UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Master Thesis

Utilizing Neural Networks for Generalizing Heliostat Calibration in Solar Tower Plants

vorgelegt der Fakultät für Informatik
der Universität Duisburg-Essen (Campus Essen) von

Leon Sabel
Drachenfelsstr. 58b
50939 Köln
Matrikelnummer: 3017033

Essen, den 22. Februar, 2025

Betreuung:	Dr. Marc Hesenius
Lehrstuhl:	Lehrstuhl für Software Engineering, insb. mobile Anwendungen
Erstgutachter:	Prof. Dr. Volker Gruhn
Zweitgutachter:	Mathias Kuhl
Studiengang:	Software and Network Engineering (M. Sc.)
Semester:	Wintersemester 2024/2025

Abstract

Heliostats play a crucial role in solar tower plants, where they reflect sunlight onto a central receiver to generate thermal energy. Ensuring precise heliostat alignment is essential for maximizing energy capture and system efficiency. However, traditional calibration methods, such as rigid body models, suffer from limited accuracy due to mechanical wear, environmental changes, and simplifying assumptions.

Neural networks have shown promise in heliostat calibration by learning complex relationships in tracking data, but their application has been limited by data requirements and a lack of generalization across heliostats. This thesis introduces Helioformer, a Transformer-based neural network designed for heliostat calibration, capable of predicting both normal vectors and direct motor positions while maintaining high accuracy.

The model was trained and evaluated using datasets from the Solar Tower Jülich (STJ), which is operated by the German Aerospace Center (DLR). The dataset includes heliostat positions, motor positions, and Sun positions. Results demonstrate that *Helioformer* significantly outperforms traditional rigid body models, reducing aimpoint offsets from 2.05 mrad (rigid body model) to 1.45 mrad when predicting normal vectors, and achieving even better accuracy of 1.31 mrad when predicting motor positions. In addition, this work introduces a method for comparing the accuracy of the motor position using more tangible aimpoint offsets, making the predictions of the motor position more interpretable.

These findings highlight the potential of deep learning to enhance heliostat tracking precision, reduce recalibration efforts, and improve the overall efficiency of solar tower plants.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Problem Statement . . . 2
1.3. Research Questions . . . 3
1.4. Goals of the Thesis . . . 3
1.5. Structure of the Thesis . . . 4

2. Theoretical Background **5**
2.1. Solar Tower Plant Operations and Heliostat Functionality . . . 5
2.1.1. Overview of Solar Tower Plants . . . 5
2.1.2. Heliostat Construction . . . 7
2.1.3. Tracking of Heliostat Systems . . . 7
2.1.4. Necessity and Calibration of Heliostat Systems . . . 8
2.2. Transformer Architecture . . . 10
2.2.1. Attention Mechanism . . . 10
2.2.2. Positional Encoding . . . 11
2.2.3. Encoder-Decoder Structure . . . 12
2.2.4. Adaptation of Transformer Architecture for Non-NLP Applications . . . 13

3. State of The Art in Heliostat Calibration **15**
3.1. Methods for Calibration of Heliostat Systems . . . 15
3.2. Neural Networks for Heliostat Calibration . . . 17

4. Methodology **20**
4.1. System Overview . . . 20
4.2. Dataset Description . . . 21
4.3. Data Cleaning and Selection . . . 22
4.4. Scenario Design . . . 23
4.5. Feature Engineering . . . 25
4.5.1. Vector Calculations . . . 25
4.5.2. Feature Selection . . . 26
4.6. Model Architecture . . . 27
4.7. Input Data Preparation . . . 29
4.8. Evaluation Metrics . . . 32
4.9. Training Process . . . 34
4.10. Reverse Inference for Motor Position Prediction . . . 36

5. Implementation **38**
5.1. Overview of *Helioformer* . . . 38

5.2. Dataset Preparation: The `Dataset_Creator` Module 40
5.2.1. Data Augmentation and Feature Engineering 40
5.2.2. Heliostat Selection Process 41
5.2.3. Batch Generation Using K-Means and Equal-Size Clustering 42
5.3. Data Handling: HeliostatDataModule 43
5.3.1. Data Preparation Pipeline 43
5.3.2. PyTorch Dataset and DataLoader Implementation 44
5.4. The *Helioformer* Model: Implementation and Architecture 45
5.4.1. Implementation of the Transformer Model in PyTorch . . . 45
5.4.2. Feature Embedding via Learnable Linear Projections . . . 47
5.4.3. Implementation in PyTorch Lightning 47
5.4.4. Reverse Optimization for Motor Position Prediction 50
5.5. Execution Engine 51
5.5.1. Configuration Management 51
5.5.2. Training Execution 51
5.5.3. Prediction and Model Evaluation 52
5.6. Result Visualization and Analysis 53
5.6.1. Prediction Error Analysis 53
5.6.2. Batch Execution and Summary Plots 53
5.6.3. Heliostat Field Visualization 53

6. Evaluation and Results 54
6.1. Overview of Evaluation Criteria 54
6.1.1. Prediction Accuracy Metrics 55
6.1.2. Generalization Capability 55
6.1.3. Computational Efficiency and Scalability 56
6.2. Scenario 1: Known Heliostat Evaluation 56
6.2.1. Prediction of Normal Vectors 56
6.2.2. Prediction of Motor Positions 58
6.3. Scenarios 2 and 3: Generalization to New Heliostats and Retraining 59
6.3.1. Performance on Unseen Heliostats 59
6.3.2. Improvement Through Retraining 60
6.4. Scenario 4: Cross-Target Generalization 62
6.5. Training Set Size Analysis 63
6.6. Encoder Input Size Analysis 64
6.7. Computational Efficiency and Scalability 65

7. Discussion 67
7.1. Addressing Research Questions and Objectives 67
7.2. Interpretation of Key Results 68
7.3. Implications for Solar Tower Plants 71

8. Conclusion and Future Research 72
8.1. Conclusion 72
8.2. Future Research 73

9. Bibliography **75**

Appendix **79**

B. Example Loss Curve and Learning Rate Schedule 79

A. Relevant Source Code 80

List of Figures

2.1. Illustration showing the working principle of a solar tower plant. . 6

2.2. A bird's-eye view of the Solar Tower Jülich (STJ), located in the town of Jülich, Germany. On the left, the tower houses the receiver as well as the Lagrangian targets. On the right is the heliostat field. 7

2.3. On the left, the backside of a heliostat is shown, highlighting the frame supporting the mirror facets, the stepper motors, and the stand. On the right, the front of a heliostat is displayed, featuring its four mirror facets. 8

2.4. Schematic representation of the heliostat tracking principle. The heliostat reflects sunlight from the Sun onto the receiver by aligning its mirror normal vector ($\vec{n}$) with the bisector of the Sun-to-Mirror vector ($\vec{s}$) and the Mirror-to-Receiver vector ($\vec{a}$). 9

3.1. Illustration of the Camera-Target Method. The image depicts a solar tower with multiple heliostats in the foreground. On the tower, three calibration targets are visible, with the main receiver positioned above Target 2. 16

3.2. An image of a Lagrangian target showing the focal point of a heliostat at its center, as captured by the STP's camera (left). The same image after applying image processing is shown on the right. The extracted outline of the focal spot and the computed aimpoint are highlighted in red, while the detected markers, which have known positions, are marked in green. 17

4.1. Visualization of the heliostat positions within the heliostat field, indicating the category to which each heliostat belongs. 22

4.2. Plot of the Sun's positions for an example heliostat, with the measurement points of a selected batch highlighted (left). Plot of the Sun's positions for the same heliostat, where the points are color-coded according to their batch affiliation (right). 23

4.3. Standard Transformer architecture (left) compared to the modified *Helioformer* model (right), which has been adapted for heliostat calibration by integrating feature-wise embeddings and structural modifications to more effectively process calibration data. 29

4.4. Histogram of aimpoint offsets with different fitted distributions, including a Weibull fit, a Gaussian KDE, and a Rayleigh fit. . . . 34

4.5. Illustration of the working principle of reverse inference for motor position prediction. 36

5.1. Overview of the *Helioformer* software architecture, illustrating the core components and data flow. The system comprises a Dataset Creator, Execution Engine, Transformer-based model, and visualization modules. 38

6.1. Histogram of angular offsets with a KDE for normal vector prediction in Scenario 1, achieving an accuracy of 1.45 mrad (left). Scatter plot of aimpoint predictions using normal vector prediction for an exemplary heliostat in Scenario 1 (right). 56
6.2. Aimpoint offset as a function of heliostat distance from the tower, comparing the *Helioformer* model with the rigid body model. . . 57
6.3. Aimpoint offset of the Helioformer model for all heliostats in the heliostat field based on normal vector predictions in Scenario 1. The color scale represents the mode offset in milliradians. 58
6.4. Histogram of angular offsets with a KDE for motor position prediction in Scenario 1, achieving 1.31 mrad accuracy (left). Scatter plot of aimpoint predictions using motor position prediction for an exemplary heliostat in Scenario 1 (right). 59
6.5. Histogram of angular offsets with a KDE for unseen heliostat evaluation in Scenario 2, showing an accuracy of 3.46 mrad (left). Scatter plot of aimpoint predictions for an exemplary unseen heliostat in Scenario 2 (right) 60
6.6. Histogram of angular offsets with a KDE after retraining in Scenario 3, showing an improved accuracy of 1.96 mrad (left). Scatter plot of aimpoint predictions after retraining for an exemplary heliostat in Scenario 3 (right) 60
6.7. Comparison of aimpoint offsets for all new heliostats before and after retraining. The left plot shows aimpoint offsets before retraining (Scenario 2), while the right plot presents aimpoint offsets after retraining (Scenario 3). The color scale represents the mode offset in milliradians. 61
6.8. Histogram of angular offsets with a KDE for inference on Target 3 in Scenario 4, showing an accuracy of 2.46 mrad (left). Scatter plot of aimpoint predictions for inference on Target 3 for an exemplary heliostat in Scenario 4 (right). 62
6.9. Histogram of angular offsets with a KDE for inference on Target 2 in Scenario 4, showing an accuracy of 1.79 mrad (left). Scatter plot of aimpoint predictions for inference on Target 2 for an exemplary heliostat in Scenario 4 (right) 63
6.10. Plot of mixed and worst-case accuracy as a function of training set size. The accuracy of the rigid body model is also included for comparison. 64
6.11. Plot of mixed and worst-case accuracy as a function of encoder input size. The accuracy of the rigid body model is also included for comparison. 65

A.1. Example of the loss curve during the training of the *Helioformer* model for Scenario 1, with a trainset size of 100 and an encoder input length of 20. 79
A.2. Example of the learning rate schedule used during the training of the Helioformer models, with a warmup over 5% of the epochs, starting at 0, warming up to 1.5×10^{-4}, and then decaying back to 0 over the course of the training. 79

List of Tables

4.1. Overview of the datasets created to cover the described scenarios. 25

Abbreviations

STP Solar Tower Plant

CST Concentrated Solar Thermal

CST Concentrated Solar Power

STJ Solar Tower Jülich

DLR German Aerospace Center

FFN Feedforward Network

CNN Convolutional Neural Network

RNN Recurrent Neural Network

MAE Mean Absolute Error

MSE Mean Square Error

KDE Kerndel Density Estimation

1. Introduction

1.1. Motivation

Climate change has become one of the most pressing global challenges, necessitating a transition to renewable energy sources to reduce greenhouse gas emissions and mitigate environmental impacts [1]. Among the various renewable energy options available, solar energy has emerged as a key contender because of its abundance and sustainability [2].

Solar energy can be harnessed through different technologies, primarily categorized into Photovoltaic (PV) and Concentrated Solar Thermal (CST). PV technology directly converts sunlight into electricity using semiconductor materials, making it a popular choice for residential and commercial applications [3]. In contrast, CST systems capture and convert solar radiation into heat energy, which can be used for various applications, such as power generation in conventional power plants or heat processing in industrial operations [4].

One promising CST technology is the Solar Tower Plant (STP). STPs typically consist of a central tower surrounded by a vast field of heliostats, hundreds to thousands of steerable mirrors that track the sun's movement throughout the day and focus the reflected sunlight onto a central receiver mounted on the tower [5]. The concentrated solar energy heats a working fluid, which can then be used to produce electricity via a turbine. The thermal energy can also be stored in a Thermal Energy Storage system, which is more cost-effective and scalable compared to battery storage [6, 7]. Additionally, high-temperature heat from CST can be directly utilized in industrial processes, such as driving chemical reactions or processing synthetic fuels [8, 9, 10].

Heliostats are a critical component of solar tower plants, accounting for approximately 50% of the total system cost [11]. Consequently, optimizing their design and functionality to achieve cost-effective solutions is a key focus of research and development efforts. The proper alignment of heliostats focal points is crucial to ensure optimal illumination of the receiver [12]. Uneven heating can lead to loss of efficiency and potential damage to the receiver due to localized overheating [13, 14].

Over time, heliostat tracking can drift out of alignment due to various structural and environmental factors, such as mechanical wear, thermal expansion, wind loads, and ground settling [15, 16]. Therefore, regular calibration of heliostats is essential to maintain optimal performance and maximize energy output. Several calibration techniques have been developed to address this challenge, most of which rely on geometric models to predict and correct the behavior of the heliostat [17].

Although physical models provide a solid foundation for heliostat calibration,

their accuracy is limited by simplifying assumptions and external uncertainties. A more precise control strategy could further improve efficiency and enhance the overall performance of the system. Recent advances in machine learning, specifically neural networks, offer a promising alternative to traditional physical models [18]. Neural networks have demonstrated the ability to learn complex patterns and relationships within data, which makes them suitable for heliostat calibration tasks [19].

Despite their advantages, neural networks require large amounts of data to train effectively. Since each heliostat must be calibrated individually and operational hours are limited by daylight availability, obtaining sufficient training data can be challenging and time-consuming, particularly in newly constructed solar power plants [20].

One possible solution to address the data scarcity issue is to develop neural networks that can generalize across multiple heliostats rather than rely on data from a single unit. Since heliostats within a plant are often constructed identically, a generalizing neural network that takes advantage of shared characteristics among different heliostats could enhance the accuracy and efficiency of calibration.

1.2. Problem Statement

Ensuring the precise alignment of heliostats in STPs is a complex and ongoing challenge that directly impacts the efficiency and operational reliability of the system. While traditional calibration methods based on geometric models provide a theoretical foundation for heliostat alignment, their practical effectiveness is often limited by the inability to account for dynamic real-world influences such as mechanical wear, environmental fluctuations, and structural shifts over time [16]. These limitations can lead to frequent recalibration needs and efficiency losses, necessitating more adaptive and data-driven solutions.

Machine learning techniques, particularly neural networks, have shown significant potential to overcome these limitations by leveraging historical operational data to identify complex patterns and dependencies. Unlike physics-based models, neural networks have the capacity to learn nonlinear relationships and adapt to variations in heliostat behavior. However, current approaches still face considerable challenges in terms of scalability and data efficiency [20]. The primary challenges include the following:

- **Data scarcity:** Since each heliostat must be calibrated individually, collecting sufficient training data is challenging, especially in new solar plants with limited operational hours.

- **Lack of generalization:** Current NN models are typically trained for individual heliostats, failing to take advantage of interdependencies within the heliostat field. This limits their ability to generalize across multiple heliostats.

To overcome these challenges, this thesis explores the development of a generalized neural network for heliostat calibration that takes advantage of structural uniformity among heliostats within a plant. Using calibration data from multiple heliostats, the proposed approach aims to improve calibration accuracy and reduce dependence on extensive data collection for each individual heliostat.

1.3. Research Questions

To guide the feasibility and implementation of this solution, the following research questions are formulated:

1. **How can neural networks be effectively utilized to develop a generalized approach for calibrating heliostats in a solar tower power plant, ensuring improved accuracy and efficiency compared to traditional methods?**
2. **What strategies can be employed to design and implement a robust neural network architecture that can generalize well across different heliostats, ensuring consistent performance and accuracy despite variations in environmental conditions and heliostat configurations?**

1.4. Goals of the Thesis

To address these research questions, the thesis sets the following goals:

- **Selection of a suitable neural network architecture:** A model will be chosen that aligns with the complexity and characteristics of the heliostat calibration problem.
- **Implementation and evaluation:** The selected architecture will be implemented and tested in various experimental scenarios to assess its performance and generalizability.
- **Development of effective strategies:** The thesis will propose best practices to develop, deploy, and maintain such neural networks in the operation of real-world solar power plants.

The research will leverage a preexisting dataset collected from the Solar Tower Jülich (STJ) [5], located in the German town of Jülich and operated by the German Aerospace Center (DLR). This dataset contains valuable calibration records, including heliostat positions, target positions, motor positions, and sun positions, enabling a thorough investigation into the potential of neural networks for heliostat calibration.

Ultimately, the findings of this research are expected to contribute to the improvement of the efficiency and accuracy of the heliostat calibration processes, leading to an improvement in the performance and cost-effectiveness of solar tower power plants.

1.5. Structure of the Thesis

This thesis is structured to provide a comprehensive understanding of the research problem, the proposed methodology, and the results achieved. The chapters are organized as follows.

- **Chapter 1: Introduction**
  Provides the motivation behind this work, formulates the research problem, and defines the research questions and objectives.

- **Chapter 2: Theoretical Background**
  Covers the fundamental principles of solar tower plants, including heliostat functionality, tracking, and calibration techniques. In addition, it introduces the transformer-based architecture used in this work.

- **Chapter 3: Related Work**
  Reviews existing heliostat calibration methods, including geometric and data-driven approaches. It also discusses the role of neural networks in similar applications and identifies gaps in current research.

- **Chapter 4: Methodology**
  Details the system design, data set preprocessing, feature engineering, and neural network architecture. It also explains the training strategy, evaluation metrics, and different experimental scenarios designed to assess model performance.

- **Chapter 5: Implementation**
  Describes the implementation of the *Helioformer* model, including the data pipeline, training setup, and software framework used for development.

- **Chapter 6: Results and Evaluation**
  Presents the performance of the *Helioformer* model under different experimental conditions, comparing it with traditional calibration approaches. The impact of training set size, encoder input size, and model generalization is analyzed in depth.

- **Chapter 7: Discussion**
  Interprets the key results, discusses the practical implications for solar tower plants, and reflects on the challenges and limitations encountered during development.

- **Chapter 8: Conclusion and Future Work**
  Summarizes the main findings, highlights the contributions of this work, and outlines directions for future research, including potential improvements in model architecture and real-world deployment strategies.

2. Theoretical Background

Solar tower plants rely on precise heliostat tracking and calibration to maximize energy efficiency and ensure optimal operation. This chapter provides an overview of the key principles underlying solar tower plant operations, including heliostat functionality, tracking methods, and calibration techniques. In addition, it introduces the transformer architecture, which serves as the foundation for the proposed neural network-based calibration approach. These concepts establish the basis for understanding the challenges associated with heliostat calibration and the motivation to employ advanced machine learning techniques.

2.1. Solar Tower Plant Operations and Heliostat Functionality

Solar tower plants utilize large arrays of heliostats to concentrate sunlight onto a central receiver, converting solar energy into thermal power. The precise tracking and calibration of these heliostats are essential for maintaining high efficiency and ensuring a uniform solar flux distribution on the receiver. This section provides an overview of STP operation and the role of heliostats. In addition, it explores the mechanics of heliostat movement, tracking algorithms, and structural components, which form the foundation for accurate and reliable solar tracking.

2.1.1. Overview of Solar Tower Plants

STPs are a form of concentrated solar power CSP technology. CSP systems generally utilize mirrors to reflect and concentrate incoming sunlight onto a designated receiver. This technology leverages the ability of mirrors to focus solar radiation, enabling the collection and conversion of solar energy into usable thermal energy.

The receiver, strategically positioned to capture the concentrated solar energy, collects heat and transfers it to a medium known as the working fluid. This working fluid is then utilized for various applications, such as power generation through steam turbines or as process heat in industrial chemical processes. The choice of working fluid varies depending on the intended application and the requirements.

Typical choices for working fluids are molten salts, water, synthetic oils, or even air, each of which offers distinct advantages [21]. One of the key benefits of some working fluids, such as molten salt, is their ability to store heat energy for extended periods [7]. This allows for energy availability even during non-sunny periods, enhancing the reliability and flexibility of solar power generation.

In STPs, the receiver is mounted at the top of a tall tower to maximize exposure to concentrated sunlight. The mirrors, known as heliostats, are constructed as an

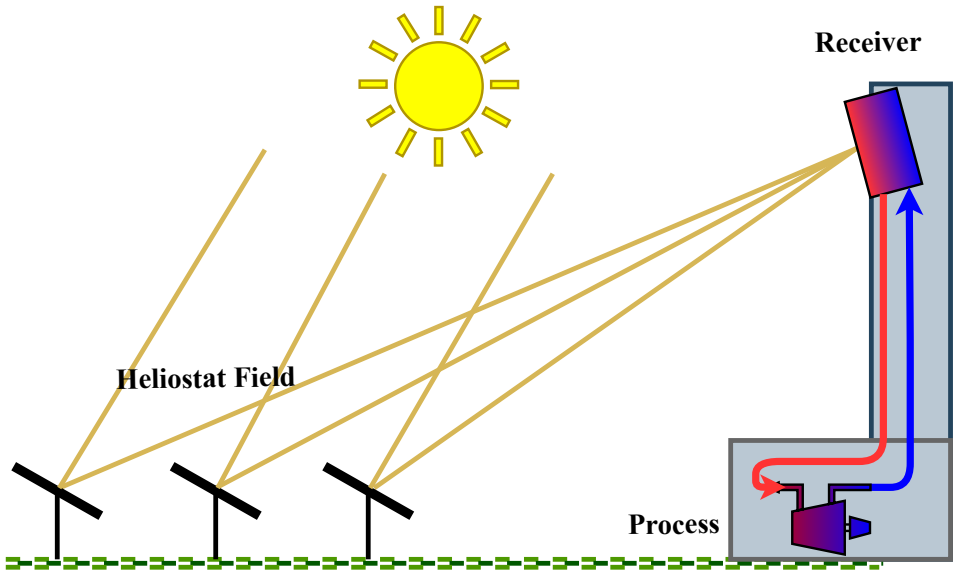

Figure 2.1.: **Illustration showing the working principle of a solar tower plant.**

array and are equipped with motors that can adjust their orientation to ensure optimal reflection of sunlight onto the receiver.

The configuration of the solar tower and the heliostat field is dependent on geographical considerations. In locations near the equator, the tower is typically located centrally within a circular heliostat field, ensuring uniform solar exposure throughout the day. In contrast, in more northern latitudes, the tower is often positioned on the southern side of the heliostat field to optimize solar-collection angles. Typically, there are multiple hundreds to multiple thousand Heliostats in a Heliostat field [22].

The receiver usually covers an area of several square meters, and uniform illumination is crucial to ensure efficient heat transfer. Uneven distribution of solar flux can lead to thermal stresses and reduced operational efficiency [23].

A tracking algorithm is used to determine the precise orientation of each heliostat. This algorithm considers multiple factors, including the position of the Sun, the heliostat's location relative to the receiver, the receiver's position, and several calibration parameters.

In addition to the main receiver, the solar tower typically includes one or more calibration targets, as seen in Figure 3.1. These targets serve as reference points for validating and adjusting the alignment of the heliostat. A heliostats focal spot can be directed toward the center of a calibration target to measure its tracking accuracy. The tracking algorithm subsequently fine-tunes the heliostat parameters to enhance the heliostat's precision.

By continuously adjusting and recalibrating the heliostat field, STPs aim to achieve optimal performance and efficient energy capture.

