theta_k) = \frac{\pi_{\theta_k+1}(A_t|S_t)}{\pi_{\theta_k}(A_t|S_t)}$,
        and $\mathcal{L}_{\pi_\theta}^{ENTROPY}(\theta) = \sum_a \pi_\theta(a|S_t) \times \log \pi_\theta(a|S_t)$
10:     Update value function by regression on mean-squared error:
        $\phi_{k+1} = argmin_\phi \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left(V_{\phi_k}(S_t) - \hat{R}_t\right)^2$
    typically via some gradient descent algorithm,
        where $\hat{R}_t = R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t} R_T$
11: end for
12: return $\pi_{\theta_N}$
```

$$\hat{\mathcal{A}}_t(S_t, A_t) = -V_{\phi_k}(S_t) + R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t+1} R_{T-1} + \gamma^{T-t} V_{\phi_k}(S_t)$$

4. The new policy, represented by the updated parameters of the actor network, is used to select actions for the agent. The new policy is denoted as $\pi_{\theta_{k+1}}(A_t|S_t)$ where S_t is the current state, A_t is the current action, and θ_{k+1} are the updated parameters of the actor network.

5. The ratio of the probability of the new policy taking a certain action over the old policy taking the same action is computed. This ratio is denoted as $r(\theta)$ and can be defined as: $r(\theta) = \frac{\pi_{\theta_{k+1}}(a|s)}{\pi_{\theta_k}(a|s)}$

6. The ratio $r(\theta)$ is used to update the parameters of the actor network, by maximizing the following objective function:

$$\mathcal{L}_{\pi_\theta}^{CLIP}(\theta) = min(r(\theta) * \hat{\mathcal{A}}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon) * \hat{\mathcal{A}}(s,a)) - c\,\mathcal{L}_{\pi_\theta}^{ENTROPY}(\theta) \tag{68}$$

where $clip(x, a, b)$ is a function that clips the value of x between a and b, and ϵ is a small positive value that controls the maximum change in the policy (as presented in Figure 8).

7. The parameters of the critic network are updated to minimize the mean squared error between the estimated value and the actual cumulative reward:

$$\mathcal{L}^{VALUE} = \left(V_\phi(S_t) - \hat{R}_t\right)^2 \tag{69}$$

where $\hat{R}_t = R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t} R_T$

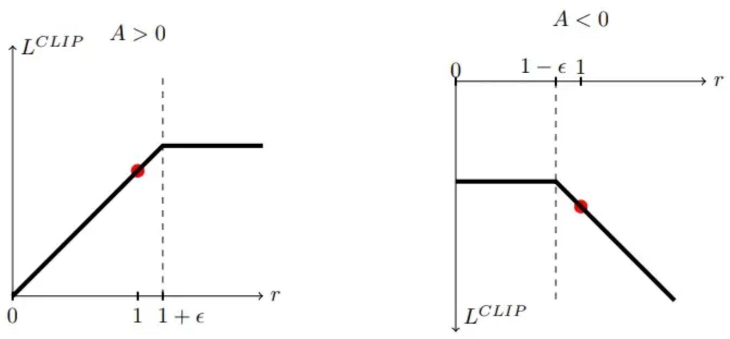


Figure 8: Plots displaying the $\mathcal{L}^{CLIP}$ surrogate function for a single timestep as a function of the probability ratio $r(\theta)$, with positive advantages on the left and negative advantages on the right. The red circle on each graph indicates the initial point for optimization [3].

Overall PPO-Clip is a powerful algorithm that can be used to train agents to perform a wide range of tasks, and it has been widely used in various applications such as robotics, video games, and autonomous vehicles.

4 Numerical experiments

This section of the thesis presents the results achieved through the application of the proposed RL framework in the design of induction machines. It demonstrates the improved computational performance of the designs compared to traditional design methods. Furthermore, the section offers additional evaluations and analyses to enhance the understanding of the framework and its potential for future enhancements. This section includes two key sub-sections: Results and Further Evaluations.

In the Results sub-section (section 4.2) showcases the outcomes obtained from applying the RL framework to the design of induction machines. It offers a detailed analysis and examination of the obtained outcomes, aiming to provide a comprehensive understanding of the framework's effectiveness in the specific domain of induction machine design. The results are presented in a clear and structured manner, allowing readers to assess the impact and implications of applying the RL framework. The section highlights the improvements achieved compared to baseline designs and demonstrates the effectiveness of the RL approach in enhancing the computational performance in the designing the machines.

The Further Evaluations sub-section (section 4.3) delves deeper into assessing the proposed framework. It discusses additional evaluations and analyses conducted to gain a deeper understanding of the system's behavior and the impact of various factors and model's hyperparameters on the design process. These evaluations include sensitivity analysis and robustness testing. The section provides insights into the strengths and limitations of the framework and suggests potential areas for further improvement.

4.1 Introduction

The numerical experiments section of this thesis is dedicated to the application of RL in the design of induction machines. The primary objective is to optimize the design parameters of these machines in order to minimize their cost. In the upcoming paragraphs, we will provide a detailed description of the step-by-step procedures employed in conducting these experiments. This will involve providing detailed explanations of the various stages encompassing the experimental process. Additionally, we will outline the execution of the experiments themselves, which involves running simulations or algorithms, capturing experimental data, and monitoring and recording observations. By presenting these details, this section aims to provide a comprehensive overview of the numerical experiments conducted and the strategies employed to achieve the desired design outcomes for induction machines using RL.

The game's design process follows the principles of a Markov Decision Process (MDP) (2). This means that the game's dynamics adhere to the MDP framework, where the state, action, and reward transitions are governed by probabilistic rules. In the context of the game, the MDP formulation provides a mathematical framework to model the decision-making process. In the game, the objective is to find an optimal design solution within a given set of constraints. The initial design is calculated based on factors such as power, voltage, and machine type. As stated in section 3.2.2, in each step, the observation state consists of both the design flags and the previous action that was taken. However, it is possible to include additional variables such as performance values into the observation space for certain training scenarios.

During each step, one of six available actions is chosen to optimize the design, including increasing/decreasing length, increasing/decreasing number of turns, and increasing/decreasing tooth tip height of the machine. However, it is possible to execute multiple actions within each step t. For instance, during timestep t, instead of increasing the height of the tooth tip only once, we can increase it multiple times, such as twice or even more. This approach has the potential to enhance the computational efficiency of the process, resulting in improved overall performance. By incorporating multiple actions within a single timestep, it enables accelerated progress towards the desired outcome.

The game concludes after 300 steps if the agent fails to uncover the solution, resulting in the agent's loss. Conversely, if the agent successfully identifies the solution within 300 steps, it will emerge as the winner of the game. There are two possible definitions for the ending criteria: The first and simpler way is to stop the game when the agent discovers a state in which all the performance flags have a value of zero. This means that the agent has found a solution that satisfies both the user's requirements and the manufacturing constraints. This is referred to as "simple ending criteria". The second way to end the game is to stop when the agent identifies the state that has the optimal mass among all the states in which the performance flags are set to zero. This is known as "optimal ending criteria". This solution is more difficult to find because it requires the agent to not only meets the requirements and constraints but also achieves the best possible mass for the machine.