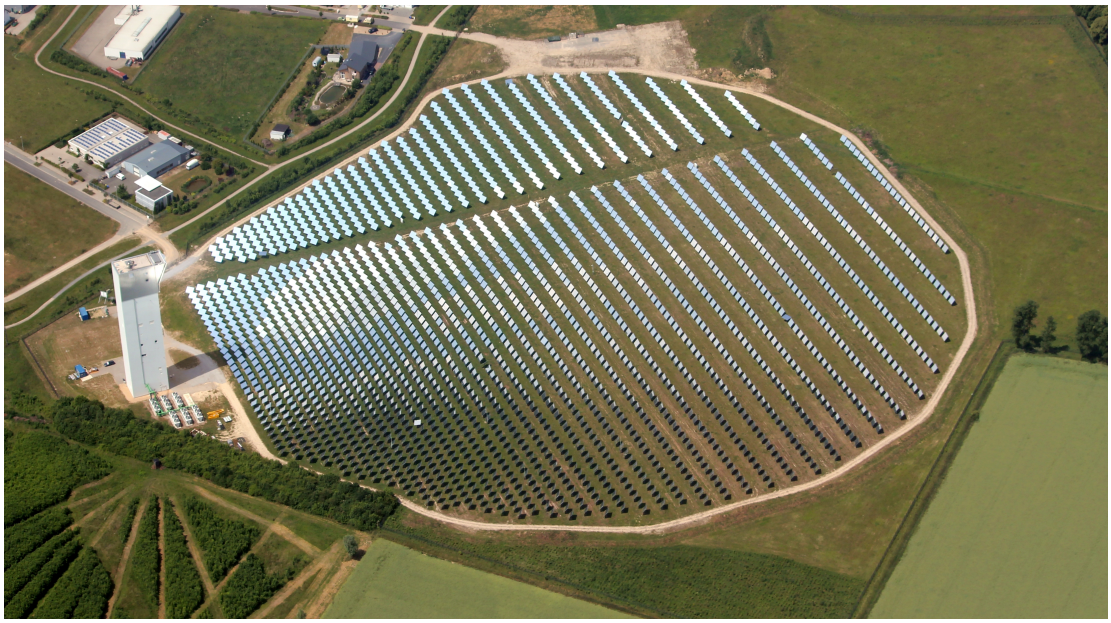

Figure 2.2.: **A bird's-eye view of the Solar Tower Jülich (STJ), located in the town of Jülich, Germany. On the left, the tower houses the receiver as well as the Lagrangian targets. On the right is the heliostat field.**

2.1.2. Heliostat Construction

In a typical heliostat field, individual heliostats are constructed with a uniform design. This standardization helps minimize costs and simplifies maintenance procedures. Each heliostat is composed of several key components: a stand, a frame, mirrors (often divided into multiple facets), and multiple stepper motors.

Most heliostats are designed to track the sun's position in both the azimuth and the elevation axis. This tracking capability is achieved using two stepper motors, which can be either linear, as seen in Figure 2.3, or circular, depending on the chosen design. These motors precisely adjust the orientation of the heliostat to ensure accurate solar tracking.

The mirror facets are individually mounted onto the frame and can be adjusted to ensure that their reflected beams converge at the receiver, forming a concentrated focal spot. This alignment ensures that concentrated solar energy is directed efficiently to the desired target. An example of such a heliostat can be seen in Figure 2.3.

2.1.3. Tracking of Heliostat Systems

The tracking principle is fundamental to the operation of heliostat systems, ensuring that mirrors precisely follow the sun's movement to direct sunlight onto a central receiver. In such a system, the positions of the Sun, the heliostat, and the receiver are known. The primary objective of the tracking algorithm is to determine the correct orientation of the heliostat to reflect incoming sunlight onto the receiver. Since the mirror can be mathematically approximated as a plane, its orientation is defined by the normal vector of the mirror. According to the law of reflection, the angle of incidence is equal to the angle of reflection. Using this

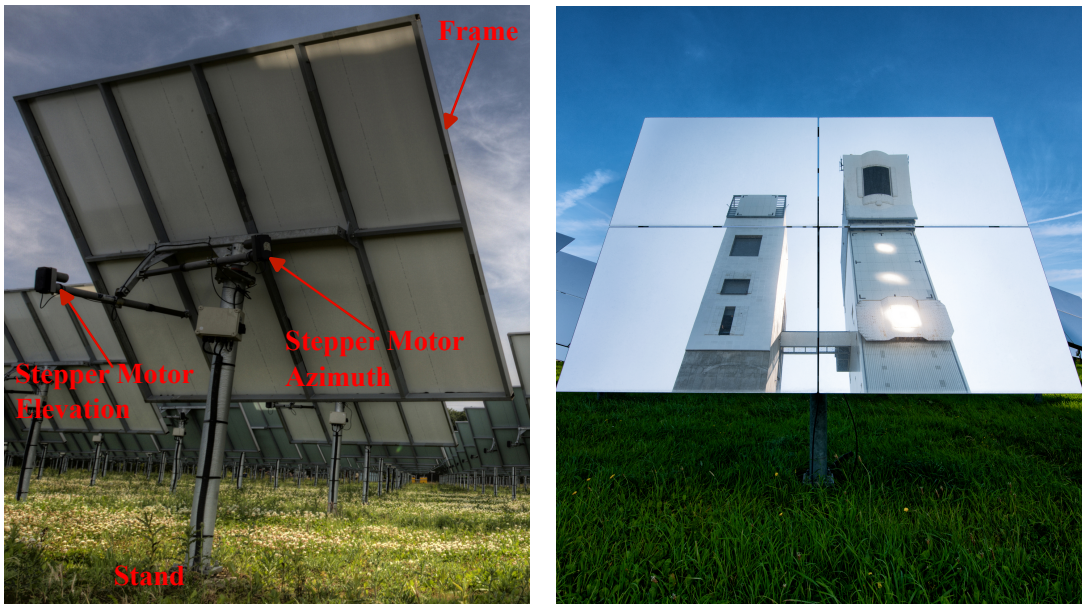

Figure 2.3.: **On the left, the backside of a heliostat is shown, highlighting the frame supporting the mirror facets, the stepper motors, and the stand. On the right, the front of a heliostat is displayed, featuring its four mirror facets.**

principle, the optimal orientation of the heliostat is achieved when the mirror's normal vector is aligned with the bisector of the angle formed between the Sun-to-Mirror vector and the Mirror-to-Receiver vector, as seen in Figure 2.4. This relationship can be expressed mathematically as:

$$\vec{n} = \frac{\vec{s} + \vec{a}}{|\vec{s} + \vec{a}|} \tag{2.1}$$

where $\vec{s}$ represents the unit vector from the heliostat to the Sun and $\vec{a}$ represents the unit vector from the heliostat to the receiver. By solving this equation, the ideal orientation of the heliostat is established to ensure optimal reflection of sunlight onto the receiver.

Once the desired normal vector is determined, a rigid body model is employed to calculate the necessary positions of the heliostat's stepper motors. This model is trained solely on calibration data, which includes motor positions, the Sun's position, and the corresponding aimpoint, allowing it to learn the relationship between these inputs and the resulting mirror orientation.

2.1.4. Necessity and Calibration of Heliostat Systems

Accurate tracking is critical to the efficiency of a heliostat system, as even minor deviations can lead to significant energy losses. Before a heliostat can be used effectively, an initial calibration is required to establish the relationship between its motor positions, the Sun's position the corresponding aimpoint locations. This step is crucial, as it provides the foundation for all subsequent tracking and calibration processes. Without this initial mapping, it would be impossible to accurately

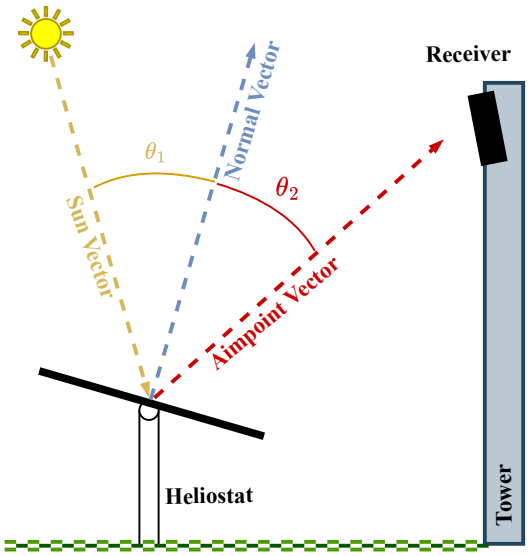

Figure 2.4.: **Schematic representation of the heliostat tracking principle. The heliostat reflects sunlight from the Sun onto the receiver by aligning its mirror normal vector ($\vec{n}$) with the bisector of the Sun-to-Mirror vector ($\vec{s}$) and the Mirror-to-Receiver vector ($\vec{a}$).**

predict the necessary motor adjustments for a given target. However, even after initial calibration, tracking errors can arise from various sources and require regular recalibration to maintain optimal performance. Some common sources of heliostat tracking errors include [16]:

- **Structural Deformation:** Over time, the structural components of the heliostat may undergo bending due to gravity, wind loading, or thermal expansion, causing misalignment.
- **Mirror Degradation:** Environmental exposure can reduce reflectivity, while mechanical stress and thermal fluctuations can distort the shape of the facet and misaligned mirrors, leading to loss of efficiency.
- **Mechanical errors:** Factors such as backlash in gears, wear and tear in actuators, and low-resolution encoders contribute to inaccuracies.
- **Installation errors:** These can include pedestal tilt, axis misalignment, initial misalignment of the mirror facets, and terrain leveling errors.

As these errors accumulate over time, the performance of the heliostat degrades, leading to a drift of the aimpoint and a misalignment of the focal spot, which can result in significant energy losses at the receiver.

To mitigate tracking errors and maintain accuracy, periodic calibration of heliostat systems is essential. Calibration ensures that the heliostat mirrors are correctly aligned with the intended target, improving system efficiency and reducing energy waste. Several calibration methods have been developed with the goal of achieving high precision and reliability, with the most widely used approach being the Camera Target Method.

2.2. Transformer Architecture

The Transformer architecture, introduced by Vaswani et al. in *Attention Is All You Need* [24], revolutionized deep learning by replacing recurrence and convolution with attention mechanisms. This approach enables parallel processing of input sequences, significantly improving training efficiency and performance across various tasks. Initially designed for natural language processing (NLP), Transformers have since been successfully adapted to fields such as computer vision, time-series forecasting, and reinforcement learning.

At its core, the Transformer follows an encoder-decoder architecture. The encoder extracts meaningful representations from the input sequences, while the decoder processes its own input sequence through a self-attention block before incorporating the encoder's representations to generate predictions. The key innovation lies in its use of attention, which allows the model to dynamically determine the importance of different elements within a sequence.

This section provides a detailed overview of the Transformer model, focusing on its key components:

- **Self-attention mechanism**, which allows flexible and context-sensitive information extraction.
- **Positional encoding**, which preserves the sequence order despite the fully parallelized nature of the model.
- **Encoder and decoder structure**, which facilitates sequence-to-sequence learning.

2.2.1. Attention Mechanism

The attention mechanism is the fundamental operation of the Transformer model [25]. Unlike traditional recurrent models, which process input sequentially, self-attention allows direct pairwise interactions between all elements of an input sequence. This enables the model to capture long-range dependencies without being constrained by the sequential nature of RNNs or the fixed-size receptive fields of CNNs.

I. Scaled Dot-Product Attention

The core computation of self-attention involves three learnable matrices, each serving a distinct role in determining how different elements of the input interact with one another:

- **Query** (**Q**): Represents the feature vector of the current token seeking contextual information. It is compared against all keys to determine the relevance of other tokens in the sequence.
- **Key** (**K**): Encodes distinguishing features of each token, serving as a reference point for similarity calculations. The attention mechanism determines the importance of each key in relation to a given query.

- **Value** ($\mathbf{V}$): Contains the actual contextual information associated with each token. Once attention scores are computed based on query-key similarity, the corresponding values are weighted and aggregated to refine the final representation.

Together, these matrices enable the model to dynamically focus on the most relevant parts of the input sequence, allowing it to capture complex dependencies and relationships across different elements.

Given an input sequence represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, where n is the sequence length and d is the feature dimension, the query, key, and value matrices are computed as:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V \tag{2.2}$$

where $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are learnable weight matrices. The attention scores are obtained by computing the dot product between the query and key matrices, scaled by the square root of the feature dimension d_k:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.3}$$

The **softmax** function ensures that attention weights are summed up to one, highlighting relevant tokens while suppressing irrelevant ones.

II. Multi-Head Attention

To improve expressiveness, the Transformer employs multi-head attention, computing multiple attention scores in parallel. Each attention head has independent weight matrices:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}_O \tag{2.4}$$

where each attention head is computed as:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_Q^i, \mathbf{K}\mathbf{W}_K^i, \mathbf{V}\mathbf{W}_V^i) \tag{2.5}$$

and $\mathbf{W}_O$ is a final projection matrix. Multihead attention allows different parts of the input sequence to attend to different contextual features, improving the model's ability to generalize.

2.2.2. Positional Encoding

Since transformers do not inherently process sequences in order, they require an additional mechanism to encode positional information. Without this, the model would treat all tokens as unordered.

The Transformer incorporates positional encodings, which are added to the input embeddings to provide sequence-order information. The most common ap-

proach is the Fixed Sinusoidal Encoding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{2.6}$$

where pos is the position index and d is the feature dimension. This method ensures that the model maintains an awareness of token order, which is essential to capture sequence dependencies.

2.2.3. Encoder-Decoder Structure

The Transformer model follows a sequence-to-sequence structure, consisting of an encoder and a decoder. Although both components are built from self-attention layers and feedforward networks, they serve different purposes.

I. Encoder

The encoder processes the input sequences and extracts meaningful features. It consists of multiple identical layers, each containing:

- **Multi-head self-attention layer:** The queries (Q), keys (K), and values (V) all come from the same input sequence, allowing each token to attend to all others.
- **Feedforward Network (FFN):** The layer applies nonlinear transformations.
- **Layer normalization and dropout:** These layers improve stability and prevent overfitting.

Encoder Forward Pass:

1. Input embeddings and positional encodings are added together.
2. The self-attention mechanism computes contextualized representations.
3. The feedforward network applies additional transformations.
4. The output is passed to the next encoder layer.

The final output of the encoder is a set of hidden representations containing contextualized information from the input sequence.

II. Decoder

The decoder generates output sequences taking care of both encoder representations and additional input from the decoder. It consists of:

- **Masked multi-head self-attention:** Like standard self-attention, but with a mask ensuring that each query (Q) only attends to keys (K) and values (V) of previous or current tokens, enforcing autoregressive decoding.

- **Cross-attention layer:** The queries (Q) come from the decoder, while keys (K) and values (V) come from the output of the encoder, allowing the decoder to incorporate information from the input sequence of the encoder.
- **Feedforward network:** The layer applies non-linear transformations.
- **Final output projection layer:** Maps the processed decoder outputs to the desired output space, typically by applying a linear transformation followed by a softmax function to generate predictions.

Decoder Forward Pass:

1. Embeddings and positional encodings of decoder inputs are added together.
2. Embeddings are passed through masked self-attention.
3. Cross-attention allows interaction with encoder outputs.
4. Feedforward layers apply transformations.
5. The final layer generates an output.

By combining the encoder and decoder, the Transformer enables efficient and flexible sequence-to-sequence learning.

2.2.4. Adaptation of Transformer Architecture for Non-NLP Applications

While the Transformer model was originally developed for NLP tasks, its underlying mechanisms, particularly self-attention and multi-head attention, allow it to be adapted to a wide range of applications beyond text-based problems [26]. However, when applied to domains that do not involve traditional sequence-to-sequence learning, such as heliostat calibration, time series forecasting, or computer vision, certain modifications to the standard Transformer architecture may be necessary.

One major consideration is the nature of the input data. Unlike NLP tasks, where word order plays a crucial role, many structured datasets lack a natural sequential ordering. In such cases, self-attention can still be leveraged to capture dependencies between data points, but the model must be designed to accommodate structured numerical data rather than discrete token sequences. Additionally, the positional encoding mechanism, which is essential for preserving word order in language processing, may need to be replaced or modified. Since the standard sinusoidal encodings may not be meaningful for non-sequential data, alternative embedding techniques, such as feature-wise learned embeddings, can be employed to represent input features more effectively.

The structure of the decoder may also require adjustments depending on the task. In problems where sequential output generation is unnecessary, the decoder can be either simplified or entirely removed, allowing the model to directly map

encoder representations to the desired output without additional autoregressive processing. This reduces computational overhead while maintaining the benefits of attention-based learning. Furthermore, in large-scale datasets or applications where local interactions are more relevant than global dependencies, alternative attention mechanisms, such as sparse attention or hybrid attention approaches, can be introduced to enhance efficiency and performance.

By tailoring the Transformer architecture to specific use cases, its powerful attention mechanism can be effectively utilized across a broad spectrum of fields, including scientific modeling, control systems, and heliostat calibration, providing a versatile framework for learning complex relationships within structured data.

3. State of The Art in Heliostat Calibration

This chapter examines existing methods for heliostat calibration, highlighting both traditional geometric approaches and emerging data-driven techniques. It first explores established calibration strategies, with a focus on the Camera Target Method, which provides accurate aimpoint measurements. The chapter then discusses recent advances in applying neural networks for heliostat calibration, emphasizing the challenge of data scarcity. Finally, it reviews methods that aim to improve accuracy while minimizing data requirements.

3.1. Methods for Calibration of Heliostat Systems

Accurate calibration of heliostat systems is essential to optimize the performance of solar tower power plants. Various calibration methods have been developed to ensure precise solar tracking and reflection. According to Sattler et al. [17], these methods can be broadly categorized as follows:

- **On-Field Observation Methods**: These involve direct observation of the heliostat's reflected beam on a target surface. By analyzing the deviation between the intended and actual reflection points, adjustments can be made to correct tracking errors.
- **Photogrammetry-Based Methods**: Using photographic techniques, these methods reconstruct the spatial geometry of heliostats to detect misalignments or structural deformations.
- **Laser Tracker Systems**: Employing laser beams, these systems measure the precise position and orientation of heliostat components, facilitating high-accuracy calibration.
- **Camera-Based Methods**: Cameras are used to capture images of the heliostat or its reflected beam, and image processing algorithms determine alignment errors.

Among these, the Camera Target Method has gained prominence due to its effectiveness and practicality. This method relies on the precise measurement of the reflected focal point of a heliostat, also called an aimpoint, to determine its tracking accuracy and correct systematic errors. The calibration process begins with a heliostat that directs its reflected sunlight to a designated target surface, typically a Lambertian white board positioned near the receiver, as shown in Figure 3.1. This target provides a uniform, diffuse reflective background, ensuring that the reflected light spot is clearly visible for image processing, aswell as several fixed markers with known spacial positions.

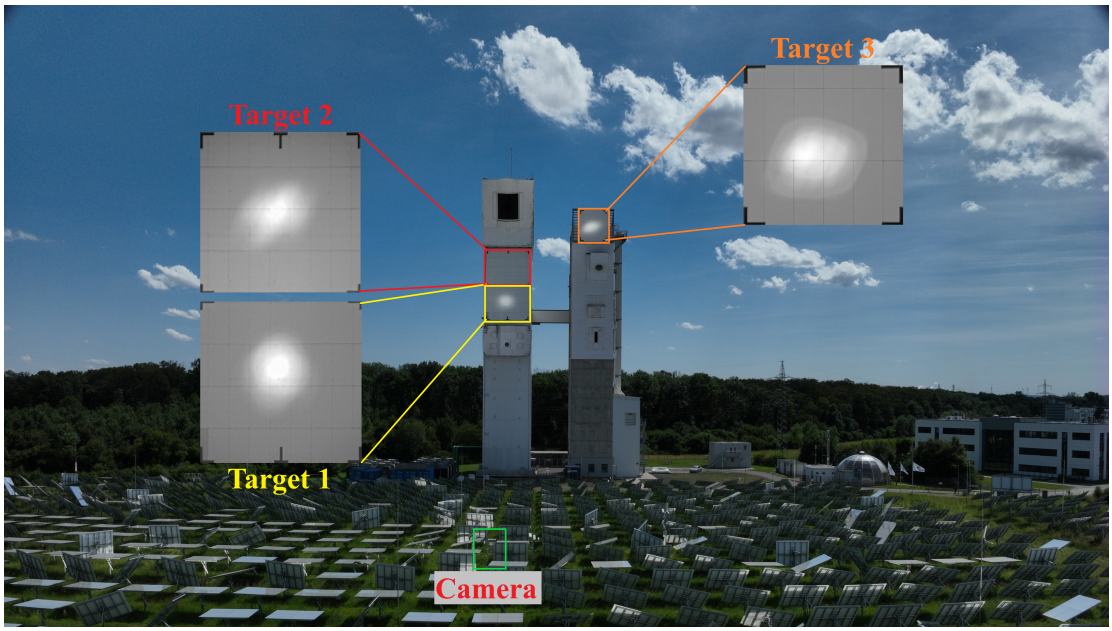

Figure 3.1.: **Illustration of the Camera-Target Method. The image depicts a solar tower with multiple heliostats in the foreground. On the tower, three calibration targets are visible, with the main receiver positioned above Target 2.**

Once the heliostat aligns its reflection onto the target, a camera captures an image of the focal spot, where the concentrated solar flux appears as a bright region. To extract the aimpoint, the centroid of the reflected beam, machine vision algorithms process the image in several steps.

First, the system detects reference markers on the calibration target, which serve as fixed points for spatial alignment. The image is then cropped and rectified to correct for perspective distortions. Next, filtering techniques, reduce noise, while thresholding isolates the high-intensity pixels corresponding to the focal spot. The centroid of this bright region is computed, determining the location of the aimpoint in the image coordinates.

Finally, the system translates the detected aimpoint into the three-dimensional coordinate system of the solar field. Using known marker positions and camera parameters, this transformation accounts for lens distortion, perspective correction, and relative positioning. The result is a precise spatial localization of the aimpoint, ensuring accurate heliostat calibration. Figure 3.2 illustrates this process, showing the captured image on the left and the processed result on the right, where the extracted focal spot, aimpoint, and detected markers are highlighted.

Each measurement cycle, from directing the heliostat to capturing and processing the image, takes approximately 30 seconds. This duration includes the time required for the heliostat to move to its designated position, stabilize, and allow the camera system to acquire and analyze the image. Because this method relies on the presence of direct sunlight to generate a clear and distinguishable focal spot, its effectiveness is dependent on sufficient solar irradiance and clear atmospheric conditions. Poor weather, such as cloud cover or heavy fog, can reduce the quality of the measurement or make calibration impossible.

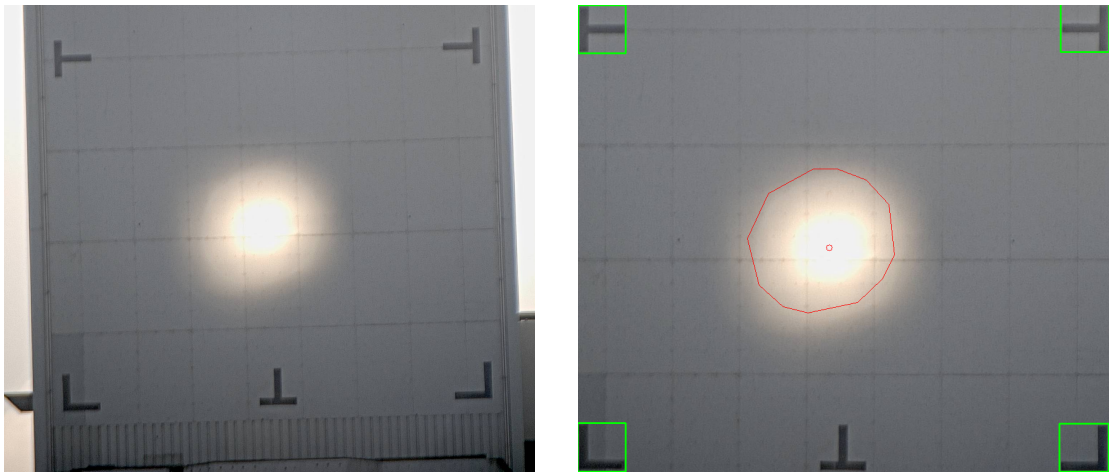

Figure 3.2.: **An image of a Lagrangian target showing the focal point of a heliostat at its center, as captured by the STP's camera (left). The same image after applying image processing is shown on the right. The extracted outline of the focal spot and the computed aimpoint are highlighted in red, while the detected markers, which have known positions, are marked in green.**

Once the spatial coordinates of the aimpoint have been determined, they are stored in a database alongside additional calibration parameters, including the heliostat motor positions and the Sun's position at the time of measurement. These data points collected form a data set of the measurement conditions and the corresponding tracking deviations. Over time, as multiple heliostat aimpoints are recorded under varying solar conditions, a comprehensive data set is built, enabling long-term monitoring and refinement of heliostat tracking models.

The final step in the calibration process involves using these measurement points to train or update predictive models that improve the accuracy of heliostat tracking. Traditionally, this is achieved using regressive rigid body models, which establish a mathematical relationship between the Sun's position, the heliostats motor positions of the heliostat and its reflective target point.

The Camera Target Method is widely adopted in commercial solar tower facilities due to its balance of accuracy, automation potential, and operational efficiency. However, its dependence on clear skies and consistent sunlight limits its applicability, as it requires the sun to shine. This restricts its operation during nighttime and in adverse weather conditions. The ongoing research aims to improve the robustness of this method by improving image processing techniques, integrating redundancy in sensor systems, and exploring alternative calibration methods that can function effectively under various environmental constraints.

3.2. Neural Networks for Heliostat Calibration

Traditional heliostat calibration methods, such as regression-based rigid-body models, rely on predefined mathematical representations of heliostat misalignment. However, these approaches struggle to capture complex, non-linear errors

introduced by mechanical imperfections, thermal expansion, or long-term degradation. To address these challenges, neural networks have been investigated as an alternative approach for heliostat calibration. Neural networks offer the ability to model complex relationships within data without the need for explicit understanding of physical relations, making them well suited for tasks where many interacting variables influence system performance.

The data required to train such neural networks is typically collected using the Camera Target Method, as described in the previous section. Each measurement consists of the Sun's position, heliostat motor positions, and the measured aimpoint recorded in a database. However, a major limitation of neural network-based heliostat calibration is the scarcity of available calibration data per heliostat. Since each heliostat must be individually calibrated, the amount of training data for a single unit remains relatively small. This limited data set size poses a challenge for deep learning models, which generally require large amounts of data to generalize effectively.

One of the first studies to investigate deep learning for heliostat calibration under these data constraints was conducted by Pargmann et al. [18]. Their work explored the use of pretraining techniques to improve the effectiveness of neural networks despite limited data availability. Three different pretraining approaches were compared:

- **Supervised pretraining using synthetic data:** A raytracer simulation was used to generate a large dataset of synthetic calibration points, which were then used to initialize the network.
- **Supervised pretraining using a baseline rigid-body model:** Instead of using synthetic data, the neural network was first trained on predictions from the conventional geometric model before being fine-tuned on real measurement data.
- **Unsupervised pretraining using stacked autoencoders:** The network was pretrained in an unsupervised manner to learn structural features within the data before being fine-tuned with labeled calibration points.

Their results demonstrated that pretraining significantly improved neural network performance, allowing the model to achieve accuracy comparable or superior to that of the rigid-body method while requiring as little as 300 calibration points per heliostat. However, a key limitation remained, as each trained model was specific to a single heliostat and could not generalize to others. This means that each heliostat required an individual neural network, limiting the scalability of the approach.

To further address the data scarcity problem, Pargmann et al. [27] proposed a hybrid approach that integrates both rigid-body kinematics and neural networks. Their model consists of two layers:

- **Rigid-body geometric model:** Provides an initial physics-based estimation of the expected behavior of the heliostat.

- **Neural network disturbance model:** Refines this estimation by learning additional dynamic variations that are difficult to capture with pure geometry-based methods.

This hybrid approach ensures that the worst-case accuracy is always at least as good as the rigid-body model, while allowing the neural network to improve predictions where the physics-based model falls short. By combining the strengths of both methods, the hybrid model was able to achieve top performance with fewer than 100 calibration points per heliostat, significantly reducing the required data compared to purely data-driven models.

Although these results are promising, existing neural network-based approaches for heliostat calibration have been primarily trained and evaluated only for individual heliostats. This means that each heliostat is treated as an independent system, without leveraging knowledge from other heliostats in the field. Given that many systematic errors, such as actuator misalignments and structural deformations, affect multiple heliostats in similar ways, there is a potential to develop a generalized neural network model that learns from the calibration data of all heliostats collectively. This model could improve performance by identifying shared error patterns throughout the heliostat field, leading to more efficient and robust calibration methods.

4. Methodology

This chapter outlines the methodology employed to develop and evaluate a generalizing neural network-based approach for heliostat calibration. It first provides an overview of the system components, including the heliostat field, calibration targets, and data collection process. The chapter then details the dataset structure, cleaning procedures, and sampling strategies used to ensure high-quality training data. A key focus is placed on feature engineering, where domain-specific transformations enhance the model's ability to generalize across heliostats. The chapter further describes the transformer-based model architecture, highlighting key modifications tailored to heliostat calibration. Finally, it presents the training procedure, evaluation metrics, and experimental scenarios designed to assess the model's accuracy, generalization ability, and robustness under different conditions.

4.1. System Overview

The system consists of several key components that enable precise solar tracking and reflection. A fundamental external factor influencing the system is the Sun, which follows a predictable yet continuously changing trajectory across the sky. This movement determines the necessary adjustments for each heliostat to ensure accurate reflection onto the designated target.

The heliostat field consists of more than 2000 individual heliostats, each equipped with reflective mirrors designed to track the Sun's movement and direct sunlight to a designated target. Each heliostat is controlled by two stepper motors that adjust the mirror's tilt and rotation angles, ensuring that sunlight is accurately reflected. The position of these motors is known, providing essential data for system calibration and performance optimization. Despite the precise control mechanisms in place, factors such as mechanical tolerances, environmental influences, and inherent system uncertainties introduce challenges in maintaining accurate aimpoints.

To ensure accurate positioning and tracking, the system includes three stationary calibration targets, which serve as reference points for evaluating the heliostats' performance. These targets are observed using a camera system that captures the reflection pattern and determines the aimpoints position.

The primary objective of the neural network is to model this entire system with sufficient accuracy and generalization capability. Ideally, the network should be able to predict either the aimpoint position on the calibration targets based on known heliostat motor positions and Sun location, or conversely, estimate the required motor positions to achieve a desired aimpoint given the Sun's position. A key requirement for the neural network is its ability to generalize across the

heliostat field rather than merely learning the specific characteristics of individual heliostats.

4.2. Dataset Description

The dataset[1] used in this study was sourced from the Solar Tower Jülich (STJ) [5], which is located in the German town of Jülich. This data set is pre-existing and provides valuable real-world data to develop and validate the proposed neural network model. It contains approximately 300,000 entries collected over a time period of 3 years (2021–2024). The data was collected using a camera-target method, which is described in Section 3.1. During each measurement, several parameters are recorded to create a comprehensive data entry referred to as a **Calibration**. The data set is structured to include the following parameters:

- **Heliostat ID:** A unique identifier for each heliostat in the heliostat field.
- **Heliostat positions:** The recorded positions of the heliostats are provided in local East-North-Up (ENU) coordinates.
- **Aimpoint coordinates:** The location where reflected sunlight is directed is captured in local ENU coordinates.
- **Sun positions:** The position of the Sun at the time of calibration is documented in ENU coordinates.
- **Motor 1 position:** Integer value representing the position of the first stepper motor.
- **Motor 2 position:** Integer value representing the position of the second stepper motor.
- **Target ID:** A unique identifier assigned to each calibration target on the towers.
- **Deleted:** A boolean value set once per data point and updated if necessary by the operator. This flag is used to mark invalid calibrations, for example, if changes or errors occur in the heliostat.
- **Picture of the Aimpoint:** An image of the reflected aimpoint captured during calibration, as seen in Figure 3.2.

This data set forms the foundation for training and validating the neural network model, ensuring that it can accurately learn the relationships between heliostat positions, motor controls, and aimpoint locations. The rich combination of numerical and visual data allows for comprehensive analysis and model optimization, ultimately enhancing the performance of the heliostat field.

[1] `https://paint-database.org/`

4.3. Data Cleaning and Selection

The dataset used for training the neural network requires thorough cleaning and selection to ensure high-quality data input. The calibration data is manually labeled each day to identify and exclude faulty calibrations, such as those affected by poor visibility conditions or multiple focal spots on a target. The calibrations that are labeled as deleted are excluded.

To facilitate reliable comparisons across different heliostats, a preselection process was implemented based on the number of available calibrations. Heliostats with 125 or more calibrations are labeled as **known heliostats** and used for training, while heliostats with 30 or more calibrations, not already included in the known heliostat category, are labeled as **new heliostats** to later be able to assess the model's generalization capabilities on completely unseen heliostats. In Figure 4.1, the spatial arrangement of the heliostats within the field is displayed, highlighting the specific heliostats selected for each dataset type.

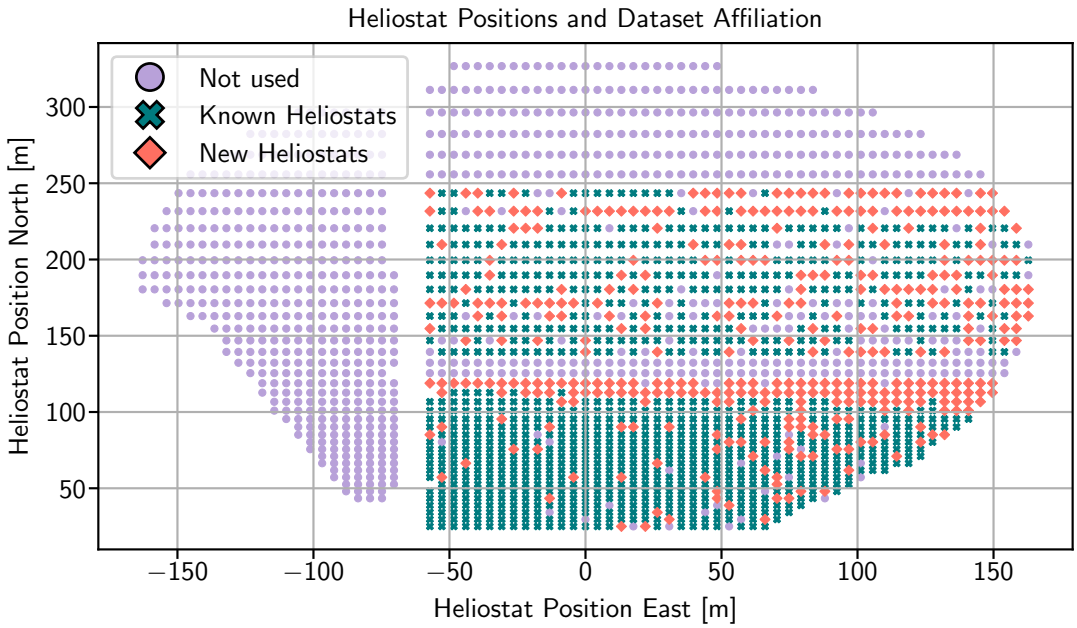

Figure 4.1.: **Visualization of the heliostat positions within the heliostat field, indicating the category to which each heliostat belongs.**

To allow flexibility in model training, the dataset is divided into multiple batches based on predefined training set sizes and calibration instances per heliostat. The batching algorithm allows to specify the number of batches and the desired batch size. To ensure an even distribution of calibrations with respect to the Sun's positions, inside each batch, a kmeans clustering algorithm for the initial selection of the used calibrations and a custom Equal-Size Clustering algorithm are used in the batching algorithm. An detailed description of this algorithm can be seen in 1 This stratification aims to maintain consistency across different training scenarios, avoiding potential biases in the distribution of the Sun's position. Then, one of these batches is designated as the validation set, ensuring a robust framework for performance evaluation. An example of the distribution of the Sun position with

the selected batches can be seen in Figure 4.2.

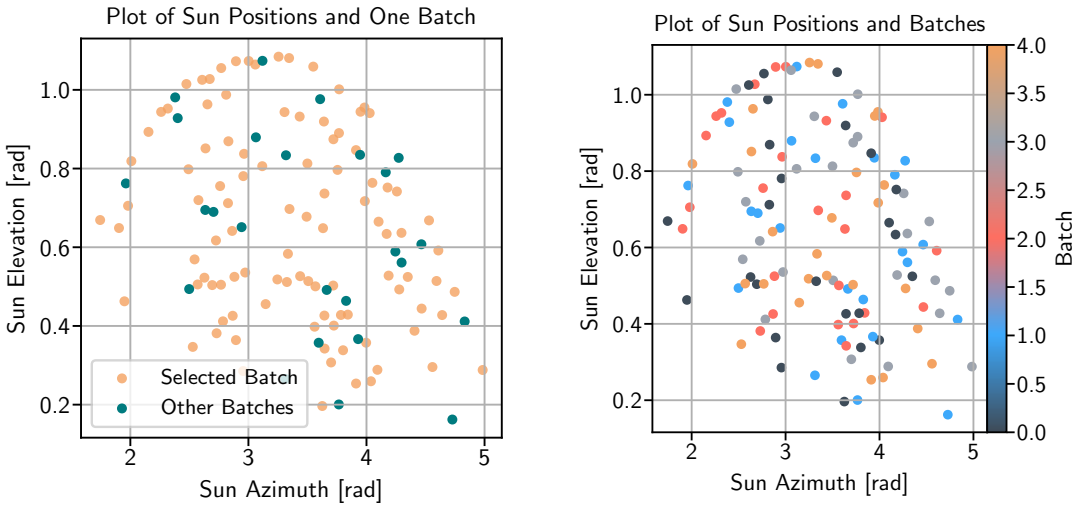

Figure 4.2.: **Plot of the Sun's positions for an example heliostat, with the measurement points of a selected batch highlighted (left). Plot of the Sun's positions for the same heliostat, where the points are color-coded according to their batch affiliation (right).**

4.4. Scenario Design

To thoroughly evaluate the performance of the proposed neural network model, a series of well-structured experimental scenarios is designed. These scenarios aim to test the generalizability, robustness, and ability of the model to adapt to varying conditions within the heliostat field. By structuring the training and validation processes in different ways, it is possible to gain a deeper understanding of how well the model performs under various real-world conditions and whether it can effectively handle new data that it has not been explicitly trained on.

The first scenario involves training and validation on the same set of known heliostats. In this case, the model is trained using data from a specific subset of heliostats and subsequently validated using data from the same heliostats. This scenario provides a baseline performance assessment, as the model is exposed to familiar conditions during both training and validation. It allows analyzing the ability of the model to capture intricate patterns within the dataset and to measure its accuracy when applied to similar data. However, this approach does not provide insight into the model's ability to generalize to previously unseen heliostats or configurations.

The second scenario involves training on known heliostats and validating on new heliostats. Here, the model is trained on data collected from a subset of heliostats and tested on a different set that was not included in the training phase. This setup simulates real-world deployment conditions in which the system may encounter heliostats with slightly different characteristics, placements, or operating conditions. Evaluating the model in this scenario helps assess its ability

Algorithmus 1 Sampling Heliostat Calibration

```
Input: Dataset df with heliostat calibrations
       dataset_size (total number of samples to select)
       size_cluster (number of samples per batch)
Output: Sampled dataset df_sampled with selected calibration points

                    ▷ Step 1: Select Maximally Distant Calibration Points per Heliostat
for each Heliostat h ∈ df do
    Extract sun position features (Sun_Azimuth, Sun_Elevation)
    Apply K-Means clustering with n = dataset_size clusters
    Compute centroids of each cluster
    Assign the closest real calibration point to each centroid
    Store selected points
end for

                                ▷ Step 2: Partition Data into Equal-Sized Clusters
for each Heliostat h ∈ df do
    Sort the selected points by azimuth and elevation
    Partition into size_cluster equal-sized clusters
    for each Cluster c do
        for each Subset s = 1 to (dataset_size/size_cluster) do
            Randomly select one point from cluster c
            Assign point to subset s
            Remove selected point from further sampling
        end for
    end for
end for

                                                ▷ Step 3: Return Sampled Dataset
Return df_sampled containing the final selected calibration points
```

to generalize and adapt to unseen heliostat configurations, ensuring that it is not overfitting to specific data patterns present in the training set.

The third scenario introduces a retraining phase with newly added heliostats. In this case, the model is initially trained on the known heliostats, as in the second scenario, and subsequently, it is retrained for a small number of epochs with data from newly introduced heliostats added to the training set. This scenario tests the model's adaptability when new heliostats are introduced into an existing solar tower plant, simulating an expansion scenario. It helps assess how efficiently the model can incorporate new data without the need for extensive retraining and whether it can retain its previously learned knowledge while adapting to new elements.

The fourth scenario consists of training on two targets and validating on the third target. In this scenario, the model is trained using calibration data from two specific targets within the heliostat field and then evaluated on a completely new third target. This experimental setup is designed to mimic real operational conditions, where the heliostat field is expected to reflect sunlight onto a central

receiver rather than calibration targets. By testing the model's performance on an unseen target, it is possible to determine whether the model can accurately infer aimpoints and motor positions under different targeting conditions. This scenario also provides insight into how well the model can extrapolate the learned relationships to new spatial configurations.

These carefully designed scenarios enable a comprehensive evaluation of the model's capabilities, identifying its strengths and potential areas for improvement. By analyzing the results in different scenarios, the reliability and suitability of the neural network for deployment in real-world heliostat field operations can be determined. A list of all created datasets to enable these scenarios can be seen in Table 4.1.

Name	Encoder Input	Heliostats	Size	Notes
DS_25_25	20	916	25	
DS_50_25	20	916	50	
DS_75_25	20	916	75	
DS_100_25	20	916	100	
DS_110_10	5	916	110	
DS_105_15	10	916	105	
DS_70_35	30	916	70	
DS_45_45	40	916	45	
DS_100_T3	20	770	100	Evaluate on Target 3
DS_100_T2	20	627	100	Evaluate on Target 2
DS_30_NH	20	376	30	New Heliostats

Table 4.1.: **Overview of the datasets created to cover the described scenarios.**

4.5. Feature Engineering

In designing the feature set for the neural network, careful consideration was given to selecting features that would not only provide the model with sufficient information but also encourage generalization across different heliostat configurations. The data set includes a variety of potential inputs, and a distinction can be made between standard baseline characteristics and engineered characteristics designed to enhance the model's ability to learn meaningful patterns beyond simple memorization.

4.5.1. Vector Calculations

To construct meaningful feature representations, several vector operations were performed based on ENU (East-North-Up) coordinate data. The key formulas used in the feature engineering process are as follows:

I. Building a Vector from Two Points

Given two points $\mathbf{P}_1(E_1, N_1, U_1)$ and $\mathbf{P}_2(E_2, N_2, U_2)$, the vector $\mathbf{v}$ from $\mathbf{P}_1$ to $\mathbf{P}_2$ is calculated as:

$$\mathbf{v} = (E_2 - E_1, N_2 - N_1, U_2 - U_1) \tag{4.1}$$

The magnitude of the vector is then given by the following:

$$\|\mathbf{v}\| = \sqrt{(E_2 - E_1)^2 + (N_2 - N_1)^2 + (U_2 - U_1)^2} \tag{4.2}$$

The unit vector can then be calculated as follows:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} \tag{4.3}$$

II. Conversion of a Vector to Azimuth and Elevation

To convert the unit vector $\hat{\mathbf{v}}(v_E, v_N, v_U)$ into azimuth and elevation angles:

Azimuth (θ), measured clockwise from the north:

$$\theta = \tan_2^{-1}(v_E, v_N) \mod 2\pi \tag{4.4}$$

Elevation (ϕ), measured from the horizontal plane:

$$\phi = \sin^{-1}(v_U) \tag{4.5}$$

III. Computing the Normal Vector from Two Vectors

Given two unit vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, the normal vector is computed as the sum of the two:

$$\mathbf{v}_{\text{normal}} = \mathbf{v}_1 + \mathbf{v}_2 \tag{4.6}$$

Normalization of the normal vector:

$$\hat{\mathbf{v}}_{\text{normal}} = \frac{\mathbf{v}_{\text{normal}}}{\|\mathbf{v}_{\text{normal}}\|} \tag{4.7}$$

4.5.2. Feature Selection

A straightforward approach would have been to provide the model with a set of baseline features, which include the following.

- **Heliostat ID**: A unique identifier for each heliostat.
- **Motor positions**: The angular positions of the heliostat motors.
- **Sun position**: The coordinates of the Sun in the local ENU coordinate system.

- **Aimpoint**: The recorded target position where the reflected sunlight is directed.

Although these features provide direct information about the system, relying solely on them may lead to the model overfitting to specific heliostat IDs and failing to generalize to new heliostats or field configurations. To mitigate this risk and improve the model's ability to generalize, additional engineered features were introduced, utilizing the formulas described earlier.

- **Heliostat normal vector**: The heliostat mirror orientation is determined using the normal vector, which is calculated based on two vectors: the Sun vector, derived from the heliostat position and the Sun position using Equation (4.1), and the aim vector, derived from the heliostat position and the aimpoint using Equation (4.1). These vectors are combined using Equation (4.6) to obtain the final normal vector representation.

- **Distance to aimpoint**: The magnitude of the aimpoint vector, using equation (4.2), quantifying the distance between the heliostat and its aimpoint.

- **Motor positions**: Retained from baseline characteristics, as they directly control the movement of the heliostat and provide critical information about its operational state.

- **Month**: Introducing the month as a feature allows the model to account for seasonal variations, such as changes in solar altitude and atmospheric conditions that affect light propagation.

All vectors were transformed into a directional spherical representation using equations (4.4) and (4.5), as this representation provides a more intuitive and consistent way to describe the directional relationships within the system.

The motivation behind selecting these engineered features was to provide the model with information that is more invariant to specific heliostat characteristics and field layouts. Instead of learning heliostat-specific behaviors, the model is encouraged to identify generalizable relationships that can be applied across different heliostats and operational scenarios.

By transforming raw position data into meaningful vectors and distances, the model gains a more holistic understanding of the dynamics of the system. Ultimately, the chosen feature set strikes a balance between providing essential system information and encouraging the model to develop a robust understanding that can generalize to new and unseen data effectively.

4.6. Model Architecture

To effectively model complex interactions within the heliostat system, a transformer architecture was selected for its ability to capture intricate dependencies between input features and provide a robust representation of high-dimensional

data. Unlike traditional neural networks such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), transformers are designed to model relationships across all input features simultaneously through the self-attention mechanism. This makes them particularly well-suited for tasks where spatial dependencies and feature interactions are crucial, such as heliostat field modeling.

The choice of a transformer is motivated by its ability to efficiently model dependencies, dynamically weighing the importance of different input features. This capability is essential for capturing intricate relationships between Sun position, heliostat configuration, and aimpoint with high precision. Additionally, transformers enable parallel processing, making them more scalable and computationally efficient than RNNs, which process inputs sequentially. Their robust feature representation also supports generalization across different heliostat configurations and environmental conditions, ensuring consistent performance.

Several key modifications were made to the standard Transformer architecture to better suit the specific requirements of the heliostat field, as visualized in Figure 4.3, including the following:

- **Modified Decoder:** The standard self-attention layer in the decoder was removed since the problem does not involve sequential dependencies. Instead of predicting the next step in a sequence, the decoder infers a missing feature for each input, capturing relationships across the entire input context.

- **Omission of Positional Encoding:** Standard transformers utilize positional encoding to retain order information, which is unnecessary in this case as input order carries no meaning. Removing positional encoding prevents introducing artificial sequential relationships.

- **Learnable Feature Embeddings:** Instead of using fixed feature representations, learnable embeddings were introduced for each input feature. These embeddings ensure a consistent representation of different input types, but the actual heliostat representation emerges within the encoder as it processes input relationships.

- **Removed Softmax layer:** The Softmax layer following the Decoder was removed to produce distinct output values rather than probability distributions.

These adaptations enhance the model's ability to process heterogeneous heliostat data effectively. While the learnable embeddings structure input features into a unified representation, the true heliostat representation is learned within the encoder itself, allowing the model to develop a deep understanding of heliostat-specific tracking relationships.