To evaluate the effectiveness of the proposed solutions, a series of experiments were conducted on five distinct electrical seed machines (refer to Table 2). In order to expand the training dataset beyond these five seed machines, we introduce power and voltage multipliers $\in [0.5, 1.5]$. These multipliers are utilized to generate different variations of the game by multiplying the power and voltage requirements of the seed machine. Essentially, each distinct combination of these multipliers modifies the objective of the game. For instance, let's consider seed machine number 1, which has a power requirement of 360 kW. If we apply a power multiplier of 0.9, the resulting machine would have a power requirement of $360 \times 0.9 = 324$ kW. This new machine will exhibit entirely different performance values and corresponding flag values. Consequently, the optimization actions for this machine will differ from those applied to the original seed machine. Moreover, to further enhance the difficulty of the training process, there is an option to choose different target I_s/I_n values for each seed machine. Similar to power and voltage multipliers, the introduction of the target I_s/I_n also alters the objective of the game.

By introducing different power and voltage multipliers and target I_s/I_n values, we can create a diverse set of scenarios that encompass a wider range of conditions and challenges. This approach allows us to explore various configurations and assess the performance and adaptability of the system under different circumstances. It helps ensure that the trained model can effectively handle a broader spectrum of real-world scenarios and exhibit robustness in its decision-making process.

Table 2: Properties of five different electrical seed machines.

Machine	Shaft Height [mm]	Number of Poles	Nominal Power [kW]	Nominal Voltage [kV]
0	560	6	1400	10
1	400	2	360	6
2	630	12	1450	6
3	500	4	750	0.69
4	1000	4	15000	10

This section describes the outcomes of the research study that evaluated the effectiveness of a RL-based solution for optimizing the performance of seed machines. The research is conducted in a systematic manner, gradually increasing the number of machines and experimental variables to gain a comprehensive understanding of the effectiveness of the solution. The initial phase of training involved a single seed machine with fixed values for the power multiplier, voltage multiplier, and target I_s/I_n. Subsequently, the training process gradually progressed by introducing additional machines and experimental variables. In the second stage, the training expanded to include one machine with a fixed target I_s/I_n but varying power and voltage multipliers, resulting in 30 distinct cases. Following that, three machines with fixed target I_s/I_n but different power and voltage multipliers were incorporated, resulting in 122 total cases. Thereafter, experiments were conducted to evaluate the RL performance by varying the the target I_s/I_n values, power multipliers, and voltage multipliers, resulting in a total of 375 cases. In the final stage of the study, five machines with different target I_s/I_n values, power multipliers, and voltage multipliers were included, resulting in a comprehensive set of 650 total cases.

This extensive range of machines and experimental variations allowed for a thorough exploration of the RL's behavior and performance under diverse conditions, providing valuable insights into the effectiveness and adaptability of the solution. It is important to note that all of these cases were utilized for both the training and testing phases. In essence, at each stage, the agent was trained using the available cases specific to that stage, and subsequently evaluated for its performance. During the training process, the agent underwent iterative training using the cases relevant to each stage. It learned from the provided data, including the various machine configurations and corresponding performance metrics. The training aimed to enhance the agent's decision-making capabilities, enabling it to adapt and optimize its actions based on the given scenarios. Following the training phase, the agent's performance was assessed during the testing phase. The agent was evaluated on its ability to make informed decisions and achieve desirable outcomes in the training cases. This testing phase served as a validation of the agent's learned knowledge.

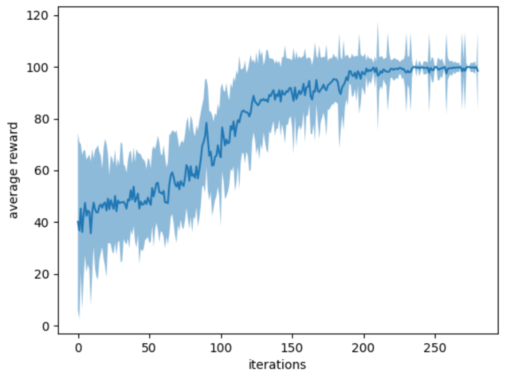
Furthermore, the evaluation of the performance of the RL agent involves a comparison with a deterministic approach known as the "reference." The reference approach represents a scenario where an engineer designs the corresponding electrical machine using their expertise and domain knowledge. The purpose of using a reference is to provide a standard or baseline for assessing the effectiveness and efficiency of the RL agent's actions and decisions. The reference typically represents a well-established approach in the field of designing electrical machines. The reference is crucial for gaining insights into the capabilities and limitations of the RL agent. Moreover, it facilitates the identification of areas where the RL agent may excel or struggle in comparison to existing methods. The results indicate that the RL-based solution can attain a level of performance comparable to, or even surpassing, the traditional design approach. Furthermore, the RL-based solution exhibits superior computational efficiency, requiring less time for the design process.

4.2 Results

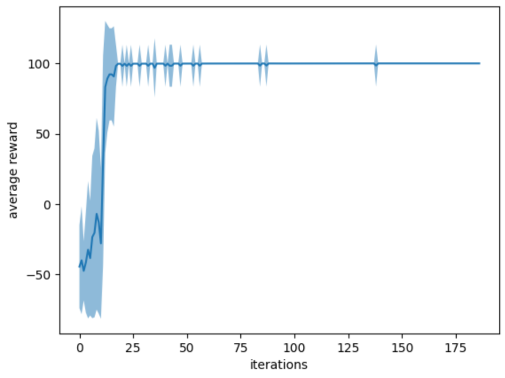
In the first stage of the study, the aim was to establish a benchmark performance for the machine and assess the effectiveness of the RL solution under controlled conditions by testing it on a single seed machine, with a constant power multiplier, voltage multiplier, and target I_s/I_n. At this stage, our focus is on the performance flags and the aim to identify a machine that has all the flags set to zero. This is the simplest reward function that is described in section 3.3.1 with the simple ending criteria. Following 700,000 number of training steps (Figure 9a), the agent was able to identify a solution using 17 steps, compared to the 18 reference steps. However, the solution was not optimal. As a result, another agent was trained using the same reward function, but with an optimal ending criteria. Subsequently, after 900,000 number of training steps (Figure 9b), the agent was able to identify the optimal solution in 21 steps, compared to the 18 reference steps (Figure 10). Table 3 summarizes the results for this stage.

Table 3: Simple reward function applied to analyze one case (one seed machine with fixed power multiplier 1.0, voltage multiplier 1.0, and target I_s/I_n 5.0). PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Total cases
Simple	1	700K	94%	0%	1
Optimal	1	900K	117%	100%	1



(a) Simple ending criteria.



(b) Optimal ending criteria.

Figure 9: Reward convergence for the first stage.