The model is designed to be trainable both in the forward and reverse direction, offering flexibility in predicting different aspects of the heliostat system. In the forward direction, the model is trained to predict the normal vector of the heliostat mirror based on the provided motor positions, distance, and month. This configuration enables the model to learn how the orientation of the heliostat is

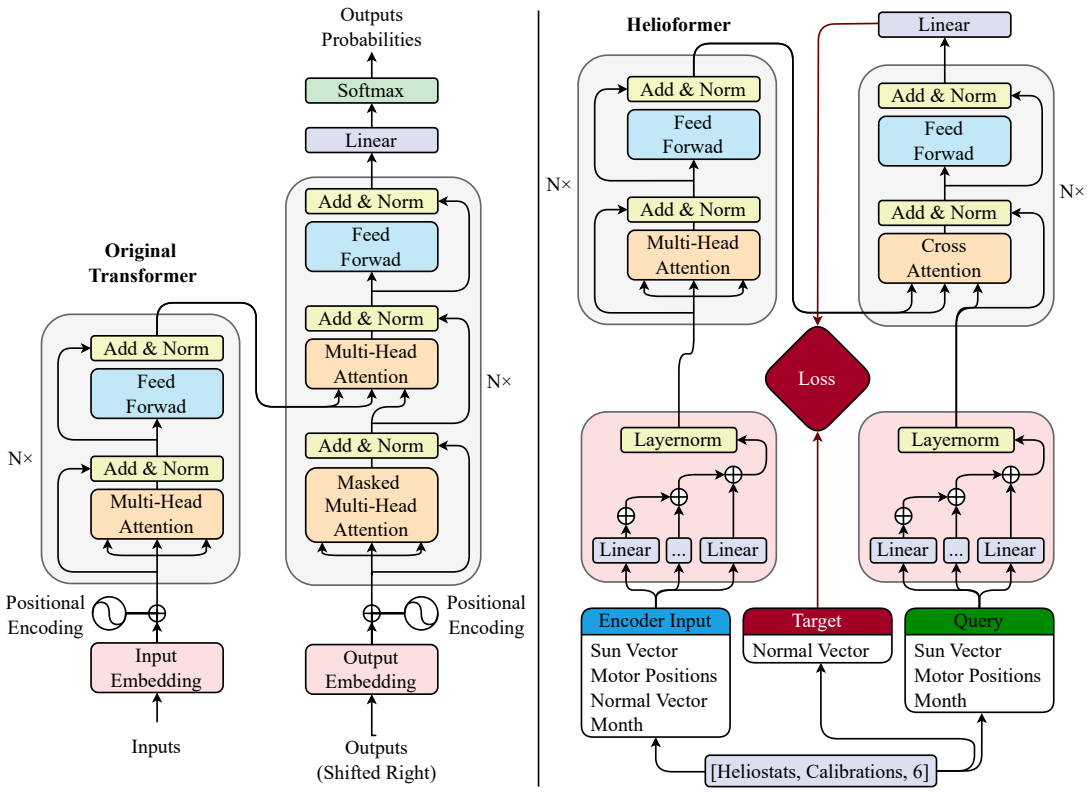

Figure 4.3.: **Standard Transformer architecture (left) compared to the modified *Helioformer* model (right), which has been adapted for heliostat calibration by integrating feature-wise embeddings and structural modifications to more effectively process calibration data.**

influenced by its control parameters and environmental conditions. In contrast, in the reverse direction, the model is trained to predict the motor positions required to achieve a desired normal vector, given the other input features. This bidirectional training approach enhances the versatility of the model, allowing it to be used to optimize the orientation of the heliostat and to evaluate the feasibility of the target aimpoints.

This model architecture enables the system to effectively predict aimpoints and motor positions while maintaining scalability and adaptability under various operational conditions.

4.7. Input Data Preparation

The preparation of input data is a critical step in ensuring that the neural network receives well-structured and normalized input, enabling effective learning and generalization across the heliostat field. The data preprocessing pipeline consists of several stages aimed at transforming raw calibration data into a structured tensor format suitable for model training.

The preprocessing pipeline begins with the normalization of the input features, which ensures that all the features are continuously scaled to improve convergence during training. For this purpose, Z-score standardization is applied to the

numerical features using the formula:

$$X_{\text{norm}} = \frac{X - \mu}{\sigma} \tag{4.8}$$

where X is the original feature value, μ is the mean and σ is the standard deviation. To account for the varying characteristics of the different input features, two normalization strategies were applied. The features Distance and Month were normalized globally over the entire dataset, meaning that their mean and standard deviation were calculated across all heliostats. Global normalization is appropriate for these features because they provide context about the heliostat's relative position and seasonal variations, which should remain consistent across the entire dataset. By normalizing these features globally, the model is better equipped to capture large-scale trends and relationships across different heliostat configurations.

In contrast, the Normal Vector and Motor Positions were normalized locally, meaning that their normalization parameters were computed separately for each heliostat. This approach preserves heliostat-specific characteristics and prevents the model from being influenced by global variations that may not be relevant to individual heliostat behavior. Local normalization ensures that the model can learn and adapt to the unique operating conditions of each heliostat.

Because the model outputs predictions in the normalized feature space, a denormalization step is necessary after the forward pass and before calculating the loss during training. This step ensures that predictions are brought back to their original scale so that the loss values can be interpreted meaningfully in relation to the actual system performance. The denormalization is performed using the inverse of the Z-score normalization formula:

$$X_{\text{original}} = X_{\text{norm}} \cdot \sigma + \mu \tag{4.9}$$

where X_{norm} is the model output in the normalized space, σ is the standard deviation of the original feature, and μ is the mean of the original feature. This step is crucial to ensure that the loss function operates within a meaningful and interpretable range, allowing a proper assessment of model performance and convergence.

Following the normalization process, the data is structured into tensors to facilitate efficient model training. The final input tensor structure used for the neural network follows the shape *[Heliostat, Calibrations, Features]*, where *Heliostat* represents the unique identifier for each heliostat in the field, *Calibrations* correspond to the number of recorded calibration instances per heliostat, and *Features* contain the processed input attributes, such as motor positions, aimpoint vectors, sun positions, and other engineered features.

In transformer-based models, the batch dimension typically corresponds to the number of independent training samples processed simultaneously, while the sequence dimension represents the length of the input context. In this setup, the *Heliostat* dimension is analogous to the batch dimension, grouping calibration data per heliostat. The *Calibrations* dimension serves as the sequence length,

providing historical calibration points for each heliostat. Finally, the *Features* dimension contains the input attributes, equivalent to the data dimension in typical deep learning models.

For example, if the training set size is 100 and 25 calibration instances per heliostat, each heliostat will have 4 batches. Given a total of 916 heliostats in the dataset, the resulting tensor shape would be *[4 * 916, 25, Features]*. This structure ensures that the model processes calibration sequences effectively while maintaining per-heliostat data grouping.

This arrangement aligns well with the requirements of the Transformer architecture, where inputs are typically processed in a batch-major format with clearly defined dimensions for different entities. Unlike traditional transformer inputs that often represent sequences in a temporal order, this structure represents calibration instances per heliostat, allowing the model to focus on spatial relationships and calibration-specific patterns rather than sequential dependencies. The organization of data in this manner ensures that each heliostat's calibration records are efficiently processed and fed into the model, optimizing the learning of feature interactions while preserving the distinct characteristics of each heliostat.

Algorithmus 2 Tensor Construction Algorithm

Require: DataFrame D, feature lists F, grouping column G
Ensure: Tensors T

```
Group the data by G
Initialize an empty list of tensors T
for each feature set f in F do
    Initialize an empty list of group tensors
    for each group g in grouped data do
        Extract unique selection values in g
        for each selection value do
            Extract corresponding feature values from g
            Convert extracted values to a tensor
            Add the tensor to the group tensor list
        end for
    end for
    Stack all group tensors into a single tensor
    Add the stacked tensor to T
end for
Return tuple of tensors T
```

The algorithm responsible for constructing the tensors takes as input the dataset, a list of selected feature sets, and a grouping column. It begins by grouping the dataset based on the specified grouping column, which organizes the data according to relevant categories such as heliostat IDs. For each set of selected features, the algorithm iterates through the grouped data, extracting unique selection values within each group. For each selection value, the corresponding feature values are extracted and converted into tensors. These tensors are accumulated and stacked to form a structured representation of the data. An detailed description can be found in Algorithm 2.

Using this algorithm, tensors can be easily created on the basis of the required features, providing flexibility to generate different inputs depending on the specific stage of model usage. For instance, during the training process, various tensors are needed, such as an encoder tensor to contain input features relevant to encoding the heliostat state, a decoder tensor for features required in the prediction task, a target tensor that holds the expected output values, and a denormalization tensor that stores the precomputed mean and standard deviation values required to revert predictions to their original scale. Additionally, the algorithm allows for the seamless creation of separate tensors for the training and validation sets. By running the algorithm with different datasets as input, it ensures a strict split between known and new data, preserving the integrity of the evaluation process.

By structuring the data in this manner, the algorithm facilitates efficient batch processing, memory management, and model optimization during training and evaluation.

4.8. Evaluation Metrics

To assess the performance of the proposed neural network model, appropriate evaluation metrics are employed to quantify the accuracy of the predictions. The Mean Squared Error (MSE) is used as the primary loss function during training, providing a straightforward measure of how closely the predicted values align with the ground truth. MSE is computed using the formula:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{4.10}$$

where $\hat{y}_i$ is the predicted value, y_i is the true value, and N is the total number of samples. However, MSE values are not easily interpretable in practical terms, making it necessary to introduce additional metrics that provide a more intuitive evaluation of model performance both during and after training. For this reason, the **angular offset** between the predicted and actual normal vectors is derived as a key metric, providing insight into how accurately the model predicts the orientations of the heliostats.

When predicting motor positions, MSE is also used as a loss function during training, ensuring a smooth optimization process. However, for better interpretability, the Mean Absolute Error (MAE) is used during the evaluation. MAE provides a clearer understanding of the average deviation of the motor positions in discrete step units, offering a tangible measure of the accuracy of the model. It is calculated using the formula:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|\hat{\alpha}_i - \alpha_i| \tag{4.11}$$

where $\hat{\alpha}_i$ and α_i represent the predicted and true motor angles for instance i, respectively.

The primary evaluation metric for normal vector prediction is the angular offset, which quantifies the deviation between the predicted normal vector $\hat{\mathbf{n}}_{\text{pred}}$ and the target normal vector $\hat{\mathbf{n}}_{\text{true}}$. This offset is calculated using the dot product of the two vectors, which provides the cosine of the angle $\Delta\theta$ between them. Since the normal vectors are provided in spherical coordinates (azimuth θ and elevation ϕ), they can be expressed in Cartesian coordinates as follows:

$$\mathbf{v} = \begin{bmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ \sin\phi \end{bmatrix} \tag{4.12}$$

Thus, for the actual and predicted normal vectors:

$$\hat{\mathbf{n}}_{\text{true}} = \begin{bmatrix} \cos\phi_{\text{true}}\cos\theta_{\text{true}} \\ \cos\phi_{\text{true}}\sin\theta_{\text{true}} \\ \sin\phi_{\text{true}} \end{bmatrix} \qquad \hat{\mathbf{n}}_{\text{pred}} = \begin{bmatrix} \cos\phi_{\text{pred}}\cos\theta_{\text{pred}} \\ \cos\phi_{\text{pred}}\sin\theta_{\text{pred}} \\ \sin\phi_{\text{pred}} \end{bmatrix} \tag{4.13}$$

The dot product of the two normal vectors is then computed as:

$$\begin{aligned} \hat{\mathbf{n}}_{\text{true}} \cdot \hat{\mathbf{n}}_{\text{pred}} = &(\cos\phi_{\text{true}}\cos\theta_{\text{true}})(\cos\phi_{\text{pred}}\cos\theta_{\text{pred}}) \\ &+ (\cos\phi_{\text{true}}\sin\theta_{\text{true}})(\cos\phi_{\text{pred}}\sin\theta_{\text{pred}}) \\ &+ (\sin\phi_{\text{true}})(\sin\phi_{\text{pred}}) \end{aligned} \tag{4.14}$$

Using the trigonometric formula

$$\cos A\cos B + \sin A\sin B = \cos(A - B), \tag{4.15}$$

the expression can be simplified to:

$$\cos(\Delta\theta) = \sin\phi_{\text{true}}\sin\phi_{\text{pred}} + \cos\phi_{\text{true}}\cos\phi_{\text{pred}}\cos(\theta_{\text{true}} - \theta_{\text{pred}}) \tag{4.16}$$

where $\Delta\theta$ represents the angular deviation between the predicted and target normal vectors.

Because the angle $\Delta\theta$ obtained from the above equation is in radians, it is converted to milliradians (mrad) for better comparability with existing results using the conversion factor.

$$\Delta\theta_{\text{mrad}} = \Delta\theta \times 1000 \tag{4.17}$$

This final metric provides an intuitive and interpretable measure of the model's accuracy in terms of heliostat orientation deviation.

When evaluating the prediction of the aim point, the law of reflection must be considered. According to the reflection law (as depicted in Figure 2.1), any deviation in the normal vector results in a doubled deviation of the reflected aimpoint. Consequently, to obtain the final deviation between the predicted and actual aimpoints, the calculated normal vector offset must be multiplied by a factor of two:

$$\Delta\text{Aimpoint} = 2 \times \Delta\theta_{\text{mrad}} \tag{4.18}$$

This adjustment ensures that the impact of angular deviation is properly reflected in terms of aimpoint positioning accuracy.

To compare the performance of the different models, a histogram of the offset values is generated, which represents the density distribution of the angular deviations. Several distribution-fitting methods, including Weibull, Gaussian kernel density estimation (KDE), and Rayleigh, are then applied to the data. The Gaussian KDE resulted in the best fit, as can be seen in Figure 4.4, and was selected for the final evaluation. Key statistical measures such as the mode, mean, and the highest density bin are extracted and can be used to assess model performance effectively.

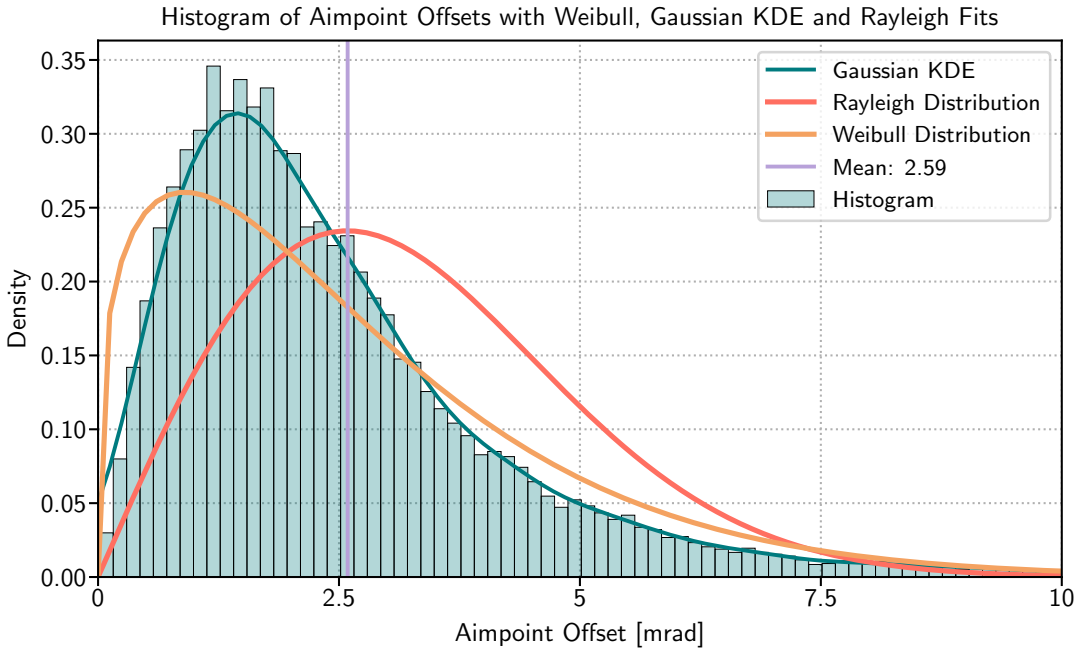

Figure 4.4.: **Histogram of aimpoint offsets with different fitted distributions, including a Weibull fit, a Gaussian KDE, and a Rayleigh fit.**

4.9. Training Process

The neural network training process is designed to optimize the performance of the model while ensuring generalization in different configurations of the heliostats. Several optimization strategies and validation techniques have been employed to achieve robust and reliable predictions.

The model is trained using the Rectified Adam (RAdam) optimizer [28], which has demonstrated strong performance for transformer architectures, particularly in handling the warm-up phase of learning rate scheduling. Unlike the standard Adam optimizer, RAdam incorporates an adaptive warm-up mechanism that stabilizes convergence during the initial training phase, mitigating potential issues

arising from sudden changes in the learning rate. A linearly decreasing learning rate schedule is applied with an initial warm-up phase, where the learning rate gradually increases to a peak value before linearly decaying over the course of training. This approach ensures a balanced transition from an exploration phase to stable convergence, allowing the model to efficiently refine its learned parameters.

To avoid overfitting and improve generalization, dropout and weight decay regularization techniques are incorporated into the training process. Dropout is applied within the Transformer layers to randomly deactivate a fraction of neurons during training, which helps the model learn more robust feature representations by reducing the dependency on specific neurons. Weight decay is utilized as an additional form of regularization by penalizing large weight magnitudes, thereby encouraging the model to favor simpler, more generalizable solutions. Both techniques are crucial in improving the model's ability to adapt to unseen data while avoiding overfitting into the training set.

The training process follows an iterative loop in which the model updates its parameters by minimizing the MSE loss function described above (4.10). The training loop consists of a forward pass, where the model generates predictions, followed by the computation of the loss, and an optimization step to update the model weights based on the calculated gradients.

To enhance the generalization capability of the model and prevent overfitting, rigorous data-shuffling strategies are applied at each training step. Both the heliostat order and the calibration sequence order within each batch are randomized before being fed into the model. The shuffling of the heliostats ensures that the model does not develop biases toward specific heliostat configurations, while shuffling of calibration sequences allows the model to learn across diverse operational conditions, improving its robustness and adaptability to varying scenarios.

A crucial aspect of the training process is the validation strategy, which serves to monitor the performance of the model and guide optimization decisions. A standard validation step is performed at regular intervals, where the model is evaluated on a held-out validation dataset to assess both the MSE loss and the angular offset metric (4.16). In addition to this conventional validation, a *mixed offset* evaluation is introduced to better simulate real-world operating conditions. In this evaluation, known calibrations from the training set are provided as input to the encoder, while unseen calibrations from the validation set are passed to the decoder. This approach mirrors practical deployment scenarios, where historical calibration data is used to predict new operational states.

By incorporating these optimization techniques, regularization methods, shuffling strategies, and validation methods, the training process ensures that the model achieves high accuracy while remaining adaptable to unseen heliostat configurations and changing environmental conditions.

4.10. Reverse Inference for Motor Position Prediction

To evaluate the performance of the model in predicting motor positions, a specialized inference strategy is employed to effectively address the inverse problem. Unlike aimpoint prediction, where model accuracy can be directly evaluated using the offset between predicted and actual normal vectors, motor position prediction requires a different approach to ensure meaningful evaluation and facilitate comparisons with other results. To achieve this, an optimization-based technique is introduced, leveraging the trained model to iteratively refine the input normal vector until it corresponds to the predicted motor positions.

In this reverse inference process, the trained Transformer model is first used to perform inference on the validation set, predicting the motor positions based on the given normal vectors. Following this, the model weights are frozen, and the optimization process begins. Instead of adjusting the model parameters, the optimizer is applied to iteratively adjust the input normal vector while keeping the model fixed. The objective of the optimization is to minimize the MAE between the predicted motor positions of the frozen model and the previously inferred motor positions. By refining the input normal vector, the goal is to find the vector that would result in the observed motor positions during inference. The structure of this inverse inference strategy is depicted in Figure 4.5.

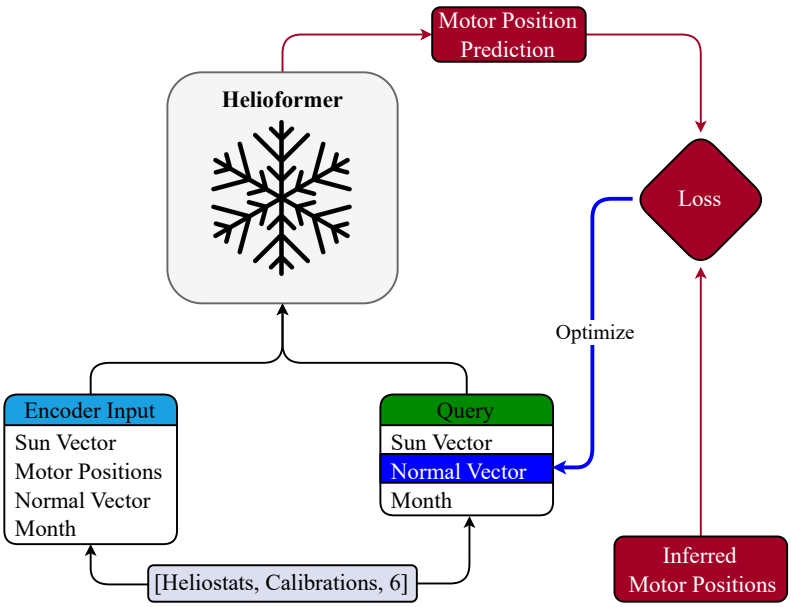

Figure 4.5.: **Illustration of the working principle of reverse inference for motor position prediction.**

Formally, given the predicted motor positions $\hat{\mathbf{m}}_{\text{inferred}}$ from the initial inference step, the optimization process seeks to find an input normal vector $\hat{\mathbf{n}}_{\text{optimized}}$ such that the predicted motor positions from the frozen model match the inferred values. The optimization objective is defined as:

$$\min_{\hat{\mathbf{n}}_{\text{optimized}}} \frac{1}{N} \sum_{i=1}^{N} |\hat{\mathbf{m}}_{\text{predicted},i} - \hat{\mathbf{m}}_{\text{inferred},i}| \tag{4.19}$$

where $\hat{\mathbf{m}}_{\text{predicted},i}$ represents the motor positions predicted by the frozen model using the optimized normal vector, and $\hat{\mathbf{m}}_{\text{inferred},i}$ are the motor positions obtained from the initial inference step.

The optimization process continues until the mean absolute error (MAE) falls below a predefined threshold, specifically less than one motor step. This stopping criterion ensures that the optimized normal vector provides a sufficiently accurate representation of the motor positions, aligned closely with the original predictions.

Once the optimization process converges, the offset between the optimized and original normal vectors is calculated to assess the accuracy of the model in the prediction of the position of the motor. This offset is calculated using the same angular offset metric used in the prediction of the aim point, as shown in (4.16).

By comparing the optimized normal vector with the original input, the effectiveness of the model in predicting motor positions can be quantitatively assessed and compared with other evaluation results. This approach provides an intuitive way to validate the inverse model by examining whether the predicted motor positions correspond to physically meaningful normal vectors.

This reverse inference approach offers a reliable method for evaluating the model's ability to accurately infer motor positions, ensuring that the predictions align with the expected physical properties of the heliostat system.

5. Implementation

This chapter presents the implementation of *Helioformer*, covering its software architecture, data processing pipeline, neural network training, and execution strategy. The software is designed to handle the entire heliostat calibration workflow, from dataset preparation to model inference, using a modular and scalable approach. The following chapter provides a detailed description of each module and its role within the system.

5.1. Overview of *Helioformer*

Helioformer is a Python-based software framework developed to handle the entire heliostat field calibration workflow, covering data preprocessing, model training, and evaluation. The software is designed with a modular structure to allow flexibility in adapting different processing techniques, model architectures, and evaluation strategies without requiring major changes to the overall system. An overview of the software structure and its key components is illustrated in Figure 5.1.

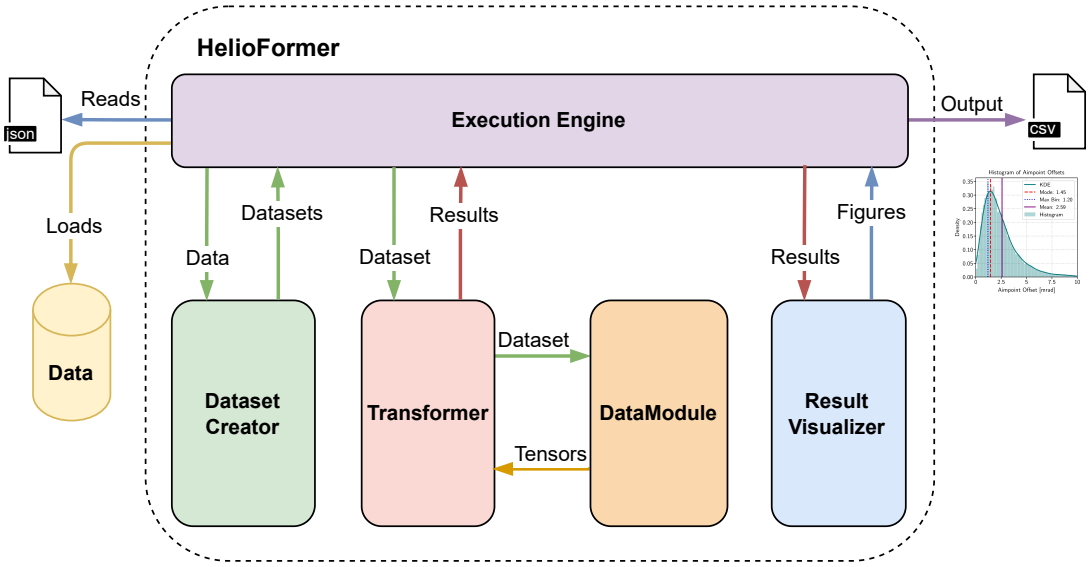

Figure 5.1.: **Overview of the *Helioformer* software architecture, illustrating the core components and data flow. The system comprises a Dataset Creator, Execution Engine, Transformer-based model, and visualization modules.**

The core functionality of *Helioformer* is divided into several key components, each responsible for a specific part of the workflow. These components can operate independently, but work together to ensure a seamless end-to-end process. The main components of the framework are:

- **Dataset_Creator**: Responsible for data pre-processing, feature engineering and splitting of the data into training and validation sets.