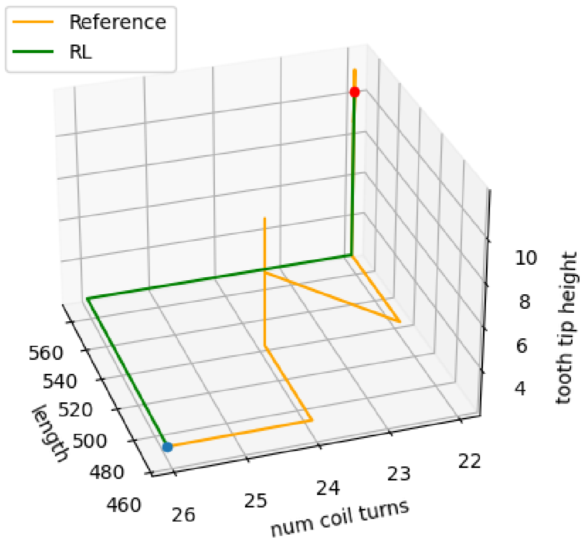


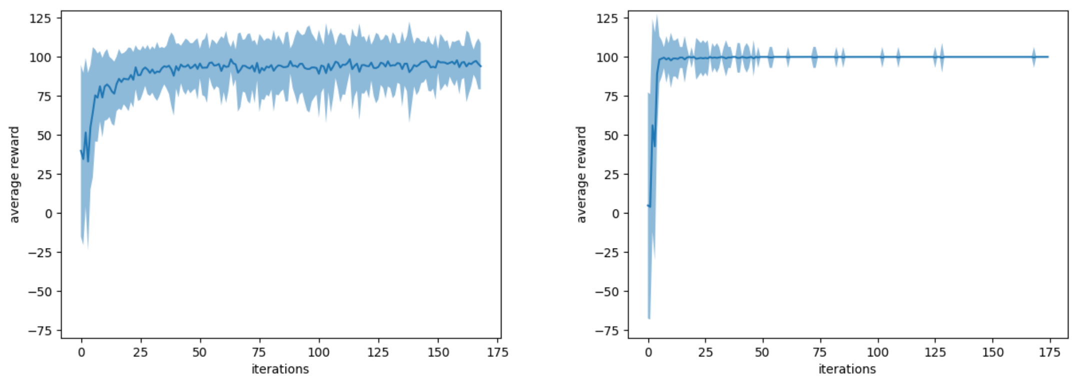
Figure 10: The sample case for the simple reward function to achieve optimal goal as ending criteria for one case. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

In the second stage, in order to scale our cases, the experiments are extended to include one seed machine, with a fixed target I_s/I_n but varying power multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1, 1.15]$ and voltages multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1]$ along with the simple reward function (as presented in section 3.3.1) and the simple ending criteria. The solution's ability to adapt to different operating conditions was tested in 30 different cases of induction machines. After undergoing 1,000,000 number of training steps (Figure 11a), the agent was able to find a solution for all cases using

430 total steps (14.3 average steps), which is fewer than the 665 total reference steps (22.2 average steps). However, it only managed to find the optimal solution for 26 cases. Another agent was trained using the same reward function but with the optimal ending criteria, and after the same number of training steps (Figure 11b), it found the optimal solution for all 30 cases using a total of 478 steps (15.9 average steps), which is also fewer than the 665 total reference steps (22.2 average steps). Figure 12 displays a sample scenario (involving seed machine number 1 with power multiplier 1.15, voltage multiplier 1.0, and the target I_s/I_n 5.5) that represents the training with the optimal ending criteria. In this case, the PPO steps are 38, while the reference steps are 81. According to Table 4, it can be inferred that while the simple ending criteria finds the solution for all the cases with fewer total steps required, it cannot find the optimal solution for all of them.

Table 4: Simple reward function applied to analyze one machine with varying power multipliers and voltage multipliers but fixed target I_s/I_n. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Total cases
Simple	1	1M	65%	87%	30
Optimal	1	1M	72%	100%	30



(a) Simple ending criteria. (b) Optimal ending criteria.

Figure 11: Reward convergence for the second stage.

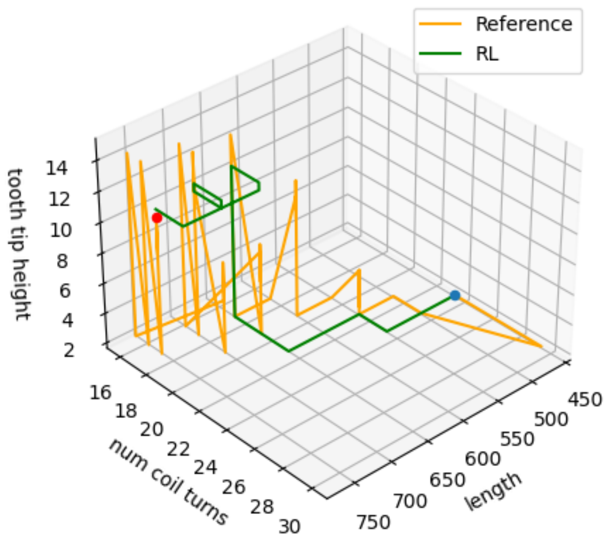
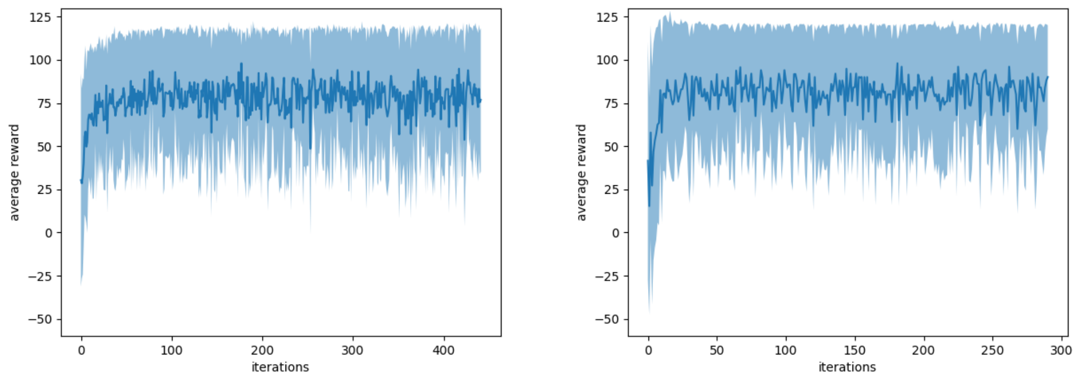


Figure 12: A sample case for the simple reward function to achieve optimal goal as ending criteria for one machine with varying power and voltage multipliers. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

In the third stage, the scale was increased, and the solutions on three different seed machines, each with a fixed target I_s/I_n but different power multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1, 1.15]$ and voltages multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1]$ were tested. The solution's ability to optimize performance across different machines and operating conditions was evaluated in 122 different cases. The same as the previous stages, first the model was trained with the simple reward function (section 3.3.1) in conjunction with the simple ending criteria. After undergoing 2,500,000 number of training steps (Figure 13a), the agent was able to find a solution for all cases using 884 total steps (7.2 average steps), which is fewer than the 1350 total reference steps (11.8 average steps). However, it only managed to find the optimal solution for 114 cases. Another agent was trained using the same reward function but with the optimal ending criteria, and after 2,000,000 number of training steps (Figure 13b), the optimal solution was discovered for all 122 cases, requiring only 934 total steps in total (7.7 average steps). This is significantly less than the 1350 total reference steps (11.8 average steps), as presented in Table 5. Figure 14 represents a sample scenario (involving seed machine number 2 with power multiplier 1.4, voltage multiplier 1.1, and target I_s/I_n 6.0) that represents the RL steps with the optimal ending criteria compared with the reference steps. In this case, the PPO steps are 28, while the reference steps are 27. This indicates that in certain scenarios, PPO may not be more computationally efficient than the reference method, even if the total number of PPO steps is significantly lower. Consequently, it is necessary to develop a better reward function.

Table 5: Simple reward function applied to analyze three machines with varying power and voltage multipliers but fixed target I_s/I_n. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Total cases
Simple	3	2.5M	65%	93%	122
Optimal	3	2M	70%	100%	122



(a) Simple ending criteria. (b) Optimal ending criteria.