- **DataModule**: Handles the normalization and the transformation of processed datasets into tensors compatible with PyTorch, facilitating efficient data loading in preparation for the model.
- **DataModel**: Implements the neural network model and integrates the PyTorch Lightning functionality, enabling the training, validation, and checkpointing of structured models.
- **Execution Engine**: Orchestrates the entire workflow by coordinating the interaction between different modules, ensuring proper execution of data processing, model training, and evaluation.
- **DataVisualizer**: Generates plots and visualizations from model results to aid in the interpretation and evaluation of inferences.

The implementation of *Helioformer* follows a functional programming approach in its modular design. Each of the core components, except for the PyTorch Lightning integration, is implemented as separate Python files that define a set of functions. These functions primarily operate on Pandas Dataframes, process the data, and return new modified Dataframes rather than modifying them in place. This ensures immutability and makes the workflow more predictable and modular. The execution engine acts as a pipeline controller, sequentially calling these functions and passing the resulting data between different stages of the workflow.

In contrast, the PyTorch Lightning module within *Helioformer* follows a more object-oriented programming approach. The model, training routines, and evaluation logic are encapsulated in classes, as is standard in PyTorch-based deep learning frameworks. This structured approach ensures better reusability, scalability, and integration with advanced training functionalities such as checkpointing, logging, and distributed training.

The implementation of *Helioformer* relies on several key technologies:

- **Python** serves as the primary programming language for software development.
- **Pandas**, **NumPy** and **Scipy** are used for efficient data processing and numerical computations.
- **PyTorch** is employed to implement and train the neural network model, offering flexibility in experimenting with different architectures.
- **PyTorch Lightning**[1] provides a structured training process by managing tasks such as checkpointing, logging, and distributed training.
- **TensorBoard** is used to visualize key training metrics, including loss curves, learning rate schedules, and validation performance.

[1] `https://lightning.ai/`

The software runs on a shared high-performance computing cluster, meaning that not all resources may be fully available at all times due to concurrent usage by multiple users. The cluster specifications are as follows:

- **Processor:** AMD EPYC 7F32 with 8 cores.
- **Memory:** 252 GB of RAM.
- **GPU:** NVIDIA A6000 with CUDA support, featuring 48 GB of memory and 6912 CUDA cores.

Since resources are allocated dynamically based on demand, computational performance may vary depending on overall system load.

5.2. Dataset Preparation: The `Dataset_Creator` Module

The `Dataset_Creator` module in *Helioformer* is responsible for preparing the datasets used in training and evaluation. The module follows a structured pipeline that involves loading raw calibration data, performing feature engineering, filtering heliostats based on predefined selection criteria, and generating training batches.

5.2.1. Data Augmentation and Feature Engineering

The data preparation process begins with loading the raw calibration data using Pandas. The calibration data, stored as CSV files, are read into a DataFrame. The function `prepare_df()` is then executed to process the dataset by applying multiple feature engineering steps.

The first step in the data augmentation process is the unification of calibration target IDs. Since the original dataset contained multiple identifiers for each target, they are standardized into three categories (IDs 1, 2, and 3) to ensure consistency and ease of use. This is implemented in the function:

`unify_CalibrationTargetId(df)` Unifies calibration target IDs in a DataFrame.

Input: DataFrame containing calibration records with inconsistent target IDs.

Output: DataFrame with unified calibration target IDs (1, 2, or 3).

Afterwards the engineered features are added through the use of the following functions:

`add_sun_vector_to_df(df)` Computes vectors pointing from the sun to the heliostats, using formulas (4.4) and (4.5), and adds them to the DataFrame.

Input: DataFrame containing sun position coordinates in ENU format and the heliostat positions.

Output: Input DataFrame with added sun vector components (azimuth and elevation).

`add_aim_vector_to_df(df)` Computes the directional vectors pointing from the heliostats to the targets, using formulas (4.4) and (4.5), and adds them to the DataFrame.

Input: DataFrame containing heliostat position and target positions.

Output: Input DataFrame with added aim vector in azimuth-elevation representation.

`add_normal_vector_to_df(df)` Calculates the heliostat mirror's orientation using formula (4.6) and adds it to the DataFrame.

Input: DataFrame containing heliostat position, aimpoint, and sun vectors.

Output: Input DataFrame with added calculated normal vector components in both Cartesian and azimuth-elevation representation.

To ensure the correctness of the generated vectors, a visual sanity check is performed, allowing for manual inspection to verify that the computed values align with physical expectations:

`sanity_check_vectors(df, normal_scale, sun_scale, use_azimuth)` Performs a sanity check by visualizing computed vectors.

Input: DataFrame with computed vectors, optional scaling parameters, and a flag for azimuth-based calculations.

Output: A manipulatable 3D visualization of heliostat, normal, and sun vectors.

5.2.2. Heliostat Selection Process

Once feature engineering is completed, a subset of heliostats is selected to align with the experimental scenarios described in Section 4.4. For the first three scenarios, heliostats are selected based solely on their total number of calibrations. The primary criterion is ensuring that each heliostat has accumulated more than a specified minimum number of calibrations, making it suitable for training the neural network. This is handled by the following function:

`filter_by_calibs(df, minCount)` Filters heliostats based on the minimum required calibration count.

Input: DataFrame containing calibration records, minimum required calibration count.

Output: Filtered DataFrame including only heliostats meeting the calibration count requirement.

In contrast, the fourth scenario introduces an additional constraint: the selection is not just based on the total number of calibrations but also on the specific targets used for those calibrations. Instead of merely ensuring that a heliostat has accumulated a sufficient number of calibrations, this function selects heliostats that meet the calibration count requirement specifically for targets that are not the validation target. This ensures that the heliostats included in training have a sufficient number of calibrations where the designated training targets were used, making them suitable for training the neural network. To accommodate this refined selection process, the following function is used:

`filter_by_target_and_calibs(df, minTargetCount, valTargetId)`
Filters heliostats based on a minimum required calibration count while ensuring these calibrations were collected using two specific training targets.

Input: DataFrame with heliostat calibrations, required count of calibrations for the targets used in training, target used for validation.

Output: Filtered DataFrame including only heliostats with sufficient calibrations for the two targets to be used in training.

Each of these selection steps is performed using Pandas queries, ensuring efficient filtering while maintaining flexibility in dataset configuration.

5.2.3. Batch Generation Using K-Means and Equal-Size Clustering

After selecting the heliostats, the next step is generating structured training sequences. This is achieved using a two-step process: first, selecting a fixed number of calibration points per heliostat, followed by a custom clustering algorithm that assigns these points into structured batches. The batch generation process follows algorithm 1 and consists of the following steps:
Selecting 125 Evenly Distributed Calibrations Per Heliostat: Since some heliostats have significantly more than 125 calibrations, a subset of evenly spaced data points is selected using K-Means clustering. This ensures that the chosen calibrations are distributed across different solar conditions.

`select_n_evenly_distant_points_kmeans(df, n)` Selects a representative subset of calibration points for each heliostat.

Input: DataFrame with calibration records, desired number of calibrations per heliostat.

Output: DataFrame with 125 representative calibration points per heliostat.

Generating Batches Using Custom Equal-Size Clustering: The selected calibration points are then grouped into batches using a custom clustering algorithm that ensures that each batch is evenly distributed and contains the same number of calibrations.

`sample_equal_clusters(df, n, y)` Clusters selected calibration points into structured batches of equal size.

Input: DataFrame with selected calibration points, desired number of batches, and number of calibrations per batch.

Output: DataFrame with structured batch assignments. The batch of each calibration is now expressed in the column 'selected'.

Storage Format and File Structure: The processed dataset is saved as a CSV file, ensuring that each experimental configuration remains reproducible.

5.3. Data Handling: HeliostatDataModule

The `HeliostatDataModule` serves as the interface between the preprocessed dataset and the neural network, ensuring efficient data loading and management during training, validation, and inference. Implemented using PyTorch Lightning, the DataModule is responsible for dataset normalization, tensor construction, and dataloader configuration. This modular design enables seamless data handling and batch management, optimizing performance and memory utilization.

5.3.1. Data Preparation Pipeline

The dataset preparation process begins with transforming the processed dataset into a PyTorch-compatible format. The dataset, initially stored as a Pandas DataFrame, is converted into tensors to facilitate efficient model training. The `setup()` method in the `HeliostatDataModule` orchestrates this transformation by normalizing the dataset and splitting it into training and validation sets.

Normalization: Normalization is applied before conversion into tensors. Depending on the chosen normalization scope, either per-heliostat or global normalization is performed:

Per-heliostat normalization Applied to angular features such as azimuth and elevation angles, as well as motor positions. This ensures that the feature distribution of each heliostat is independent of other heliostats.

Global normalization Applied to features such as distance and month, which retain global significance across the dataset.

The normalization process is handled using Z-score standardization, as described in Section 4.7, using equation (4.8). The denormalization parameters are also stored in the dataset, allowing for denormalization after inference with equation 4.9.

Dataset Splitting: After normalization, the dataset is split into training and validation sets based on the pre-selected batches from the `Dataset_creator`. The validation set consists of the first batch, where `selected == 0`, while all other calibrations form the training set.

Tensor Construction: The tensors are then constructed using the function `_make_tensors()`, which groups the dataset by `HeliostatId` and converts feature sets into PyTorch tensors, following the algorithm 2 described in Section 4.7. The tensorized dataset is structured as follows:

$$[\text{Heliostat}, \text{Calibrations}, \text{Features}] \tag{5.1}$$

Mixed Validation Setup: During the validation setup, an additional tensor is created for a mixed validation scenario. This mixed validation set consists of encoder inputs from the training set and decoder inputs from the validation set, simulating real-world conditions where known calibrations are used to predict new ones.

5.3.2. PyTorch Dataset and DataLoader Implementation

The `HeliostatDataModule` is designed to efficiently handle data batching and shuffling. It achieves this by first grouping the tensors within a PyTorch `TensorDataset` class and then wrapping them in a PyTorch `DataLoader` to facilitate structured batch processing. Since PyTorch's DataLoader only applies shuffling along the first dimension, heliostats are randomly sampled between batches while maintaining the integrity of their calibration sequences.

Dataset Structure Three separate datasets are created to handle different stages of model execution. Each dataset is constructed with a different set of features, tailored for its specific purpose:

Training dataset Contains tensors for encoder input, decoder input, targets, and denormalization parameters.

Validation dataset Includes the same structure as the training dataset but also incorporates a mixed validation encoder input tensor, where known calibrations from the training set are used to predict new ones.

Prediction dataset Consists of encoder input, query input, Calibration IDs to match predictions with the original data after inference, and denormalization parameters.

Efficiency Considerations By pre-splitting the datasets into their respective tensors before model execution, a slightly higher GPU memory footprint is required. However, this approach significantly reduces computational overhead during training and inference, as no additional tensor manipulations are necessary within each training step, thereby improving overall processing efficiency.

DataLoader Integration The datasets are wrapped in subsequent DataLoaders, ensuring efficient batch processing. Since the `HeliostatDataModule` inherits from PyTorch Lightning's `LightningDataModule`, it provides standardized functions for data retrieval:

`train_dataloader()` Returns batches from the training dataset.

`val_dataloader()` Loads the validation dataset.

`predict_dataloader()` Handles inference data.

These functions seamlessly integrate with the model during execution, automatically supplying the required data at each stage of training and evaluation.

5.4. The *Helioformer* Model: Implementation and Architecture

To predict heliostat behaviors and optimize calibration, a custom Transformer model was implemented using PyTorch, with PyTorch Lightning managing training, validation, and inference. This model, referred to as *Helioformer* model, adapts the standard Transformer architecture to handle structured heliostat calibration data, where traditional sequence-based assumptions do not apply. The implementation consists of two core components:

- **The Transformer Model (Implemented in PyTorch)**: Defines the architecture, including the encoder, decoder, and learnable feature embeddings.
- **The Lightning Module (Implemented in PyTorch Lightning)**: Wraps the Transformer in a structured training framework, handling data loading, optimization, loss computation, and logging.

This section provides a detailed overview of the model implementation, focusing on architectural choices, modifications, and how PyTorch Lightning facilitates streamlined training.

5.4.1. Implementation of the Transformer Model in PyTorch

The *Helioformer* model follows a encoder-decoder structure, implemented using PyTorch's `nn.Module` class. Each component is modularized to ensure reusability and flexibility during experimentation. The model processes heliostat calibration sequences and makes predictions based on previously observed calibration data.

The encoder processes calibration data, capturing meaningful relationships between features. It consists of:

- A **learnable feature embedding layer**, where each input feature is independently projected into a higher-dimensional space.
- Multiple encoder layers, each containing:
  - **Multi-head self-attention** to model dependencies between different calibration points.

    - **Feedforward layers** for **nonlinear transformations** of feature representations.
    - **Layer normalization and dropout** to stabilize training and prevent overfitting.

- A **final normalization layer**, refining feature representations before passing them to the decoder.

The forward pass of the encoder follows these steps:

1. **Feature embeddings** are applied to map raw inputs into a continuous latent space.
2. The input is **transposed** to match the expected format of the Transformer [*sequence*, *batch*, *features*].
3. The stacked encoder layers process the input through self-attention and feedforward operations.
4. A **final normalization step** ensures the output is stable and ready for the decoder.

The encoder outputs a latent representation of the calibration sequence, containing rich feature dependencies that the decoder will use for prediction.

The decoder generates predictions by attending to the encoder's output while processing query inputs (e.g., new calibration data). Unlike a traditional Transformer decoder, the first self-attention layer is removed, as query inputs do not form a sequence and are independent of each other.

The decoder consists of:

- The **same learnable feature embeddings** as the encoder, applied to query inputs.
- **Cross-attention layers** that attend to encoder outputs, allowing calibration knowledge to guide predictions.
- **Feedforward transformations** to refine the extracted features.
- An **output projection layer**, mapping the decoder's final representation to the **target space** (either normal vectors or motor positions).

The forward pass of the decoder follows these steps:

1. Query features are embedded in the same way as calibration features.
2. The query embeddings interact with the encoder output through cross-attention.
3. The processed features are transformed by feedforward layers to extract meaningful patterns.

4. The final output projection layer maps hidden states to either normal vector predictions or motor positions.

The encoder and decoder are then combined into a unified PyTorch model, referred to as the *Helioformer* model. This model integrates both components within a single architecture, ensuring seamless interaction between the calibration data processed by the encoder and the query inputs handled by the decoder.

During the forward pass, the model first processes the calibration inputs through the encoder, generating a structured latent representation of the heliostat's calibration state. This encoded information is then passed to the decoder, which receives the query inputs and utilizes the encoder's output to generate predictions.

5.4.2. Feature Embedding via Learnable Linear Projections

Instead of positional encodings, each input feature is embedded separately using a dedicated linear transformation, as shown in Figure 4.3. This is implemented using PyTorch's `nn.Linear` layers, stored in a `ModuleList`. The embedding process ensures the following:

Feature-Specific Transformations Each feature has its own dedicated transformation layer, ensuring individualized processing.

Independence of Features Features remain independent, avoiding artificial order dependencies that could otherwise impact model performance.

Increased Model Expressiveness Mapping input features to a higher-dimensional space enhances the network's ability to capture complex relationships.

The embedding function applies the transformation through the following steps:

Feature Extraction Each feature is extracted individually from the input tensor.

Learnable Transformation A corresponding transformation is applied to each feature using its dedicated linear layer.

Final Input Representation The transformed values are summed to construct the final representation used as input to the model.

5.4.3. Implementation in PyTorch Lightning

To streamline training, validation, and inference, the *Helioformer* model is implemented within the PyTorch Lightning framework. PyTorch Lightning provides a structured way to organize deep learning workflows, automating aspects such as data loading, checkpointing, and logging while keeping the core model

implementation in native PyTorch. It achieves this by defining standard training, validation, and inference loops through the `training_step()`, `validation_step()`, and `predict_step()` functions within the `LightningModule`, which directly correspond to the data provisioning methods (`train_dataloader()`, `val_dataloader()`, and `predict_dataloader()`) of the `HeliostatDataModule`. This integration ensures seamless interaction between data loading and model execution, enabling efficient batch processing and reducing boilerplate code.

The *Helioformer* model is encapsulated in a PyTorch Lightning module called `HelioformerLightning`, which extends `LightningModule`. This module is responsible for defining the model architecture, training logic, and optimization strategy in a structured format. Within its initialization, it sets up the encoder-decoder architecture, optimizer, loss function, and learning rate scheduler, ensuring that all key components are defined before training begins. All initialization parameters, e.g. the normalization scope and the target feature, are saved via the checkpointing feature. This way the trained model can later be loaded for inference without the need to keep track of the initialization parameters that were used during training. Additionally, TensorBoard is integrated to monitor training progress, visualize loss curves, and track evaluation metrics in real time. This modular design enhances reusability and makes it easier to modify hyperparameters or experiment with different training strategies.

I. Training Step

During training, the model optimizes the Mean Squared Error (MSE) loss using the `training_step()` function. Each training batch consists of encoded calibration data, query inputs, and corresponding target values. The training step follows these steps:

1. **Batch Preparation**: The dataset is shuffled in the second dimension at the beginning of each epoch, ensuring the model does not rely on specific data orders. Each batch is retrieved from the `train_dataloader()`, which shuffles the dataset in the first dimension.

2. **Forward Pass**: The encoder processes the calibration input, generating a latent representation. The decoder then utilizes this representation along with the query input to predict the target values.

3. **Denormalization**: If the model operates on data that is normalized per heliostat, predictions and targets are denormalized before loss computation.

4. **Loss Computation**: The MSE loss is calculated between predicted and actual target values, and backpropagation updates the model parameters.

5. **Logging and Optimization**: The loss value is recorded, and the gradients are updated via the optimizer.

This iterative process continues until convergence is reached or an early stopping criterion is met. Through the automatic checkpointing of Pytorch Lightning the weights of the best epoch are saved.

II. Validation Step

Every 5 epochs, a validation step is performed. This step follows a similar structure as the training step but differs in several key aspects:

1. **Batch Retrieval**: Validation data is loaded using the `val_dataloader()` without shuffling to maintain consistency across epochs.
2. **Model Inference**: The encoder processes calibration sequences, and the decoder predicts target values.
3. **Denormalization**: If the model operates on data that is normalized per heliostat, predictions and targets are denormalized before loss computation.
4. **Loss Computation**: The Mean Absolute Error (MAE) is used as an additional metric to provide an interpretable measure of prediction accuracy. Additionally, the Angular Deviation Loss is computed using equation (4.16) to evaluate the difference between predicted and true normal vectors.
5. **Mixed Offset Evaluation**: To simulate real-world deployment, an additional validation step is conducted where encoder inputs are taken from the training set while decoder inputs remain from the validation set. This scenario tests the model's ability to generalize using previously known calibrations.

III. Prediction Step

The prediction step is used to generate predctions for the different scenarios. The process consists of:

1. **Batch Processing**: The dataset is loaded using `predict_dataloader()`.
2. **Forward Pass**: The encoder generates representations from known calibration data, and the decoder predicts new target values.
3. **Output Denormalization**: The final predicted values are converted back to their original scale.
4. **Return**: The predicted results are returned to the Execution Engine for further evaluation or visualization.

IV. Optimization Strategy

The optimization process is designed to ensure stable convergence and improved generalization. The model is trained using the RAdam optimizer [28], a variant of the Adam optimizer designed to enhance training stability. Unlike traditional Adam, which can suffer from large variance in the early training stages, RAdam incorporates a rectification term that dynamically adjusts the learning rate warm-up. This ensures smoother convergence and better performance in small batch training scenarios.

The optimizer is configured with the following.

- **Learning rate:** 2×10^{-4} as the initial value.
- **Weight decay:** A penalty of 10^{-3} to prevent overfitting by discouraging large weight magnitudes.
- **Learning Rate Scheduling:** A **linear decay schedule** with an initial warmup phase (5% of total epochs), where the learning rate gradually increases before decreasing linearly throughout training. This is handled using a custom learning rate scheduler.

Furthermore, **dropout layers** are incorporated in both the encoder and the decoder to prevent over-reliance on specific calibration patterns, enhancing the model's ability to generalize to unseen heliostats. A dropout value of 10% is used.

5.4.4. Reverse Optimization for Motor Position Prediction

While the *Helioformer* model can be trained to predict motor positions directly, evaluating the accuracy of these predictions requires a different approach. Since the relationship between motor positions and normal vectors follows a well-defined physical transformation, a reverse optimization procedure is applied to infer the normal vectors corresponding to the predicted motor positions. The reverse optimization implementation follows these steps:

1. The trained Transformer model is loaded using PyTorch Lightning's checkpointing feature. The `LightningTransformer` class is restored with all saved hyperparameters. The model's encoder and decoder components are extracted and explicitly set to evaluation mode using `.eval()`, ensuring that regularization techniques such as dropout are not applied and that their parameters remain unchanged during the backward pass.

2. The normal vectors, which serve as input to the decoder, are initialized as learnable parameters using PyTorch's `nn.Parameter` function. These parameters are detached from the computation graph and reintroduced as trainable tensors, allowing them to be updated independently during optimization.

3. The optimization process is defined within a new PyTorch Lightning module, `LightningInputOptimizer`. This module takes the frozen encoder and decoder as inputs and treats the normal vectors as trainable parameters. During the forward pass, the encoder processes the calibration data, while the decoder takes the learnable normal vector parameters and predicts motor positions.

4. The optimizer, implemented using the Rectified Adam (RAdam) algorithm, iteratively updates the normal vectors to minimize the loss between the predicted and target motor positions. The loss function used is Mean Absolute Error (MAE), which provides a direct interpretation in motor steps. Learning rate scheduling is handled using `ReduceLROnPlateau`, which dynamically reduces the learning rate if no improvement is detected.

5. The optimization loop runs until the loss falls below a predefined threshold of one motor step.

6. Once optimization converges, the optimized normal vectors are extracted, detached from the computation graph, and returned alongside their corresponding calibration IDs. The final deviation between the optimized normal vectors and the original normal vectors is computed using the angular deviation metric (see Equation (4.16)). These optimized normal vectors are then stored and later integrated into the prediction evaluation pipeline.

5.5. Execution Engine

The training and execution pipeline of *Helioformer* is designed to be modular and automated, ensuring that training, validation, and inference can be conducted with minimal manual intervention. The execution process is controlled through a JSON-based configuration system, where users specify key parameters such as model hyperparameters, training settings, and dataset details. The program is structured to be run from the command line, where the user provides the JSON configuration file along with the desired GPU ID, allowing flexible execution across different hardware setups.

5.5.1. Configuration Management

The training and execution process is controlled through a structured JSON configuration file. This file defines essential parameters such as:

- Learning rate, batch size, and number of epochs.
- Model architecture settings, such as Transformer dimensions, number of attention heads, and layer counts.
- Data-related options, including the prediction type (motor positions or normal vectors) and dataset filtering criteria.
- Checkpoint settings for resuming training or loading pre-trained models.

By leveraging this configuration system, users can easily modify hyperparameters and re-run experiments without modifying the core codebase. It is also possible to configure multiple trainings in the configuration file. These runs will then be executed one after another.

5.5.2. Training Execution

The execution engine follows a structured pipeline to train the *Helioformer* model. After receiving the dataset and the user-defined configuration file, the program automatically initiates the training process. The key steps include:

1. Loading the user-provided training data, which contains calibration records, heliostat IDs, sun positions, aimpoint positions, heliostat positions, and timestamps.
2. Generating structured datasets as described in Section 4.7, ensuring a proper split into training and validation sets.
3. Initializing the PyTorch Lightning `LightningTrainer` with the specified model architecture and hyperparameters.
4. Conducting training with automated checkpointing, storing the best-performing model based on validation loss.
5. Logging key training metrics, such as loss curves and learning rate scheduling, using TensorBoard.

This automated workflow ensures that all key training artifacts, including hyperparameters, validation performance, and trained model weights, are stored efficiently, allowing easy model reuse.