Figure 13: Reward convergence for the third stage.

Until now, our reward function has been basic and only takes into account the values of the flags. Comparing it with the reference implementation, which incorporates engineering knowledge, would not be fair. Hence, it would be advantageous to augment the reward function with some engineering knowledge. This human expert knowledge based reward function is described in section 3.3.2. Therefore, in the fourth stage, the experiments were extended to include the same three seed machines as the previous stage, each with fixed target I_s/I_n values, different power and voltage multipliers but with the human expert knowledge based reward function. The agent with the simple ending criteria underwent 2,500,000 number of training steps and was able to solve all 122 cases using 834 total steps (6.8 average steps), which is less than the 1350 reference steps (11.8 average steps). However, it only achieved the optimal solution in 102 cases. Another agent was trained with the same reward function, but with the optimal ending criteria, and after 3,000,000 number of training steps, it found the optimal solution for all 122 cases, using a total of 946 steps (7.8 average steps), as presented in Table 6. Moreover, for the sample case in the previous stage (seed machine number 2 with power multiplier 1.4, voltage multiplier 1.1, and target I_s/I_n 6.0), it managed to find the solution in 23 steps compared to 27 reference steps (Figure 16).

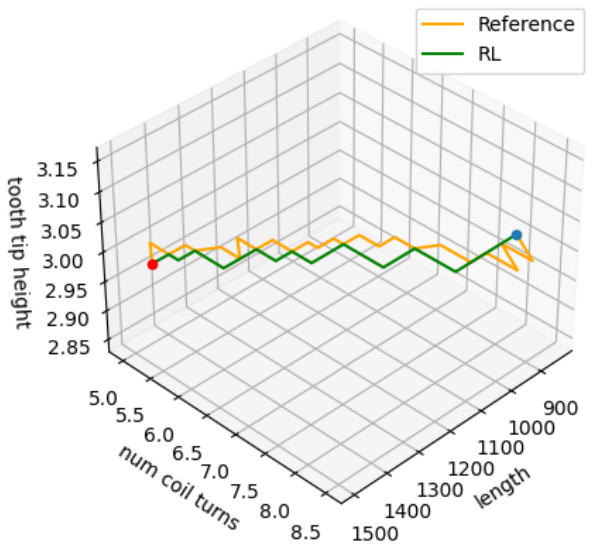
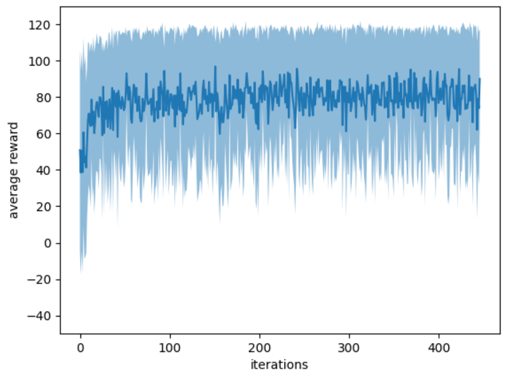


Figure 14: A sample case for the simple reward function to achieve optimal goal as ending criteria for three different machines with varying power and voltages multipliers. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

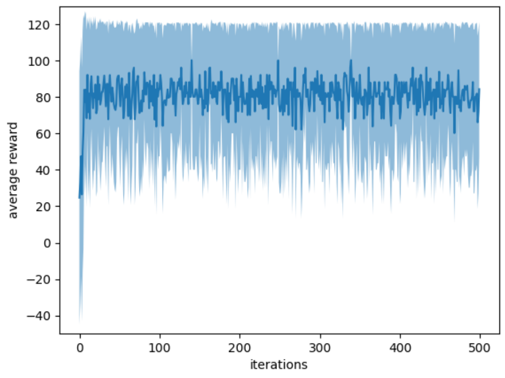
Table 6: Human expert knowledge based reward function applied to analyze three machines with varying power and voltage multipliers but fixed target I_s/I_n. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Total cases
Simple	3	2.5M	62%	89%	122
Optimal	3	3M	70%	100%	122

In the fifth stage, the experiments were extended to include the same three different seed machines, each with different target I_s/I_n values $\in [5.0, 5.5, 6.0]$, power multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1, 1.15]$, and voltages multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1]$ along with the human expert knowledge based reward function (section 3.3.2). This delves deeper into the productivity of the RL-based solution across a wider range of operating conditions and various machines. After 15,000,000 number of training steps (Figure 17a), the agent managed to solve 373 cases out of all 375 cases using 2742 total steps (7.4 average steps), which is fewer than the 4588 total steps required as a reference (12.2 average steps). Nonetheless, the optimal solution was only reached in 336 cases. In contrast, a different agent that was trained using the same reward function but with an optimal ending criteria took 40,000,000 number of training



(a) Simple ending criteria.



(b) Optimal ending criteria.

Figure 15: Reward convergence for the fourth stage.

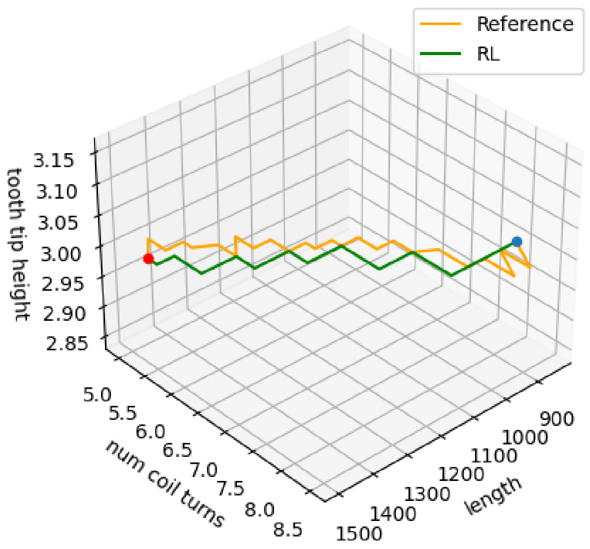
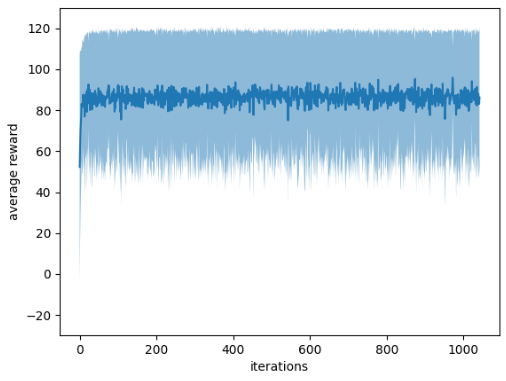


Figure 16: A sample case for the human expert knowledge based reward function to achieve optimal goal as ending criteria for three different machines with varying power and voltages multipliers. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

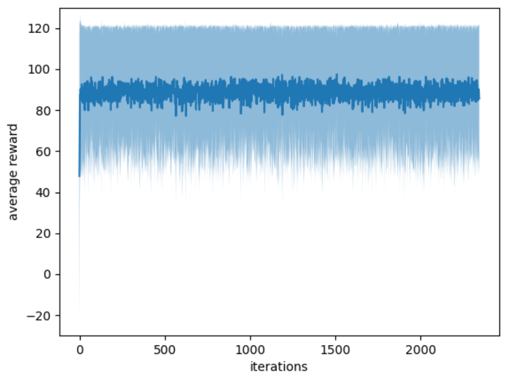
steps (Figure 17b) to achieve the optimal solution in 370 cases, using a total of 2878 steps (7.8 average steps) compared to 4414 reference steps for the successful cases (11.9 average steps), as presented in Table 7. Figure 18 shows the two failed cases in the simple ending criteria training. It is evident that the current experiment's functionality is limited in terms of scalability, and thus altering either the reward function or the observation space needs to be considered.

Table 7: Human expert knowledge based reward function applied to analyze three machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
Simple	3	15M	60%	90%	2	375
Optimal	3	40M	65%	100%	5	375



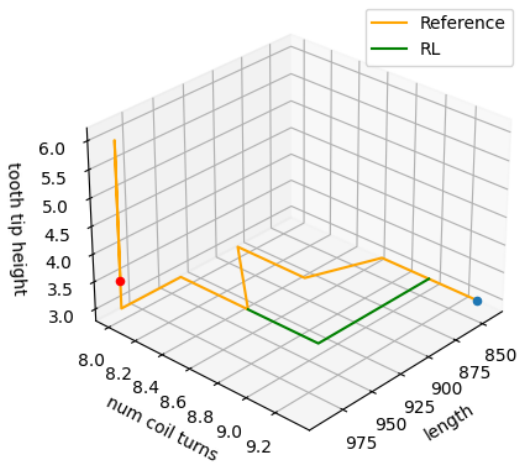
(a) Simple ending criteria.



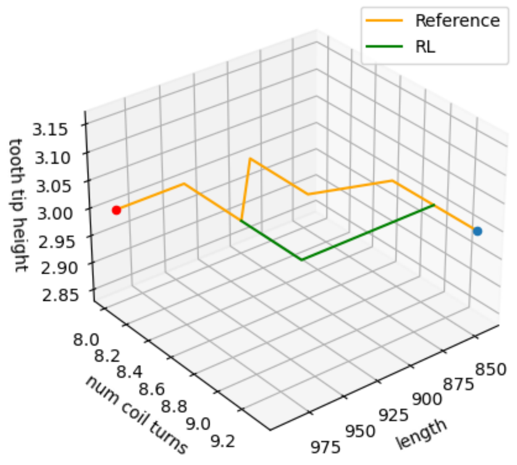
(b) Optimal ending criteria.

Figure 17: Reward convergence for the fifth stage.

During the sixth stage, without any changes to the reward function (human expert knowledge based reward function as presented in section 3.3.2) modifications were made to the observation space which previously only included the values of performance flags and the previous action taken. To enhance the investigation of the RL-based solution's effectiveness across various machines and operating conditions, additional performance values were incorporated into the observation space. These included differences in current I_s/I_n and the target I_s/I_n (current I_s/I_n - target I_s/I_n), stator temperature rise, and airgaps flux density. Following 5,000,000 number of training steps (Figure 19a), the agent successfully resolved all 375 cases, utilizing only 2778 total steps (7.4 average steps) in compared to the 4606 total reference steps (12.3 average steps). It is worth noting that the optimal solution was achieved in 365 cases. In contrast, another agent trained using the same reward function but with an optimal ending criteria required 11,500,000 number of training steps (Figure 19b) to attain the optimal solution in all 375 cases, employing a total of 2864 steps (7.6 average steps) in compare to the 4606 reference steps (12.3 average), as presented in Table 8. Figure 20 shows the two failed cases for the previous step that are successful now. The utilization of the new observation space is evident in the significant reduction of the total number of training steps (from 15,000,000 to 5,000,000). This reduction is attributed to the agent having access to more



(a) First failure.



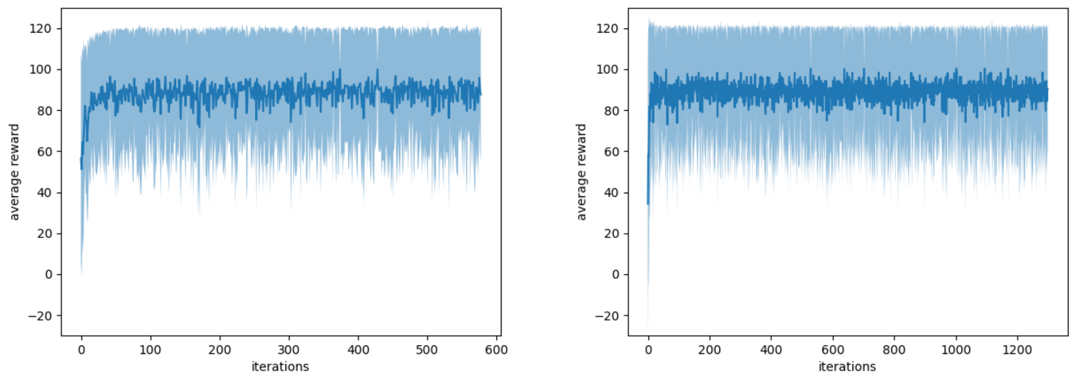
(b) Second failure.

Figure 18: Two sample failure cases for the human expert knowledge based reward function to achieve simple goal as ending criteria for three different machines with varying power multipliers, voltages multipliers, and target I_s/I_n values. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

comprehensive information during the training process.

Table 8: Human expert knowledge based reward function applied to analyze three machines with varying power multipliers, voltage multipliers, and target I_s/I_n values with performance values added to the observation space. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

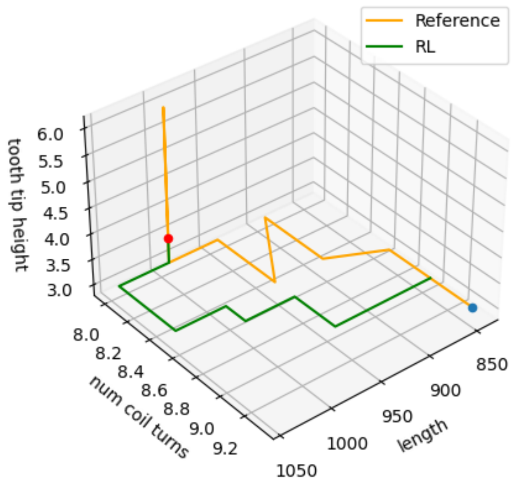
Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
Simple	3	5M	60%	97%	0	375
Optimal	3	11.5M	62%	100%	0	375



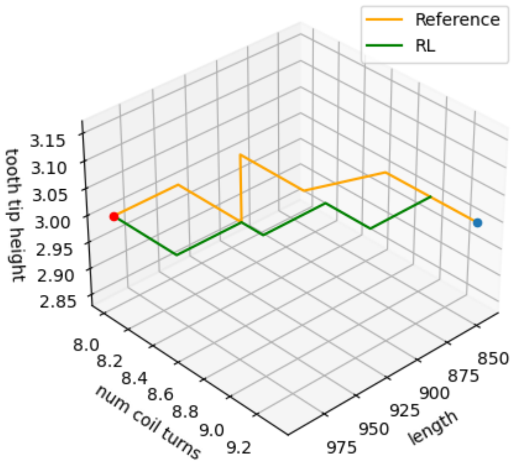
(a) Simple ending criteria. (b) Optimal ending criteria.

Figure 19: Reward convergence for the sixth stage.