5.5.3. Prediction and Model Evaluation

Once training is complete, the trained model can be used for inference on new datasets. Since all hyperparameters and model configurations are stored within PyTorch Lightning's checkpointing system, the user only needs to provide the trained model checkpoint and a new dataset for evaluation. The inference process follows these steps:

1. The new dataset is loaded and preprocessed using the same pipeline as during training. An already prepared Dataset can also be loaded if desired.
2. The trained model is applied to generate predictions.
3. The predicted values (motor positions or normal vectors) are added back to the dataset using the Calibration ID to match them with the original input data.
4. If motor positions were predicted, the reverse optimization procedure (as described in Section 5.4.4) is applied to compute the corresponding normal vectors.
5. The offsets between the predictions and ground truth values are computed to evaluate model accuracy.
6. The results are saved to disk for further analysis.
7. The results are visualized via the ResultVisualizer.

By structuring the training and inference pipelines in this manner, *Helioformer* ensures an efficient, scalable, and reproducible workflow for training and evaluating Transformer-based heliostat calibration models.

5.6. Result Visualization and Analysis

The `ResultVisualizer` module in *Helioformer* is responsible for analyzing and visualizing the model's predictions. It provides graphical representations of prediction performance, aiding in the interpretation and evaluation of results. This module primarily uses Matplotlib, NumPy, and SciPy to generate plots that summarize model accuracy and error distributions.

5.6.1. Prediction Error Analysis

To assess the quality of the model's predictions, the `ResultVisualizer` generates histograms of the prediction offsets. These histograms illustrate the distribution of errors across all heliostats in the dataset, offering insights into the model's performance.

A Gaussian Kernel Density Estimation (KDE) fit is applied using SciPy to estimate the probability density function of the errors. This helps in identifying patterns or biases in the predictions, as described in Section 4.8.

5.6.2. Batch Execution and Summary Plots

If multiple configurations are defined in the JSON configuration file, they can be executed sequentially. In this case, after processing the final configuration, the `ResultVisualizer` generates a combined summary plot that consolidates the results of all configurations. This allows for an aggregated comparison of different model runs.

5.6.3. Heliostat Field Visualization

In addition to statistical error analysis, the `ResultVisualizer` generates a Heliostat Field Plot to provide a spatial representation of prediction accuracy. In this plot, each heliostat is placed at its actual position, and the color of each marker corresponds to the mean prediction offset for that heliostat. This visualization allows for an intuitive evaluation of model performance across different regions of the heliostat field.

6. Evaluation and Results

This chapter evaluates the performance of the proposed *Helioformer* model in predicting aimpoints and motor positions for heliostat calibration. The analysis focuses on accuracy, generalization ability, and computational efficiency, comparing the model to traditional rigid body model calibration methods.

The evaluation is structured into different scenarios. First, the model's accuracy is assessed on known heliostats to establish a baseline. Then, its generalization ability is tested on unseen heliostats, with and without retraining. Further experiments examine cross-target generalization, the impact of training set size, and encoder input size. Finally, computational efficiency is analyzed in terms of memory usage, training time, and scalability.

Both numerical metrics and visualizations are used to assess performance. The primary accuracy metric is the angular offset, computed using kernel density estimation (KDE). Scatter plots and histograms provide further insight into prediction quality. The results in this chapter highlight the model's strengths and limitations in real world heliostat calibration.

6.1. Overview of Evaluation Criteria

The evaluation of the proposed *Helioformer* model is structured to assess its accuracy, generalization capability, and computational efficiency. Since the primary objective is to improve aimpoint prediction and motor position estimation for heliostat calibration, the evaluation is centered around both numerical metrics and qualitative visualizations that reflect the model's effectiveness in these tasks.

A key focus is the angular offset, which measures how accurately the model predicts the orientation of the heliostat. During training, the mean angular offset across all heliostats is computed to monitor performance trends. After training, the final accuracy is determined by applying Gaussian kernel density estimation (KDE) to the offset distribution and extracting its mode. In addition, scatter plots comparing predicted and actual aimpoints, as well as histograms of the angular offset distribution, provide qualitative insights into the model's effectiveness. Furthermore, tracking accuracy is analyzed as a function of the distance between heliostats and the tower to examine spatial performance variations.

To better simulate real-world deployment conditions, the model is evaluated under two different conditions. First, accuracy is assessed using data from the training set as input to the encoder, representing a realistic operational scenario where past calibration data is available. Second, a worst case accuracy evaluation is conducted using unseen data for the encoder.

Additionally, experiments are conducted to analyze the influence of training set size and encoder batch size on model performance.

6.1.1. Prediction Accuracy Metrics

To quantify the model's predictive performance, the mean squared error (MSE) is used as the primary loss function during training, ensuring stable optimization by penalizing large deviations. Additionally, the mean absolute error (MAE) provides an interpretable measure of deviation in motor step units. However, the most relevant metric for evaluating the model's accuracy is the angular offset between the predicted and actual normal vectors of the heliostat mirror, as defined in Equation (4.16).

Since any deviation in the mirror's orientation results in double the deviation of the reflected aimpoint, the final aimpoint offset is computed using Equation (4.17). During training, the mean angular offset across all heliostats is recorded to provide an overview of the model's learning progress. For final evaluation, the offset distribution of all test heliostats is analyzed using KDE, and the mode of this distribution is reported as the final accuracy of the trained model on the corresponding dataset. This method ensures that the reported accuracy reflects the most common deviation rather than being influenced by extreme outliers.

To further evaluate the model's real-world performance, a scatter plot is used to visualize the predicted and actual aimpoints. This provides an intuitive representation of how well the model aligns with the true calibration targets and whether systematic deviations occur. Additionally, a histogram of the angular offset distribution is used to analyze the spread of errors, ensuring that model performance is not overly dependent on specific heliostat positions.

Beyond these primary accuracy evaluations, an additional analysis is conducted to examine how tracking accuracy varies with the distance of the heliostat from the tower. Since heliostats located further from the tower experience different optical conditions and tracking constraints, evaluating accuracy as a function of distance provides valuable insights into the model's robustness across the heliostat field.

6.1.2. Generalization Capability

A key requirement of the model is its ability to generalize between different heliostats in the heliostat field. To assess this, performance is evaluated across different experimental conditions, as described in Section 4.4. First, the model is trained and tested on the same set of known heliostats to establish baseline performance. Then, it is tested on a different set of unseen heliostats to assess its adaptability. In another experiment, the model is initially trained on known heliostats and later retrained for a limited number of epochs with newly added heliostats to measure adaptability. Lastly, the model is trained on data from two calibration targets and tested on a third target to evaluate its ability to generalize across different spatial configurations.

6.1.3. Computational Efficiency and Scalability

Since large-scale deployment in a solar tower plant requires both accuracy and efficiency, computational performance is also analyzed. Training time is measured to determine how efficiently the model optimizes on a large dataset. Inference speed is evaluated to ensure that real time deployment is feasible. Additionally, memory utilization is assessed to determine whether the model can scale to larger heliostat fields without excessive computational cost. These factors contribute to the feasibility of integrating the model into real-world solar power plants for continuous calibration.

6.2. Scenario 1: Known Heliostat Evaluation

This section presents the results when the model is trained and tested on the same set of known heliostats. Since the model has already seen this data during training, this scenario serves as a baseline to assess how well the model fits the training data. The training dataset consists of 100 calibration points per heliostat, with a total of 916 heliostats. Figure 4.1 illustrates the spatial distribution of the heliostats used in training.

The model was trained for 100,000 epochs using a batch size of 4×916 heliostats, meaning that each epoch consisted of a single training step. During training, the encoder received 20 calibration points as input to provide context, while the decoder was given 5 query points to predict the corresponding aimpoints or motor positions.

6.2.1. Prediction of Normal Vectors

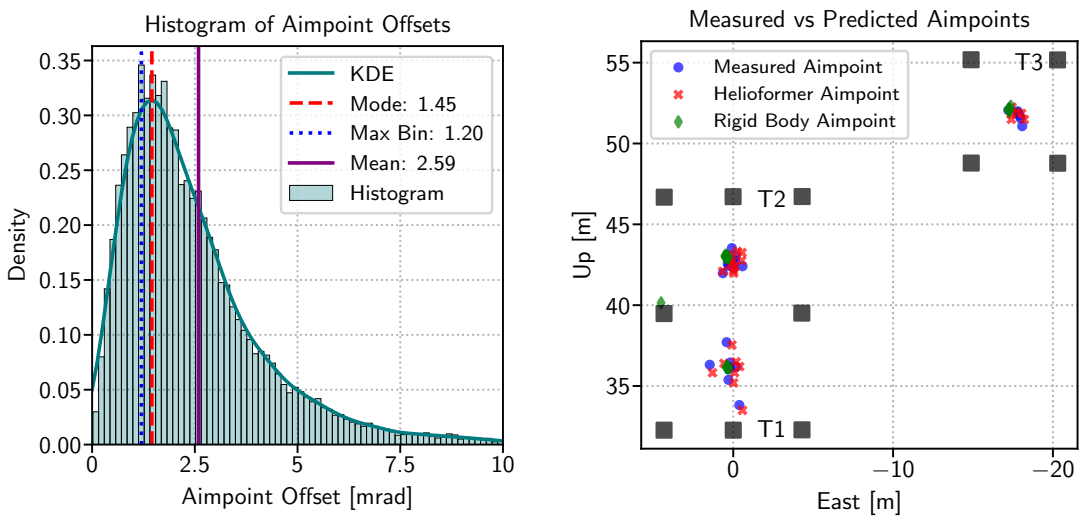

Figure 6.1.: **Histogram of angular offsets with a KDE for normal vector prediction in Scenario 1, achieving an accuracy of 1.45 mrad (left). Scatter plot of aimpoint predictions using normal vector prediction for an exemplary heliostat in Scenario 1 (right).**

The accuracy of the model in predicting heliostat normal vectors is evaluated using the angular offset metric. The histogram in Figure 6.1 displays the offset distribution, with the KDE mode showing an achieved accuracy of 1.45 mrad when using known encoder inputs. In a worst-case scenario, where the encoder is provided with completely unseen data, the accuracy is slightly reduced to 1.72 mrad. This demonstrates that the model maintains relatively stable accuracy even under less favorable conditions.

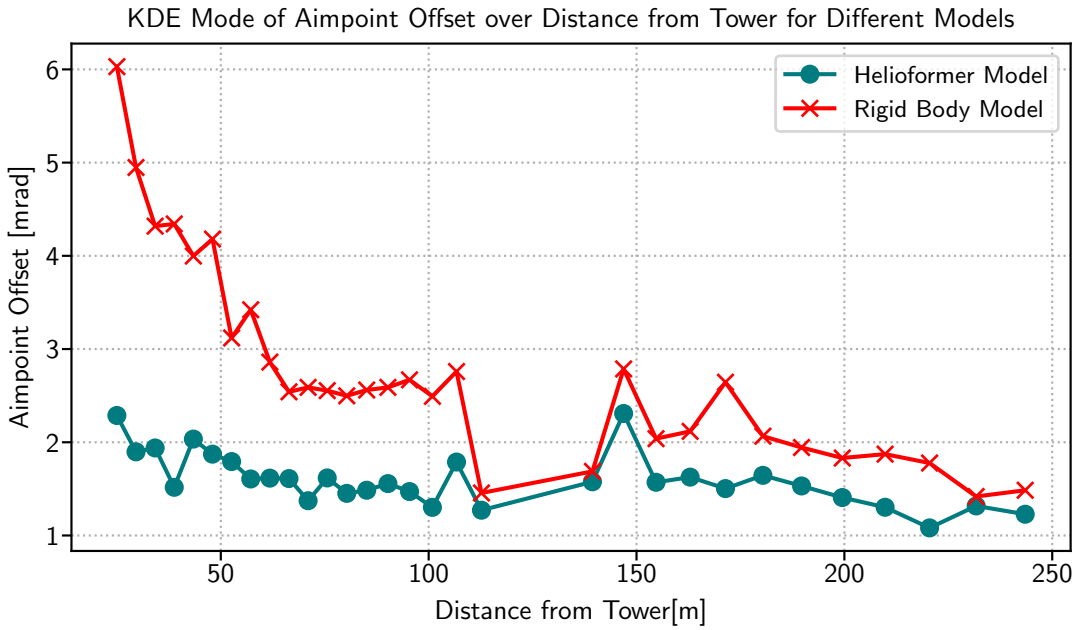

Figure 6.2.: **Aimpoint offset as a function of heliostat distance from the tower, comparing the *Helioformer* model with the rigid body model.**

To further assess prediction accuracy, Figure 6.1 also presents a scatter plot of predicted aimpoints for a heliostat that achieves the KDE mode accuracy. The results confirm that the predicted aimpoints closely align with actual calibration target locations, validating the effectiveness of the model.

Additionally, the model's accuracy is evaluated as a function of heliostat distance from the tower. Figure 6.2 compares the *Helioformer* model to a traditional rigid body model, demonstrating that the *Helioformer* consistently outperforms the rigid body model across all distances, especially for distances smaller than 100m. This result highlights the robustness of the neural network approach in predicting heliostat orientations more accurately than traditional methods. It also shows that the accuracy tends to decrease as the heliostat moves closer to the tower, suggesting that near-field calibration presents additional challenges compared to distant heliostats.

Additionally, to provide a more detailed spatial representation of the model's performance across the heliostat field, Figure 6.3 visualizes the KDE mode of the offset for each heliostat using a color gradient. This method allows for an intuitive assessment of how well the model performs across different heliostat positions. The majority of heliostats exhibit an offset below 2 mrad, indicating consistent accuracy across the field. However, a few outliers with higher offsets

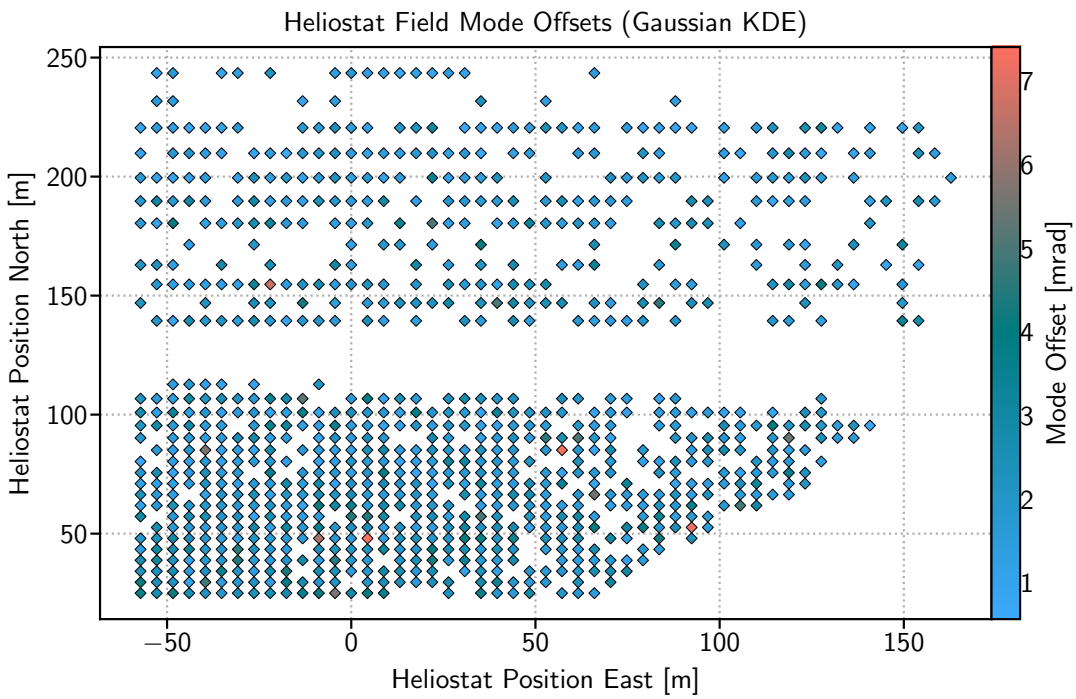

Figure 6.3.: **Aimpoint offset of the Helioformer model for all heliostats in the heliostat field based on normal vector predictions in Scenario 1. The color scale represents the mode offset in milliradians.**

are present, suggesting that certain heliostats experience greater aimpoint offsets. Notably, no clear spatial patterns emerge, meaning that the distribution of errors does not strongly correlate with specific regions in the heliostat field. This could indicate that individual calibration variations or mechanical deviations might be contributing to these inconsistencies rather than a systematic location dependent trend.

6.2.2. Prediction of Motor Positions

In addition to predicting heliostat normal vectors, the model was trained to predict motor positions directly. The accuracy of this approach is evaluated using the angular offset metric, achieving a KDE mode accuracy of 1.31 mrad, as shown in Figure 6.4. This accuracy was obtained using the optimization process described in Section 5.4.4, where the predicted motor positions were iteratively refined to match the desired aimpoint locations. The optimization ensures that the model provides motor settings that lead to physically meaningful heliostat orientations.

Figure 6.4 presents a scatter plot of predicted aimpoints using motor position-based predictions. The results confirm that the model can successfully infer the required motor positions to achieve accurate aimpoint targeting. This capability makes it a viable alternative calibration approach and opens up the possibility of using the model for direct heliostat control.

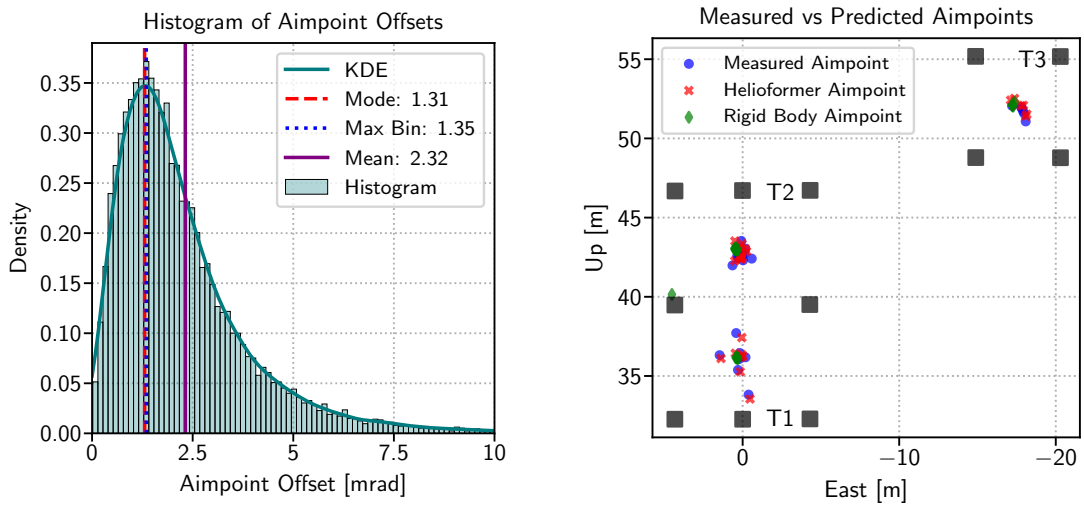

Figure 6.4.: **Histogram of angular offsets with a KDE for motor position prediction in Scenario 1, achieving 1.31 mrad accuracy (left). Scatter plot of aimpoint predictions using motor position prediction for an exemplary heliostat in Scenario 1 (right).**

6.3. Scenarios 2 and 3: Generalization to New Heliostats and Retraining

This section evaluates the model's ability to generalize to unseen heliostats and how retraining with a limited amount of additional data improves performance. In Scenario 2, the model trained in Scenario 1 was used for inference on heliostats that were not included in the training set. The heliostats used for this evaluation are shown in Figure 4.1. This experiment assesses whether the model can infer aimpoints and motor positions for heliostats it has never seen before, relying only on its learned representations.

In Scenario 3, the model is further retrained using calibration data from the previously unseen heliostats to assess whether additional training can restore performance to levels comparable to those observed in Scenario 1. The retraining process is conducted over 1000 epochs, incorporating only a limited amount of new data. Following retraining, the model is re-evaluated under the same conditions to determine the extent of accuracy improvement. This experiment provides insight into how efficiently the model can adapt to new heliostats with minimal additional training and whether such an approach could be used for continuous model updates in a real world deployment.

6.3.1. Performance on Unseen Heliostats

The accuracy achieved when using the pre-trained model for inference on new heliostats is 3.46 mrad, as shown in the histogram in Figure 6.5. While the model is still able to predict aimpoints with some degree of accuracy, its performance is significantly worse compared to Scenario 1. This suggests that, without exposure to a specific heliostat during training, the model struggles to generalize perfectly

to new heliostats, though it loses some predictive capability.

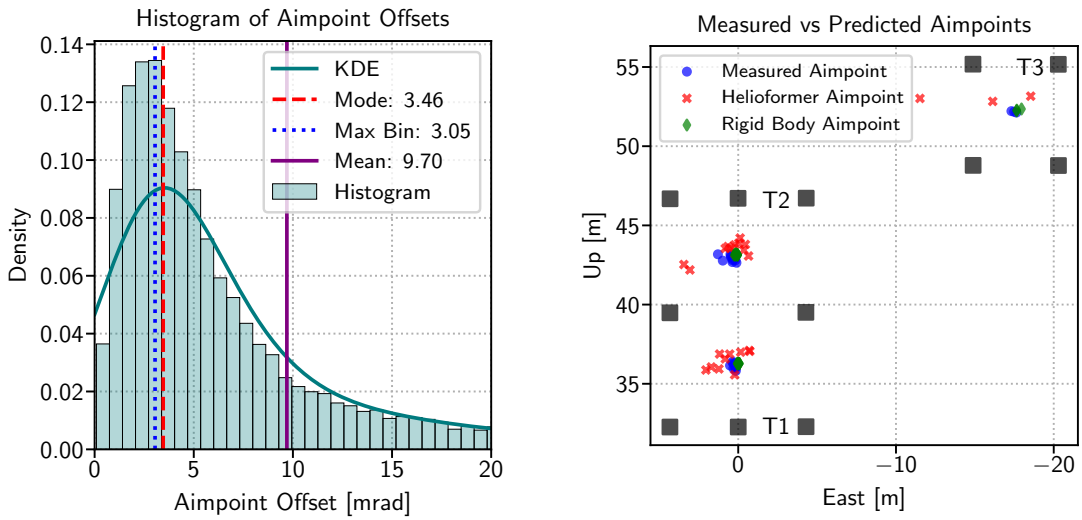

Figure 6.5.: **Histogram of angular offsets with a KDE for unseen heliostat evaluation in Scenario 2, showing an accuracy of 3.46 mrad (left). Scatter plot of aimpoint predictions for an exemplary unseen heliostat in Scenario 2 (right)**

Figure 6.5 presents a scatter plot of predicted aimpoints for these unseen heliostats. The results indicate that while the model still aligns aimpoints reasonably well with actual calibration targets, the predictions are more scattered and less precise compared to those in Scenario 1. This confirms that the model has some level of generalization ability, but the performance gap suggests that adaptation to new heliostats requires additional training.

6.3.2. Improvement Through Retraining

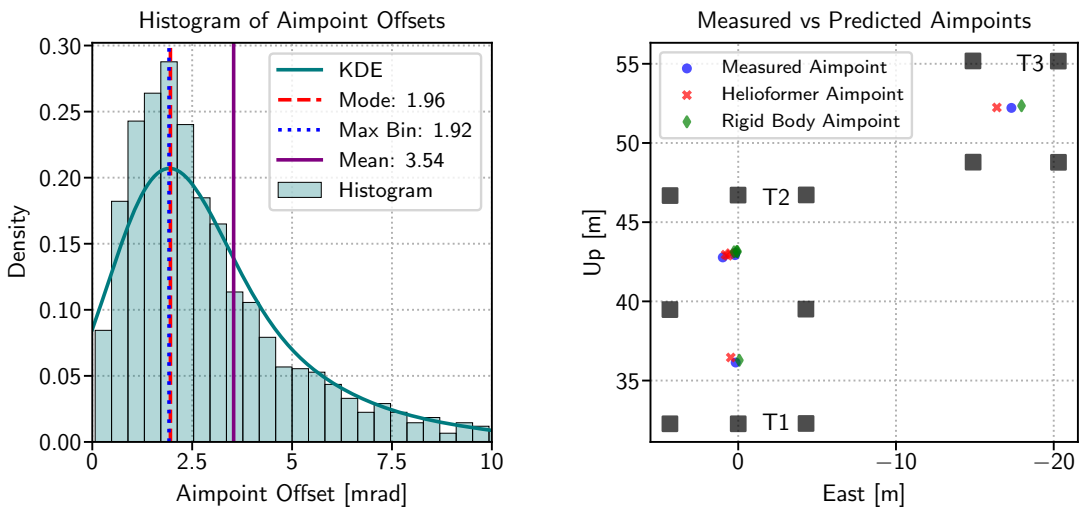

Figure 6.6.: **Histogram of angular offsets with a KDE after retraining in Scenario 3, showing an improved accuracy of 1.96 mrad (left). Scatter plot of aimpoint predictions after retraining for an exemplary heliostat in Scenario 3 (right)**

After retraining, the model achieved an accuracy of 1.96 mrad, as shown in Figure 6.6. This is a substantial improvement from the initial 3.46 mrad and brings performance almost back into the same range as Scenario 1. The retraining process required significantly less data for the new heliostats compared to the full training set used previously, demonstrating that a small amount of targeted additional data is sufficient to fine-tune the model for new heliostats.