In the next stage, the experimentation was expanded by introducing two more machines to the training process with the human expert knowledge based reward function (as presented in section 3.3.2). As a result, the experiment encompassed five different seed machines, each with with varying target I_s/I_n values, power multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1, 1.15]$, and voltages multipliers $\in [0.7, 0.8, 0.9, 1.0, 1.1]$. After completing 11,000,000 number of training steps (as shown in Figure 21a), the agent was able to successfully resolve all 650 cases using only 4320 total steps (6.6 average steps), which is fewer than the 6876 total reference steps (10.6 average steps). Notably, the optimal solution was achieved in 631 cases. In comparison, another agent trained with the same reward function but with an optimal ending criteria required 25,000,000 number of training steps (as illustrated in Figure 21b) to attain the optimal solution for 647 cases, and used a total of 4275 steps (6.6 average steps) instead of the 6806 total reference steps for the successful cases (10.5 average steps), as presented in Table 9. Since this model was unable to find a solution that met the optimal ending criteria for three induction machines, it is necessary to develop a more effective reward function.



(a) First failure.



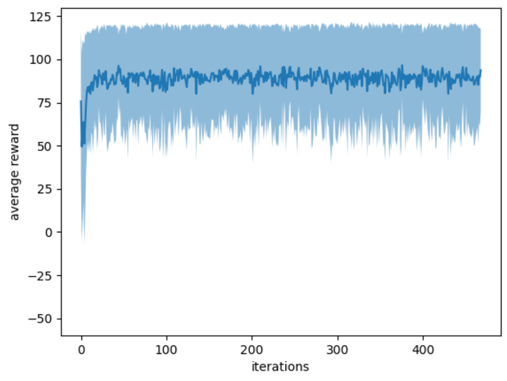
(b) Second failure.

Figure 20: Two sample failure cases for the human expert knowledge based reward function to achieve simple goal as ending criteria with performance values added to the observation space for three different machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

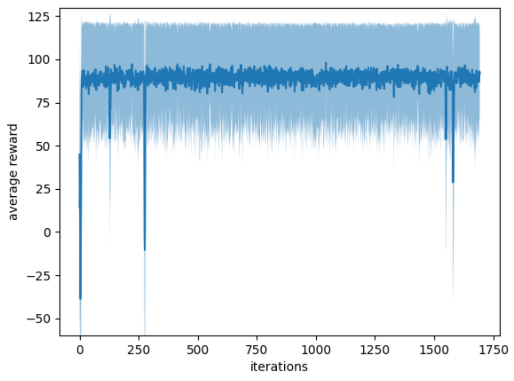
The ultimate phase of the process comprises training the innovative distance reward function (outlined in Section 3.3.3). In conjunction with the distance reward function, these five machines were employed in the study. The agent successfully solved all the 650 cases using 4008 total steps (6.2 average steps), which is fewer than

Table 9: Human expert knowledge based reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
Simple	5	11M	63%	97%	0	650
Optimal	5	25M	63%	100%	3	650



(a) Simple ending criteria.



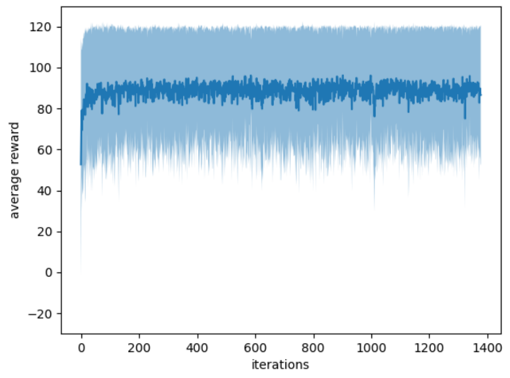
(b) Optimal ending criteria.

Figure 21: Reward convergence for the seventh stage.

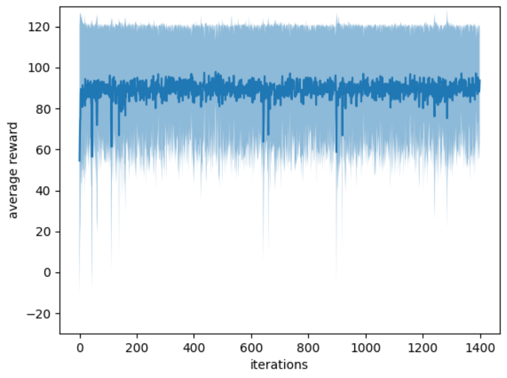
the 6876 total steps required as a reference (10.6 average steps), after undergoing 20,000,000 number of training steps (Figure 22a). However, the optimal solution was only attained in 616 cases. On the other hand, another agent that employed the same reward function but with an optimal ending criteria required 20,000,000 number of training steps (Figure 22b) to achieve the optimal solution in 648 cases. The successful cases used a total of 4227 steps (6.5 average steps), which is fewer compare to the 6815 total reference steps for the successful cases (10.5 average steps), as presented in Table 10.

Table 10: Distance reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Ending criteria	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
Simple	5	20M	58%	95%	0	650
Optimal	5	20M	62%	100%	2	650



(a) Simple ending criteria.



(b) Optimal ending criteria.

Figure 22: Reward convergence for the final stage.

The findings of the study indicate that the suggested solution can adeptly adjust to diverse design scenarios and yield appropriate designs. The solution was shown to substantially decrease the amount of computational time required for design, while still maintaining excellent design quality. Moreover, the solution can conform to various design scenarios, indicating its versatility.

4.3 Further evaluations

In this section, further improvements and comparisons are being explored after the appropriate reward function (the distance reward function) and the observation space have been discovered. Additionally, a sensitivity analysis has been carried out to investigate the impact of various hyperparameters on the performance of the RL-based solution. The results indicate that the effectiveness of the solution is relatively robust with respect to the choice of hyperparameters.

Initially, the problem involved six distinct actions, all of which had an impact on three design variables - the length of the machine, number of coil turns, and rotor tooth tip height. As it is widely recognized that the primary design variables of induction motors are the length of the machine and the number of coil turns, this experiment focused on assessing the effect of varying the rotor tooth tip height in our electrical machine design game. The comparison between these two action spaces, using the distance reward function, is made in Table 11, based on the training set that includes all instances where solutions were found for the initial tooth tip height, across all five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values with the simple ending criteria. Table 11 highlights that by removing the tooth tip height actions, even though the agent could not provide the optimal solution for many cases, it performed well overall. However, the total number of training steps almost remains constant between these two experiments

Moreover, the performance of the agent is assessed by dividing all cases into separate training and testing sets. In this approach, a random selection of 20% of the

Table 11: Distance reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. The comparison is conducted between scenarios involving three design variables and those involving two design variables. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Design variables	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
2	5	14M	48%	78%	0	590
3	5	13M	57%	97%	0	590

cases is assigned to the test set, which remains inaccessible to the agent during the training. By utilizing this approach, the performance of the agent on unseen cases can be evaluated and ensured that it can generalize well to new data. By conducting tests on the unseen test set, the performance of the agent can be estimated on new cases, allowing for the assessment of whether it overfits or underfits the training data. According to the information presented in Table 12, the model that incorporates three design variables outperforms the model that uses only two. For the three design variables, it was found that splitting the dataset into test and train sets did not result in a significant increase in the number of training steps, which only increased from 13,000,000 to 15,000,000. Additionally, the number of PPO steps and optimal cases remained almost unchanged, and there were no instances of failure observed as a result of the dataset split. However, the model with the two design variables failed to find the solution for two cases.