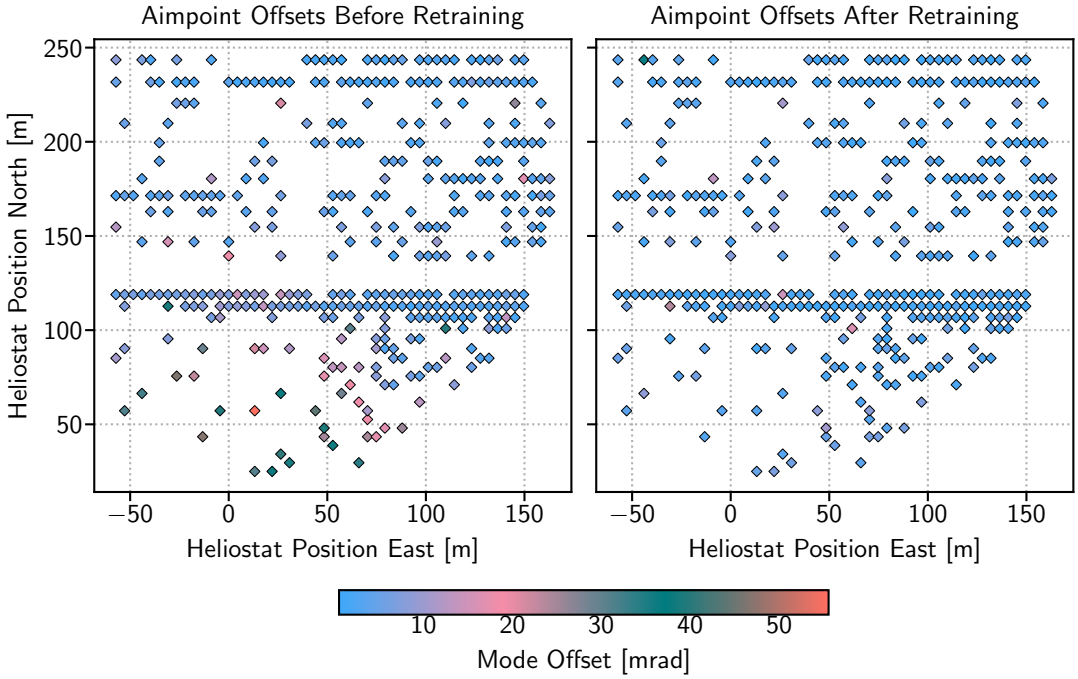

Figure 6.7.: **Comparison of aimpoint offsets for all new heliostats before and after retraining. The left plot shows aimpoint offsets before retraining (Scenario 2), while the right plot presents aimpoint offsets after retraining (Scenario 3). The color scale represents the mode offset in milliradians.**

Figure 6.6 presents a scatter plot of aimpoint predictions after retraining. Note that the number of predictions has decreased, as 25 of the 30 existing calibrations were used to retrain the model and can no longer be used for validation. As a result, only 5 calibrations remain available for evaluation. The alignment of predicted aimpoints is much closer to the actual targets compared to the initial evaluation on unseen heliostats. This confirms that retraining significantly improves the model's accuracy and adaptability, even with a limited amount of new data.

The results indicate that while the model has some generalization ability, its performance on completely unseen heliostats is notably lower. However, with relatively little additional training, it can be adapted to new heliostats, achieving accuracy levels comparable to those of fully trained heliostats.

Before retraining, heliostats positioned closer to the tower exhibit significantly higher aimpoint offsets, as shown in Figure 6.7. After retraining, these offsets decrease substantially, bringing their accuracy in line with the rest of the heliostat field. Additionally, several outliers that previously had high aimpoint offsets show a significant reduction, though some remain.

6.4. Scenario 4: Cross-Target Generalization

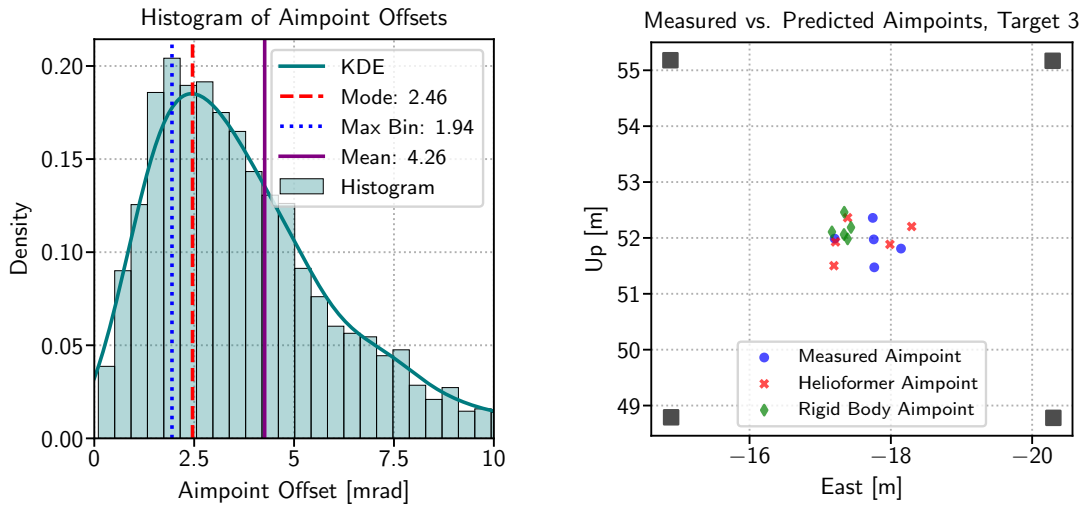

Figure 6.8.: **Histogram of angular offsets with a KDE for inference on Target 3 in Scenario 4, showing an accuracy of 2.46 mrad (left). Scatter plot of aimpoint predictions for inference on Target 3 for an exemplary heliostat in Scenario 4 (right).**

This section evaluates the model's ability to generalize across different calibration targets. Unlike previous scenarios, where the model was trained and tested on the same target locations, here the model is trained on a subset of targets and tested on an unseen target. Reference Figure 3.1 for the target locations on the tower. Two separate models were trained for this experiment:

- The first model was trained using only calibration data from **targets 1 and 2**, with inference performed on **target 3**.
- The second model was trained using data from **targets 1 and 3**, with inference performed on **target 2**.

This experiment assesses whether the model can generalize its understanding of heliostat calibration across different target locations, considering the geometric relationships between the targets.

The first model, trained on targets 1 and 2 and tested on target 3, achieved an accuracy of 2.46 mrad. The corresponding histogram and scatter plot of predicted aimpoints are shown in Figure 6.8, respectively. The results indicate that while the model retains reasonable accuracy, the predictions on target 3 are less accurate and more scattered compared to previous scenarios.

The second model, trained on targets 1 and 3 and tested on target 2, achieved a higher accuracy of 1.79 mrad, which is in line with scenario 1. The histogram and aimpoint scatter plot for this model are presented in Figures 6.9. The improved accuracy compared to the previous case is likely due to the geometric positioning of the targets. Target 3 is much farther from targets 1 and 2, making it more difficult for the model to extrapolate the learned calibration relationships. In

contrast, target 2 is positioned between targets 1 and 3, making generalization easier due to its proximity and intermediate location.

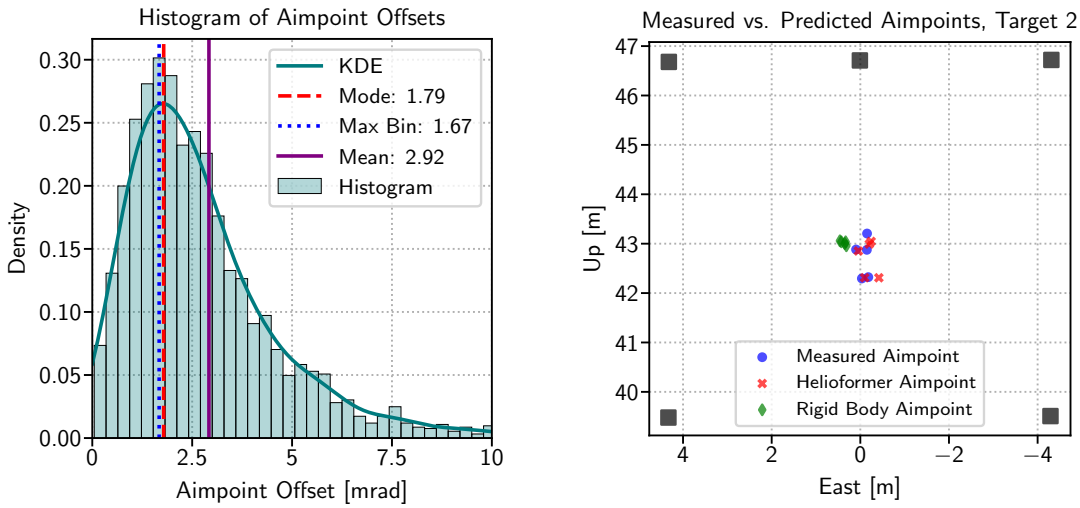

Figure 6.9.: **Histogram of angular offsets with a KDE for inference on Target 2 in Scenario 4, showing an accuracy of 1.79 mrad (left). Scatter plot of aimpoint predictions for inference on Target 2 for an exemplary heliostat in Scenario 4 (right)**

6.5. Training Set Size Analysis

This section explores how the size of the training dataset affects the accuracy of the model. To determine how many calibration points would be required to achieve sufficient accuracy in a newly constructed solar tower plant, four models were trained with different training set sizes. The goal of this experiment is to assess the impact of dataset size on both mixed and worst-case accuracy and to identify the point at which additional calibration data provides no improvement.

Figure 6.10 presents a scatter plot of the mixed and worst-case accuracy as a function of training set size. The results show a clear trend, as the number of calibration points increases, the model's accuracy improves. The mode of the mixed accuracy is consistently better across all training set sizes, reflecting the expected real-world performance where past calibration data is available for inference.

The worst-case accuracy, where the encoder has no prior calibration data for a heliostat, starts off worse than the rigid body model at very small training set sizes. However, once the training set size exceeds 50 calibrations per heliostat, the worst-case accuracy also surpasses the rigid body model. This indicates that while the *Helioformer* model requires a minimum dataset size to generalize effectively in the absence of prior calibration data, it eventually outperforms traditional methods even in this scenario.

These results provide insight into the data requirements for deploying the *Helioformer* model in a new solar tower plant. While increasing the number of calibration points improves accuracy, the rate of improvement decreases at larger

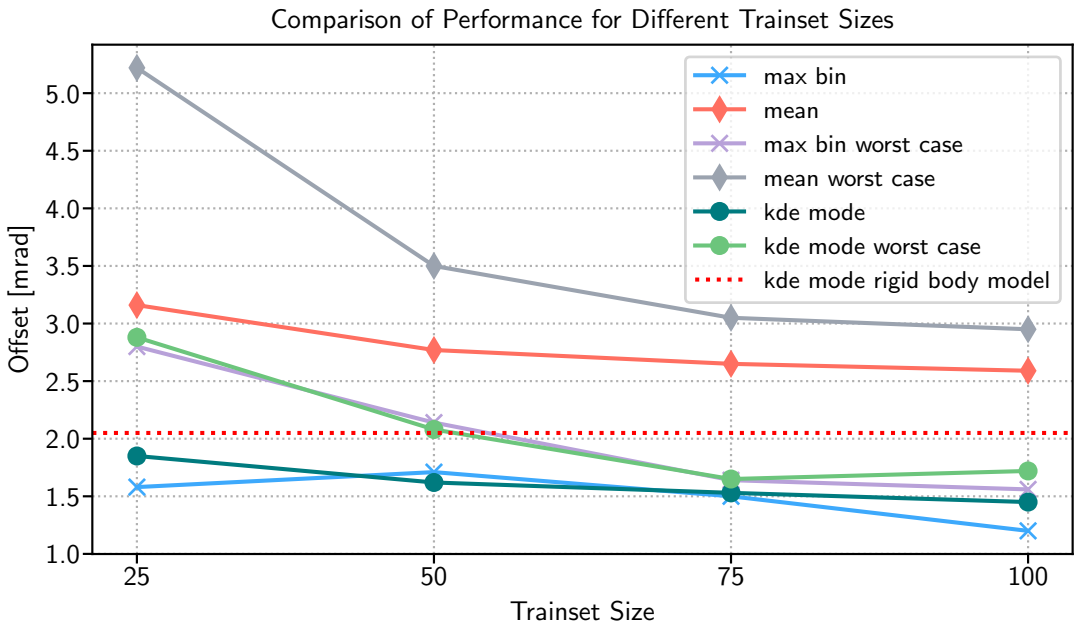

Figure 6.10.: **Plot of mixed and worst-case accuracy as a function of training set size. The accuracy of the rigid body model is also included for comparison.**

training set sizes. In a practical setting, this suggests that while having at least 50 calibrations per heliostat is necessary to ensure robust performance even in worst-case scenarios, the model can already outperform geometric calibration methods with smaller training sets when using past calibration data. This trade-off between data collection effort and model performance should be considered when determining an optimal calibration strategy for real-world deployments.

6.6. Encoder Input Size Analysis

This section explores how the number of calibration points provided to the encoder affects model performance. Using the same dataset as in Scenario 1, different encoder input sizes were tested to analyze how the amount of available calibration data influences prediction accuracy. The goal of this experiment is to determine whether providing more input data to the encoder significantly improves accuracy and whether there is an optimal number of calibration points beyond which additional data no longer provides substantial benefits.

Figure 6.11 presents a scatter plot comparing mixed and worst-case accuracy for different encoder input sizes. The results show that the *Helioformer* model consistently outperforms the rigid body model across all tested encoder sizes. However, the improvements diminish beyond a certain point, with an encoder size of 20 providing the best accuracy.

The results suggest that while increasing the encoder input size initially leads to better accuracy, further increases do not yield substantial improvements. This could be due to model constraints, where the current model architecture reaches its capacity to process additional calibration data in the encoder efficiently.

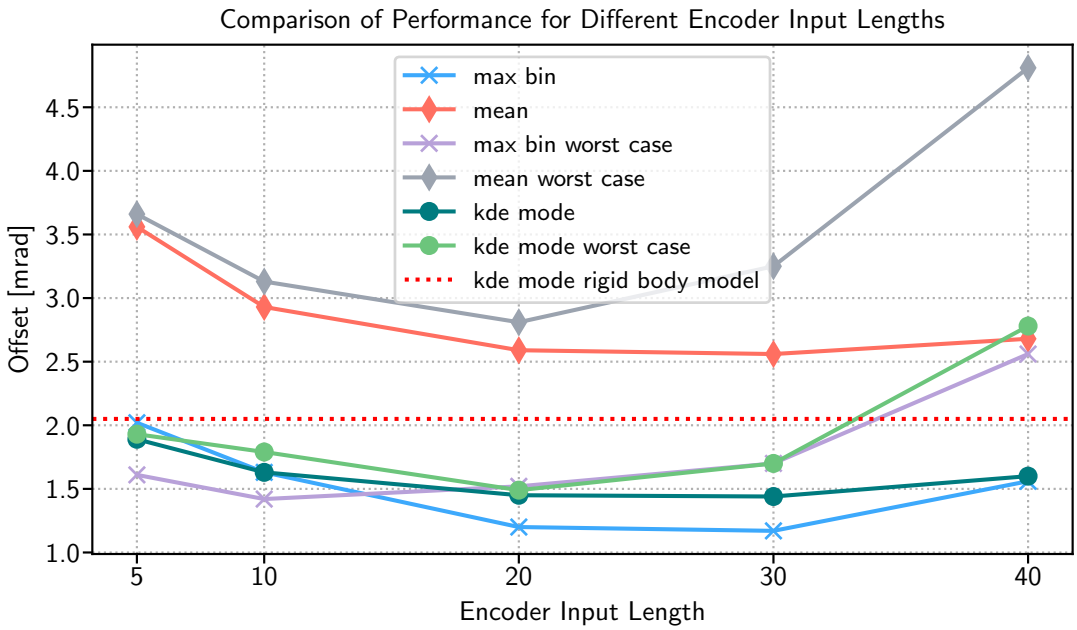

Figure 6.11.: **Plot of mixed and worst-case accuracy as a function of encoder input size. The accuracy of the rigid body model is also included for comparison.**

6.7. Computational Efficiency and Scalability

The computational efficiency of the *Helioformer* model is a critical factor in determining its feasibility for large-scale deployment in solar tower plants. The model was trained using the dataset from Scenario 1, which consists of 916 heliostats with 100 calibrations each, resulting in a total of 916,100 data points. During training, the model utilized 15,478 MiB of GPU memory, reflecting the computational demands required to process the dataset efficiently.

The *Helioformer* model has a total of 22.2 million parameters, occupying approximately 88,847 MB of memory. While this parameter count is relatively moderate compared to large-scale deep learning models used in other domains, it still presents challenges in terms of memory consumption, particularly when scaling to larger datasets or real-time inference applications. The memory footprint is primarily influenced by the number of transformer layers, attention heads, and the dimensionality of the embeddings.

Training the largest model, corresponding to the dataset from Scenario 1, took approximately 14 hours to complete 100,000 epochs. Each epoch required only a fraction of a second, as the large batch size of 4×916 heliostats allowed every step to process the entire dataset once. While this makes training computationally intensive, it also ensures that the model sees all available calibration data in every iteration, promoting faster convergence. The relatively short overall training time suggests that retraining or fine-tuning the model for new heliostats is feasible, making it adaptable to dynamic calibration needs.

In addition to efficient training, inference is equally fast. A single inference pass, just like a training step, takes approximately half a second. This makes real-time

deployment feasible, as predictions can be made at a speed suitable for continuous heliostat tracking and adjustment. The ability to process large heliostat fields within such a short time frame indicates that the model could be integrated into real-time control systems without introducing significant computational delays.

The dataset size in this study represents a mid-sized solar tower plant, and future implementations in larger fields would require proportionally more memory and computational power. As the number of heliostats and calibration points increases, the primary bottleneck becomes both GPU memory usage during training and real-time inference efficiency. While the current model can handle nearly a million calibration points, increasing the dataset size significantly may require additional optimization techniques such as distributed training, model compression, or mixed precision training to maintain scalability without compromising accuracy.

7. Discussion

This chapter presents a comprehensive analysis of the findings of the conducted experiments, addressing the research questions, and evaluating the implications of the proposed neural network-based calibration approach. The discussion begins by assessing how well the *Helioformer* model meets the research objectives, comparing its performance with traditional rigid body calibration methods. The key results are then interpreted, focusing on generalization capabilities, dataset size effects, and computational efficiency. Finally, the broader implications for solar tower plants are explored, highlighting the potential benefits of deep learning-based calibration and improvements in tracking precision.

7.1. Addressing Research Questions and Objectives

The primary goal of this research was to develop a generalized neural network model for heliostat calibration, leveraging shared characteristics among heliostats to improve accuracy and reduce reliance on extensive calibration data for each individual unit. To achieve this, the thesis investigated the following research questions:

RQ1: How can neural networks be effectively utilized to develop a generalized approach for calibrating heliostats in a solar tower power plant, ensuring improved accuracy and efficiency compared to traditional methods?

The evaluation results demonstrate that neural networks, specifically the *Helioformer* model, can successfully learn heliostat calibration patterns and significantly improve accuracy over traditional rigid body model calibration methods. In all experimental scenarios, the model outperformed the rigid body approach, with worst-case accuracy surpassing the rigid body model when trained with at least 50 calibration points per heliostat. Additionally, the ability to predict motor positions directly provides an alternative usage method that enables integration into automated heliostat control systems. These findings confirm that deep learning techniques offer a viable and efficient solution for heliostat calibration.

RQ2: What strategies can be employed to design and implement a robust neural network architecture that can generalize well across different heliostats, ensuring consistent performance and accuracy despite variations in environmental conditions and heliostat configurations?

The experimental results indicate that the Helioformer model generalizes well within the training distribution, but requires additional data for optimal performance on unseen heliostats. In Scenario 2, when tested on previously unseen heliostats without retraining, the model's accuracy dropped to 3.46 mrad. However, after limited retraining on only 1,000 epochs with new heliostats, accuracy

improved to 1.96 mrad, not quite comparable to the performance on known heliostats, but already beating the rigid body model. This suggests that while the model benefits from shared structural characteristics across heliostats, some degree of fine-tuning is necessary for optimal generalization. Furthermore, the cross-target evaluation in Scenario 4 demonstrated that the geometric relationship between targets influences the generalization performance, emphasizing the need for diverse training data to enhance robustness.

Beyond these empirical findings, insights from model development highlight key software engineering considerations that contributed to *Helioformer's* performance.

A crucial design choice that led to substantial improvements was the use of learnable embeddings for each input feature. Instead of relying on predefined feature transformations, the model was able to dynamically learn optimal representations, improving its ability to generalize across different heliostats and calibration conditions.

Another important factor in training efficiency was the combination of a decreasing learning rate and a small dropout rate. This approach significantly improved training convergence, allowing the model to reach minima in a relatively short amount of time while maintaining robustness and avoiding overfitting.

The computational efficiency of the model was also evaluated. Although the transformer-based approach was chosen for its ability to capture spatial dependencies, memory usage remained relatively small despite the fact that the model had 22.2 million parameters. This suggests that the model has room to scale, either by increasing the number of parameters to improve capacity or by incorporating larger training datasets without exceeding typical GPU memory constraints.

The research objectives outlined in the introduction were successfully addressed.

The transformer-based *Helioformer* model was selected for its ability to capture spatial dependencies and process calibration data efficiently. Its strong performance across various experimental scenarios validates this choice.

The model was implemented and tested on a real-world dataset, demonstrating substantial accuracy improvements over traditional methods.

The experiments identified key factors influencing model performance, including the size of the training set, the size of the input of the encoder, and generalization strategies. The results suggest that a moderate amount of calibration data is necessary for reliable accuracy, but retraining can significantly enhance adaptability.

7.2. Interpretation of Key Results

The results highlight several important trends in the calibration of heliostats using neural networks, providing information on both the advantages and limitations of the proposed *Helioformer* model.

Neural network outperforms rigid body model. In all experiments, the *Helioformer* model consistently achieved higher precision than the traditional rigid body calibration method. This improvement was observed even with a relatively

small training dataset, suggesting that data-driven approaches are more effective in modeling real-world deviations in heliostat tracking. The results confirm that neural networks can capture complex, non-linear relationships that rigid body models fail to account for, making them a more robust alternative for heliostat calibration. Additionally, the model's ability to predict motor positions directly further expands its potential applications, offering an alternative way to control heliostat movements based on learned relationships between motor positions and aimpoints. Although existing methods also allowed for motor position control, the key innovation lies in predicting motor positions while evaluating performance based on the resulting aimpoints. This approach provides a meaningful interpretation of otherwise abstract motor positions by directly linking them to their respective aimpoint.

Generalization is possible but requires fine-tuning. The model demonstrated the ability to generalize to new heliostats, although with some loss in accuracy. When tested on previously unseen heliostats without retraining, accuracy decreased to 3.46 mrad, highlighting that while the model benefits from shared structural characteristics across heliostats, some heliostat-specific calibration information is still needed. However, retraining on only 1,000 additional epochs using data from the new heliostats significantly improved performance, bringing the accuracy back to 1.96 mrad. These results are still not as good as those obtained for known heliostats, but they already show an improvement over the rigid body model.

This rapid adaptation suggests that the model is not simply memorizing individual heliostat behaviors but instead learning shared properties among heliostats in the field. This finding has significant implications for data collection strategies. It suggests that instead of requiring a large number of calibrations per heliostat, it may be possible to train a generalized model with fewer calibration points per heliostat while still achieving high accuracy. This approach could be particularly valuable in new solar tower plants, where collecting extensive calibration data is time-consuming and resource-intensive. Additionally, the success of fine-tuning implies that periodic retraining with small amounts of new data could allow the model to adapt to long-term changes such as mechanical wear or environmental shifts.