Figure 23 shows one of the failure case for this model. In this particular instance, the agent became trapped in a cycle of executing and reversing certain actions until it reached the maximum permitted number of iterations.

Table 12: Distance reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. This evaluation is performed in the context of a test/train split for the valid cases with 2d design variables, incorporating simple ending criteria. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Design variables	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
2	5	20M	48%	79%	2	590
3	5	15M	58%	97%	0	590

Additionally, Table 13 displays how well the distance reward function performs across all scenarios involving the three design variables with splitting the data to test/train cases. Upon examining the data, it is clear that the split cases perform similarly to the non-split case, indicating that the distance reward function has a

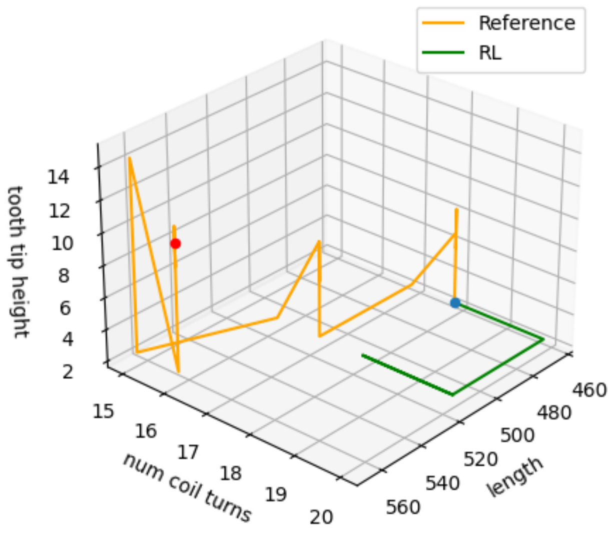


Figure 23: Failure case for distance reward function with simple ending criteria with the train/test split. The blue point is the initial design and the red point is the goal design, where the user requirements are met.

high level of generalization ability. Based on these results, it can be concluded that the split cases do not significantly impact the overall performance of the distance reward function. This suggests that the function is able to effectively handle the complexities of the three design variables, and can be reliably used in a variety of scenarios.

Table 13: Distance reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values. This evaluation is performed in the context of a test/train split, incorporating simple ending criteria. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Test splitted	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
No	5	20M	58%	94%	0	650
Yes	5	20M	59%	93%	0	650

Additionally, a sensitivity analysis was carried out to investigate the impact of different hyperparameters on the performance of the RL-based solution. The results indicate that the solution is relatively robust to variations in hyperparameter selection. Table 14 is utilized to incorporate the effect of entropy coefficient c (65) for all possible scenarios of the five seed machines with varying power multipliers, voltage multipliers, and target I_s/I_n levels for the model employing the distance reward function. As explained in 3.4.6, entropy Coefficient is a hyperparameter that controls the extent to which the agent's policy is encouraged to explore during training. The evidence indicates that the hyperparameter in question has little impact on both the PPO steps and optimal cases, with its sole effect being on the total number of iterations. This implies that the entropy coefficient hyperparameter does not exert a significant influence on the reward function, suggesting that the function is relatively insensitive to changes in this hyperparameter.

Table 14: Distance reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values, relative to the entropy coefficient. This assessment is conducted using simple ending criteria. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Entropy coef	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
0	5	15M	61%	96%	0	650
0.5	5	20M	58%	94%	0	650
0.8	5	22M	59%	94%	0	650

In addition, Table 15 is employed to evaluate the impact of the clip rate hyperparameter on all possible scenarios of the model that uses the distance reward function, considering the five seed machines with varying power multipliers, voltage multipliers, and target I_s/I_n levels. As explained in 3.4.6, the clip rate ϵ (65) refers to the maximum ratio by which the updated policy is allowed to deviate from the old policy during training. The available data suggests that the aforementioned hyperparameter has minimal influence on the overall number of PPO steps and iterations, but it does affect the total number of optimal solutions.

Table 15: Distance reward function applied to analyze five machines with varying power multipliers, voltage multipliers, and target I_s/I_n values, relative to the clip rate. This assessment is conducted using simple ending criteria. PPO/reference steps represents the total PPO steps divided by total reference steps across all cases.

Clip rate	Seed machines	Training steps	PPO/reference steps	Optimal solutions	Failed cases	Total cases
0.2	5	20M	58%	94%	0	650
0.5	5	25M	61%	98%	0	650

The study presents compelling findings that highlight the effectiveness and potential of using RL techniques in electrical machine design. In all of the experiments conducted, RL proved to be highly effective in reducing the number of design steps by approximately 50% compared to the required number of reference steps. This significant reduction demonstrates the efficiency and capability of RL in optimizing the learning process and achieving desired outcomes in a more efficient manner. Moreover, it is particularly noteworthy that the implementation of the final reward function, specifically the distance reward function, resulted in a remarkable outcome: there were no failed cases observed. This indicates the robustness and reliability of the reward function in guiding the RL agent towards successful and desirable outcomes consistently throughout the experiments. The results demonstrate the ability of RL to learn complex relationships between design variables and performance metrics, showcasing its potential to revolutionize conventional approaches to machine design and optimization.

5 Conclusion and future work

The thesis on the design of electrical machines using RL provides evidence that RL can be a promising approach to enhance the design process of electrical machines. In the results section 4.2, it is demonstrated that RL-based design approaches outperformed traditional methods by effectively reducing the number of steps necessary for the design process. This reduction in steps indicates that RL offers a more efficient and streamlined approach to designing electrical machines.

Furthermore, the thesis introduces a novel aspect in the form of a proposed distance reward function. This reward function is designed to guide the RL algorithm without the need for extensive engineering knowledge. This aspect enhances the power and applicability of RL in the design process, as it reduces the dependency on domain-specific expertise. By eliminating the requirement for specialized engineering knowledge, RL becomes a more accessible and versatile tool for enhancing the design process of electrical machines.

The use of RL algorithms enables the optimization of various design parameters of the machine, resulting in improved computational efficiency. With RL, the design process can be automated, reducing the need for manual adjustments and trial and error methods. It is a useful tool for designers to explore the design space of electrical machines, enabling them to find the best solutions to meet specific design requirements. RL algorithms can learn from previous design experiences and continuously improve the design process.

The study's findings highlight the advantages of using RL for the design of electrical machines, such as enhanced computational efficiency and faster performance compared to traditional optimization approaches. Despite the advantages, there are some limitations to using RL algorithms in designing electrical machines. One of the significant drawbacks is the requirement for extensive training data. The algorithms need to learn from previous design experiences to provide optimal solutions, which can be time-consuming and computationally expensive. The algorithms require careful tuning and selection of hyperparameters, and the lack of interpretability of the results can be a challenge for designers to understand and improve upon the design. To overcome these limitations, further research is needed to refine the RL algorithms and reduce the computational burden.

As for future work, there are several directions that can be explored. The first direction involves investigating the potential of RL in the design of various types of electrical machines, such as synchronous machines, to expand its applications beyond induction motors. The second direction is to explore alternative RL algorithms to determine the most effective one for the specific optimization problem at hand. Recent research, such as [67], has proposed novel RL algorithms that show promise for use in electrical machine design optimization.

Finally, it is essential to investigate the scalability of the approach to determine its feasibility in designing larger and more complex machines. It will require overcoming challenges such as increasing the amount of training data, optimizing the computational performance of the algorithms, and developing more efficient optimization methods. Successfully addressing these challenges will be critical to fully realize the potential of RL in the design of electrical machines.

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[2] Hongbign Wang, Xin Chen, Qin Wu, Qi Yu, Xingguo Hu, Zibin Zheng, and Athman Bouguettaya. Integrating reinforcement learning with multi-agent techniques for adaptive service composition. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(2):1–42, 2017.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] Mehmet Çunkaş and Ramazan Akkaya. Design optimization of induction motor by genetic algorithm and comparison with existing motor. *Mathematical and Computational Applications*, 11(3):193–203, 2006.

[5] T György and KA Biro. Genetic algorithm based design optimization of a three-phase induction machine with external rotor. In *2015 Intl Aegean Conference on Electrical Machines & Power Electronics (ACEMP), 2015 Intl Conference on Optimization of Electrical & Electronic Equipment (OPTIM) & 2015 Intl Symposium on Advanced Electromechanical Motion Systems (ELECTROMOTION)*, pages 462–467. IEEE, 2015.

[6] Dipesh Gaur and Lini Mathew. Optimal placement of facts devices using optimization techniques: A review. *IOP Conference Series: Materials Science and Engineering*, 331(1):012023, mar 2018.

[7] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36, 1989.

[8] Takahiro Sato and Masafumi Fujita. A data-driven automatic design method for electric machines based on reinforcement learning and evolutionary optimization. *IEEE Access*, 9:71284–71294, 2021.

[9] Yu-Min Hsueh, Veeresh Ramesh Ittangihal, Wei-Bin Wu, Hong-Chan Chang, and Cheng-Chien Kuo. Fault diagnosis system for induction motors by cnn using empirical wavelet transform. *Symmetry*, 11(10):1212, 2019.

[10] Muthiah Geethanjali and Hemavathi Ramadoss. Fault diagnosis of induction motors using motor current signature analysis: A review. *Advanced Condition Monitoring and Fault Diagnosis of Electric Machines*, pages 1–37, 2019.

[11] Wei Tong. *Mechanical design of electric motors*. CRC press, 2014.

[12] Anibal T De Almeida, Fernando JTE Ferreira, and Ge Baoming. Beyond induction motors—technology trends to move up efficiency. In *49th IEEE/IAS*

Industrial & Commercial Power Systems Technical Conference, pages 1–13. IEEE, 2013.

[13] Alberto Tenconi, Silvio Vaschetto, and Alessandro Vigliani. Electrical machines for high-speed applications: Design considerations and tradeoffs. *IEEE Transactions on Industrial Electronics*, 61(6):3022–3029, 2013.

[14] Tapani Jokinen, Valeria Hrabovcova, and Juha Pyrhonen. *Design of rotating electrical machines*. John Wiley & Sons, 2013.

[15] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[16] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[17] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[18] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[19] Yuxi Li. Reinforcement learning applications. *arXiv preprint arXiv:1908.06973*, 2019.

[20] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[21] Anders Jonsson. Deep reinforcement learning in medicine. *Kidney diseases*, 5(1):18–22, 2019.

[22] Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

[23] Hilbert J Kappen. An introduction to stochastic control theory, path integrals and reinforcement learning. In *AIP conference proceedings*, volume 887, pages 149–181. American Institute of Physics, 2007.

[24] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[25] Amirhosein Mosavi, Yaser Faghan, Pedram Ghamisi, Puhong Duan, Sina Faizollahzadeh Ardabili, Ely Salwana, and Shahab S Band. Comprehensive review of deep reinforcement learning methods and applications in economics. *Mathematics*, 8(10):1640, 2020.

[26] Ann Nowé, Peter Vrancx, and Yann-Michaël De Hauwere. Game theory and multi-agent reinforcement learning. In *Reinforcement Learning*, pages 441–470. Springer, 2012.

[27] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[28] David Rohde, Stephen Bonner, Travis Dunlop, Flavian Vasile, and Alexandros Karatzoglou. Recogym: A reinforcement learning environment for the problem of product recommendation in online advertising. *arXiv preprint arXiv:1808.00720*, 2018.

[29] M Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*, 55(7):1–38, 2022.

[30] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[31] Rick Durrett. *Probability: theory and examples*, volume 49. Cambridge university press, 2019.

[32] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.

[33] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order mdps. In *IJCAI*, volume 1, pages 690–700, 2001.

[34] W Bradley Knox and Peter Stone. Reinforcement learning from human reward: Discounting in episodic tasks. In *2012 IEEE RO-MAN: The 21st IEEE international symposium on robot and human interactive communication*, pages 878–885. IEEE, 2012.

[35] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more.* Packt Publishing Ltd, 2018.

[36] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[37] Tingwu Wang, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*, 2019.

[38] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*, 2020.

[39] Sergey Levine and Vladlen Koltun. Guided policy search. In *International conference on machine learning*, pages 1–9. PMLR, 2013.

[40] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292.

[41] Josiah P Hanna, Scott Niekum, and Peter Stone. Importance sampling in reinforcement learning with an estimated behavior policy. *Machine Learning*, 110(6):1267–1317, 2021.

[42] Divyam Rastogi. Deep reinforcement learning for bipedal robots. 2017.

[43] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, 7:133653–133667, 2019.

[44] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[45] Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

[46] Faustino Gomez and Jürgen Schmidhuber. Evolving modular fast-weight networks for control. In *International Conference on Artificial Neural Networks*, pages 383–389. Springer, 2005.

[47] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2013.

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

[49] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28, 2015.

[50] George Tucker, Andriy Mnih, Chris J Maddison, John Lawson, and Jascha Sohl-Dickstein. Rebar: Low-variance, unbiased gradient estimates for discrete latent variable models. *Advances in Neural Information Processing Systems*, 30, 2017.

[51] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[52] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[53] Tommi Jaakkola, Satinder Singh, and Michael Jordan. Reinforcement learning algorithm for partially observable markov decision problems. *Advances in neural information processing systems*, 7, 1994.

[54] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[55] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[56] A Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. In *Conference on robot learning*, pages 561–591. PMLR, 2018.

[57] Jana Mayer, Johannes Westermann, Juan Pedro Gutiérrez H Muriedas, Uwe Mettin, and Alexander Lampe. Proximal policy optimization for tracking control exploiting future reference information. *arXiv preprint arXiv:2107.09647*, 2021.

[58] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.

[59] Andreas Folkers, Matthias Rick, and Christof Büskens. Controlling an autonomous vehicle with deep reinforcement learning. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2025–2031. IEEE, 2019.

[60] Wenhan Xiong, Xiaoxiao Guo, Mo Yu, Shiyu Chang, Bowen Zhou, and William Yang Wang. Scheduled policy optimization for natural language communication with intelligent agents. *arXiv preprint arXiv:1806.06187*, 2018.

[61] Jan Peters and J Andrew Bagnell. Policy gradient methods. *Scholarpedia*, 5(11):3698, 2010.

[62] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225. IEEE, 2006.

[63] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[64] David Pollard. Asymptopia: an exposition of statistical asymptotic theory. *URL http://www. stat. yale. edu/pollard/Books/Asymptopia*, 2000.

[65] Surya T Tokdar and Robert E Kass. Importance sampling: a review. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(1):54–60, 2010.

[66] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[67] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.