Training set size and encoder input size influence performance. The evaluation showed that increasing the training set size consistently improved accuracy, with diminishing returns beyond 50 calibration points per heliostat. This suggests that while more data benefits the model, excessive calibration points may not provide substantial additional gains beyond a certain threshold. The encoder input size experiment further supported this observation, with an input size of 20 calibration points that yielded the best performance. Providing more encoder input data did not significantly improve results, possibly due to architectural constraints or a saturation of useful information being extracted. This insight suggests that future models with larger capacities could be explored to determine whether they can utilize more encoder input effectively. In the case of the *Helioformer*, this could be achieved either by increasing the number of encoder

layers or by expanding the embedding dimension beyond 512, allowing the model to better capture and process additional calibration information.

Computational efficiency is viable for real-world deployment. The largest *Helioformer* model was trained in approximately 14 hours on a dataset of 916 Heliostats, 100 calibration points each, consuming 15,478 MiB of GPU memory. Despite its 22.2 million parameters, memory usage remained relatively small, suggesting that the model has room to scale. This allows for potential expansions in two directions, increasing model capacity with more parameters to improve generalization or incorporating larger training datasets to enhance robustness.

The training process itself was computationally intensive, but remained manageable within typical high-performance computing resources available for industrial applications. The use of a decreasing learning rate and small dropout helped the model to converge efficiently, reaching a stable solution within a reasonable time frame. The results suggest that scaling the model to larger heliostat fields would be feasible without significant modifications to computational infrastructure.

Spatial relationships affect calibration accuracy. The cross-target evaluation revealed that the accuracy of the calibration is influenced by the geometric distribution of the calibration targets. The model was able to make reasonable predictions on targets far outside the training distribution, although with slightly reduced accuracy. This ability to extrapolate beyond the trained target locations confirms that the model does not simply interpolate between existing data points, but can infer meaningful relationships between heliostat positions and aimpoints.

This finding also suggests a practical consideration for solar tower plant design. Since the positioning of calibration targets affects model accuracy, an optimal strategy could be to place targets in spatially distributed locations to provide diverse training data. For example, placing one calibration target directly to the left and one to the right of the receiver could improve generalization across the heliostat field.

Training on motor positions is possible with similar accuracy. In addition to normal vector prediction, the model was successfully trained to predict motor positions, achieving a KDE mode accuracy of 1.31 mrad. This performance is even better than that of normal vector-based calibration, demonstrating that both approaches are viable for practical use. The ability to train a model that directly predicts motor positions opens up new possibilities for integrating deep learning-based calibration with real-time heliostat control systems. Instead of relying on intermediate normal vector representations, a model trained on motor positions could directly provide optimal actuation commands, potentially simplifying the deployment of automated tracking systems.

These findings confirm that deep learning is a powerful tool for heliostat calibration, capable of exceeding traditional rigid body models, adapting to new heliostats with minimal retraining, and enabling alternative calibration approaches such as motor position prediction. The results provide a strong foundation for future improvements in model efficiency, scalability, and real-world implementation

strategies.

7.3. Implications for Solar Tower Plants

The *Helioformer* model's ability to accurately predict aimpoints and motor positions has significant implications for the efficiency of solar tower plants. Although regular calibrations are still required, the model greatly improves the accuracy achieved, ensuring that heliostats maintain better alignment and reducing cumulative tracking errors. This enhanced precision leads to more effective energy capture, as reflected sunlight is more consistently concentrated on the receiver. As a result, the system operates closer to its optimal efficiency, minimizing energy losses and maximizing the overall power output of the plant.

Beyond efficiency gains, better calibration accuracy also leads to more optimal and controlled receiver illumination. Precise targeting reduces the risk of local overheating, which can degrade or even damage receiver components over time. By preventing such failures, the model can not only enhance energy output but also contribute to the longevity and reliability of the solar tower system, reducing maintenance costs and operational disruptions.

Another promising application is the potential transferability of trained models across STPs. Since many facilities use similar heliostat designs, a model trained on one operational STP could be adapted to a newly constructed plant by fine-tuning it with a limited amount of calibration data. This approach could significantly reduce the time required to achieve optimal performance in a new STP, allowing it to reach high efficiency much faster than with conventional calibration methods.

Additionally, the model's ability to predict motor positions instead of normal vectors opens up the possibility of direct integration into heliostat field control systems. By providing motor position predictions as control signals, the *Helioformer* model could serve as the core of an automated heliostat tracking system, eliminating the need for indirect computations based on normal vectors and simplifying real-time control implementations.

From a computational perspective, the model remains highly efficient. Despite the deep learning framework, inference requires only about half a second, making real-time or near-real-time calibration feasible. The training process, while computationally intensive, remains manageable even for large datasets, and memory consumption leaves room for expansion in model complexity or dataset size. These factors make large-scale deployment of deep learning-based heliostat calibration both practical and scalable.

8. Conclusion and Future Research

This chapter summarizes the key findings of this thesis and outlines potential directions for future research. The conclusion highlights the success of the *Helioformer* model in improving the accuracy of the heliostat calibration through deep learning, demonstrating its ability to outperform traditional rigid body models while maintaining computational efficiency. The generalization capabilities of the model were evaluated across different heliostats and calibration targets, revealing both strengths and areas for improvement. The chapter also discusses the practical implications of integrating the model into real-world heliostat control systems. Finally, future research directions are proposed, focusing on enhancing generalization, incorporating raw calibration images, enabling real-time deployment, and scaling the model to other solar tower plants.

8.1. Conclusion

This thesis explored the application of deep learning to heliostat calibration in STPs, introducing the *Helioformer* model as a data-driven alternative to traditional rigid body calibration methods. The results demonstrated that neural networks can successfully learn complex, non-linear relationships governing heliostat tracking, significantly outperforming rigid body models across all evaluated scenarios. The model achieved high accuracies in predicting both the aimpoints and motor positions, even with a relatively small training dataset, confirming the viability of machine learning for improving the efficiency of STPs. By leveraging shared characteristics among heliostats, the model was able to capture intricate dependencies between motor positions, sun position, and aimpoint location, improving calibration precision beyond what is possible with physics based models.

One of the most significant findings was the model's ability to generalize across different heliostats. When tested on previously unseen heliostats without retraining, the accuracy dropped compared to that of known heliostats, indicating that heliostat-specific variations still play a role in calibration accuracy. However, retraining with a relatively small amount of additional data, only 1,000 epochs, allowed the model to recover its original performance level, suggesting that it effectively learns shared properties of heliostats rather than simply memorizing individual behaviors. This finding implies that a deep learning-based approach can significantly reduce the data requirements for new solar plants, as a model trained on an existing facility could be fine-tuned with minimal additional calibration data, accelerating the deployment of heliostat fields, and reducing operational downtime.

Furthermore, cross-target generalization evaluation revealed that spatial relationships between calibration targets influence model performance. When the

model was trained on two targets and tested on a third, accuracy varied depending on the geometric distribution of the targets. The model performed best when the unseen target was positioned between two known targets, suggesting that interpolation is more reliable than extrapolation. However, even when tested on a target located far outside the training distribution, the model still achieved reasonable accuracy, demonstrating that it can infer meaningful calibration relationships beyond direct data interpolation. This insight suggests that strategic placement of calibration targets, for example symmetrically around the receiver, could further enhance model accuracy, ensuring that learned relationships generalize well across the heliostat field.

Beyond calibration accuracy, the study also confirmed that the *Helioformer* model is computationally efficient. Despite the complexity of transformer-based architectures, inference takes only about half a second, making real-time or near-real-time application feasible. The ability to predict motor positions directly further enhances the model's practical applicability. Instead of relying on an intermediate normal vector prediction step, the *Helioformer* model can directly infer the optimal motor settings required to achieve a desired aimpoint. This capability makes it possible to integrate the *Helioformer* directly into heliostat control systems, replacing traditional tracking algorithms with an end to end neural network based solution.

8.2. Future Research

While the *Helioformer* model has shown strong performance, several areas remain for future improvement and exploration.

One key direction is improving the model's generalization capabilities to minimize the need for retraining when deployed on new heliostats. This could involve architectural modifications, such as hybrid transformer-CNN models or improved embedding strategies, to better capture heliostat field-wide patterns.

Another promising area is integrating raw calibration images directly into the model. The current approach relies on an external algorithm to extract aimpoint positions from calibration images, but this process may introduce errors. By incorporating convolutional layers into the *Helioformer* model, raw images could be processed as direct input, allowing the network to learn from the calibration images themselves rather than relying on preextracted data. This could be particularly straightforward to implement for motor position prediction, where the model could be trained to infer optimal motor settings based on visual feedback.

Further research could also explore real-time deployment strategies. Given the model's efficient inference time, implementing continuous tracking adjustments based on changing sun positions, atmospheric conditions, and structural shifts could enhance long-term STP performance. In addition, transfer learning approaches could be investigated to determine whether a model trained on data from one STP could be adapted to another plant with minimal retraining, significantly reducing the time required for a new plant to reach optimal efficiency.

Finally, scalability remains an important consideration. The model's memory footprint and training time suggest that larger datasets and higher-capacity architectures could be accommodated without excessive computational demands. Exploring distributed training methods or model compression techniques could further enhance scalability, making deep learning-based calibration viable for even the largest heliostat fields.

By addressing these research directions, future work can build on the foundation established in this thesis, refining the *Helioformer* model into an even more robust and adaptable tool for automated heliostat calibration in solar tower plants.

9. Bibliography

[1] Intergovernmental (IPCC). *Climate Change 2022 – Impacts, Adaptation and Vulnerability: Working Group II Contribution to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change.* June 29, 2023.

[2] Ehsanul Kabir, Pawan Kumar, Sandeep Kumar, Adedeji A. Adelodun, and Ki-Hyun Kim. "Solar energy: Potential and future prospects". In: *Renewable and Sustainable Energy Reviews* 82 (Feb. 1, 2018), pp. 894–900.

[3] L. El Chaar, L. A. lamont, and N. El Zein. "Review of photovoltaic technologies". In: *Renewable and Sustainable Energy Reviews* 15.5 (June 1, 2011), pp. 2165–2175.

[4] Pushkaraj D. Sonawane and V. K. Bupesh Raja. "An overview of concentrated solar energy and its applications". In: *International Journal of Ambient Energy* 39.8 (Nov. 17, 2018). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/01430750.2017.1345009, pp. 898–903.

[5] K. Hennecke, P. Schwarzbözl, G. Koll, M. Beuter, B. Hoffschmidt, J. Göttsche, and T. Hartz. "The Solar Power Tower Jülich — A Solar Thermal Power Plant for Test and Demonstration of Air Receiver Technology". In: *Proceedings of ISES World Congress 2007 (Vol. I – Vol. V).* Ed. by D. Yogi Goswami and Yuwen Zhao. Berlin, Heidelberg: Springer, 2009, pp. 1749–1753.

[6] Ugo Pelay, Lingai Luo, Yilin Fan, Driss Stitou, and Mark Rood. "Thermal energy storage systems for concentrated solar power plants". In: *Renewable and Sustainable Energy Reviews* 79 (Nov. 1, 2017), pp. 82–100.

[7] Olivier Dumont, Guido Francesco Frate, Aditya Pillai, Steven Lecompte, Michel De paepe, and Vincent Lemort. "Carnot battery technology: A state-of-the-art review". In: *Journal of Energy Storage* 32 (Dec. 1, 2020), p. 101756.

[8] Shahjadi Hisan Farjana, Nazmul Huda, M. A. Parvez Mahmud, and R. Saidur. "Solar process heat in industrial systems – A global review". In: *Renewable and Sustainable Energy Reviews* 82 (Feb. 1, 2018), pp. 2270–2286.

[9] Daniel Fernández-González, I. Ruiz-Bustinza, Carmen González-Gasca, Juan Piñuela Noval, Javier Mochón-Castaños, José Sancho-Gorostiaga, and Luis Felipe Verdeja. "Concentrated solar energy applications in materials science and metallurgy". In: *Solar Energy* 170 (Aug. 1, 2018), pp. 520–540.

[10] Deepak Yadav and Rangan Banerjee. "A review of solar thermochemical processes". In: *Renewable and Sustainable Energy Reviews* 54 (Feb. 1, 2016), pp. 497–532.

[11] Scott Jones, Ronald Lumia, Roger Davenport, Robert Thomas, David Gorman, Gregory Kolb, and Matthew Donnelly. *Heliostat cost reduction study.* SAND2007-3293, 912923. June 1, 2007, SAND2007–3293, 912923.

[12] Alberto Sánchez-González, María Reyes Rodríguez-Sánchez, and Domingo Santana. "Aiming factor to flatten the flux distribution on cylindrical receivers". In: *Energy* 153 (June 15, 2018), pp. 113–125.

[13] Clifford K. Ho and Brian D. Iverson. "Review of high-temperature central receiver designs for concentrating solar power". In: *Renewable and Sustainable Energy Reviews* 29 (Jan. 1, 2014), pp. 835–846.

[14] *Analysis of thermal stress, fatigue life and allowable flux density for the molten salt receiver in solar power tower plants | International Journal of Low-Carbon Technologies | Oxford Academic.* URL: `https://academic.oup.com/ijlct/article/doi/10.1093/ijlct/ctac110/6831626` (visited on 02/20/2025).

[15] Viorel Badescu. "Theoretical derivation of heliostat tracking errors distribution". In: *Solar Energy* 82.12 (Dec. 1, 2008), pp. 1192–1197.

[16] Mahamadou Maiga, Kokouvi Edem N'Tsoukpoe, Aboubakar Gomna, and YesuenyeagbeA. K. Fiagbe. "Sources of solar tracking errors and correction strategies for heliostats". In: *Renewable and Sustainable Energy Reviews* 203 (Oct. 1, 2024), p. 114770.

[17] Johannes Christoph Sattler, Marc Röger, Peter Schwarzbözl, Reiner Buck, Ansgar Macke, Christian Raeder, and Joachim Göttsche. "Review of heliostat calibration and tracking control methods". In: *Solar Energy* 207 (Sept. 1, 2020), pp. 110–132.

[18] Max Pargmann and Daniel Quinto. "How Can Deep Learning Be Used To Improve The Heliostat Field Calibration, Even With Small Data Sets? - A Transfer Learning Comparison Study". In: AIP Conference Proceedings. Vol. 2445. Jan. 1, 2021.

[19] Max Pargmann and Daniel Quinto. "Performance Increase of Solar Power Plants by applying Deep Learning Algorithms for Heliostatfield Calibration". In: Oct. 1, 2019.

[20] Max Pargmann, Moritz Leibauer, Daniel Quinto, Vincent Nettelroth, and Robert Pitz-Paal. "Reliable neural network heliostat calibration on low data". In: Oct. 5, 2023, p. 24.

[21] Meige Zheng, José Zapata, Charles-Alexis Asselineau, Joe Coventry, and John Pye. "Analysis of tubular receivers for concentrating solar tower systems with a range of working fluids, in exergy-optimised flow-path configurations". In: *Solar Energy* 211 (Nov. 15, 2020), pp. 999–1016.

[22] Xiudong Wei, Zhenwu Lu, Zhifeng Wang, Weixing Yu, Hongxing Zhang, and Zhihao Yao. "A new method for the design of the heliostat field layout for solar tower power plant". In: *Renewable Energy* 35.9 (Sept. 1, 2010), pp. 1970–1975.

[23] Giancarlo Gentile, Giovanni Picotti, Marco Binotti, Michael E. Cholette, and Giampaolo Manzolini. "A comprehensive methodology for the design of solar tower external receivers". In: *Renewable and Sustainable Energy Reviews* 193 (Apr. 1, 2024), p. 114153.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. Aug. 2, 2023. arXiv: `1706.03762[cs]`.

[25] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. "A review on the attention mechanism of deep learning". In: *Neurocomputing* 452 (Sept. 10, 2021), pp. 48–62.

[26] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. "A survey of transformers". In: *AI Open* 3 (Jan. 1, 2022), pp. 111–132.

[27] Max Pargmann, Moritz Leibauer, Vincent Nettelroth, Daniel Quinto, and Robert Pitz-Paal. *Enhancing Heliostat Calibration on Low Data by Fusing Robotic Rigid Body Kinematics with Neural Networks*. May 12, 2023.

[28] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. *On the Variance of the Adaptive Learning Rate and Beyond*. Oct. 26, 2021. arXiv: `1908.03265[cs]`.

Appendix

A. Example Loss Curve and Learning Rate Schedule

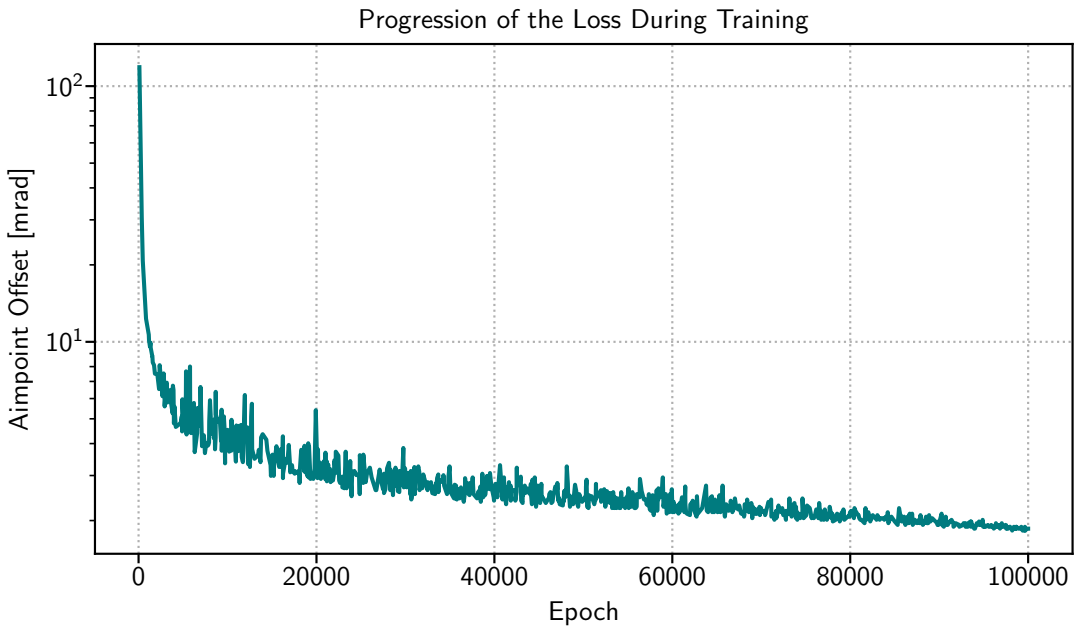

Figure A.1.: **Example of the loss curve during the training of the *Helioformer* model for Scenario 1, with a trainset size of 100 and an encoder input length of 20.**

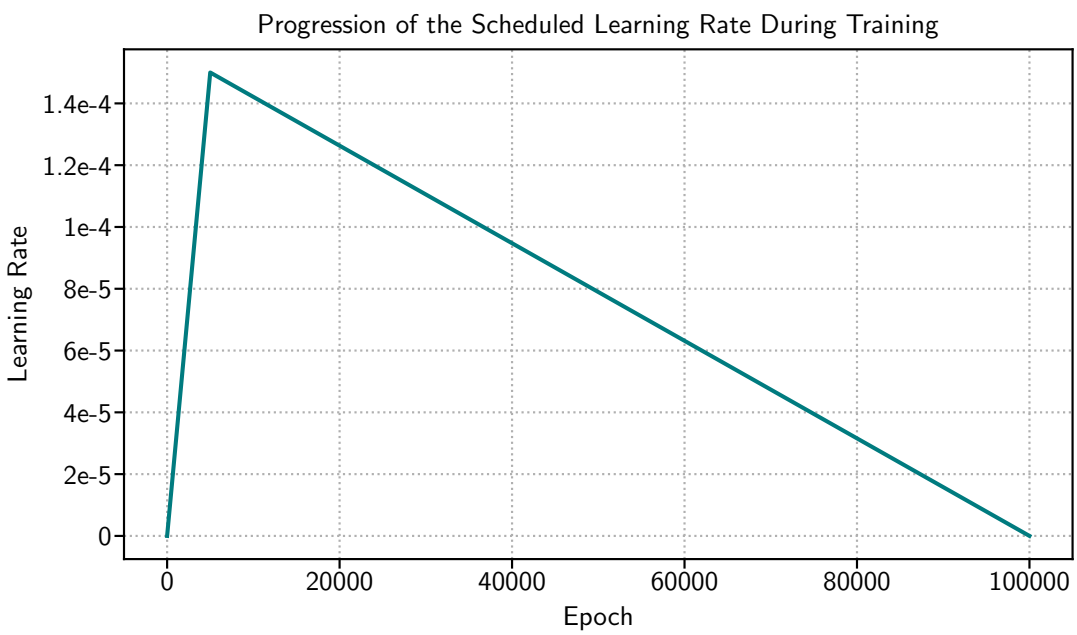

Figure A.2.: **Example of the learning rate schedule used during the training of the Helioformer models, with a warmup over 5% of the epochs, starting at 0, warming up to 1.5×10^{-4}, and then decaying back to 0 over the course of the training.**

B. Relevant Source Code

Listing A.1: **Learnable_Embedding.py**

```
import torch
import torch.nn as nn

class LearnedFeaturePositionalEncodingAll(nn.Module):
    def __init__(self, d_model: int, n_features: int):
        super(LearnedFeaturePositionalEncodingAll, self).__init__()

        # Create a separate linear transformation for each feature dimension
        self.linears = nn.ModuleList([nn.Linear(1, d_model) for _ in range(n_features)])
        self.d_model = d_model
        self.norm = nn.LayerNorm(d_model)

    def forward(self, feature_inputs: torch.Tensor, mask=[]):

        batch_size, seq_len, n_features = feature_inputs.shape

        # Initialize a tensor to accumulate the positional encodings
        pos_encoding = torch.zeros((batch_size, seq_len, self.d_model), device=feature_inputs.device)

        # Apply each linear transformation to the corresponding feature dimension
        for i in range(n_features):
            if i not in mask:
                # Extract the i-th feature, shape: (batch_size, sequence_length, 1)
                feature_i = feature_inputs[:, :, i].unsqueeze(-1)

                # Apply the corresponding linear transformation and add to the positional encoding
                embedding += self.linears[i](feature_i)
        embedding = self.norm(pos_encoding)
        return embedding
```

Listing A.2: **Dataset_creator.py**

```
import Pipeline.Data_Loader as dl
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans

def add_sun_vector_to_df(df):
    df = df.copy()

    # Extract Sun position components
    sun_vector_e = df['SunPosE']
    sun_vector_n = df['SunPosN']
    sun_vector_u = df['SunPosU']

    # Vectorized calculation of azimuth and elevation
    azimuth_rad = np.arctan2(sun_vector_e, sun_vector_n)
    elevation_rad = np.arcsin(sun_vector_u)

    # Normalize azimuth to [0, 2)
    azimuth_rad = azimuth_rad % (2 * np.pi)

    # Convert to degrees
    azimuth_deg = np.rad2deg(azimuth_rad)
    elevation_deg = np.rad2deg(elevation_rad)

    # Add results to the DataFrame
    df['Sun_Azimuth_rad'] = azimuth_rad
    df['Sun_Elevation_rad'] = elevation_rad
    df['Sun_Azimuth_deg'] = azimuth_deg
    df['Sun_Elevation_deg'] = elevation_deg

    return df

def add_normal_vector_to_df(df):
    df = df.copy()

    # Vectorized calculations for aim vectors
    aim_vector_e = df['TargetOffsetE'] - df['x']
    aim_vector_n = df['TargetOffsetN'] - df['y']
    aim_vector_u = df['TargetOffsetU'] - df['z']

    # Normalize aim vectors
    aim_magnitude = np.sqrt(aim_vector_e ** 2 + aim_vector_n ** 2 + aim_vector_u ** 2)
    aim_vector_e /= aim_magnitude
    aim_vector_n /= aim_magnitude
    aim_vector_u /= aim_magnitude

    # Normalize sun vectors
    sun_magnitude = np.sqrt(df['SunPosE'] ** 2 + df['SunPosN'] ** 2 + df['SunPosU'] ** 2)
    sun_vector_e = df['SunPosE'] / sun_magnitude
    sun_vector_n = df['SunPosN'] / sun_magnitude
    sun_vector_u = df['SunPosU'] / sun_magnitude

```

```
    # Calculate normal vectors
    normal_e = (sun_vector_e + aim_vector_e)
    normal_n = (sun_vector_n + aim_vector_n)
    normal_u = (sun_vector_u + aim_vector_u)

    # Normalize the normal vectors
    normal_magnitude = np.sqrt(normal_e ** 2 + normal_n ** 2 + normal_u ** 2)
    normal_e /= normal_magnitude
    normal_n /= normal_magnitude
    normal_u /= normal_magnitude

    # Assign normalized vector components to the DataFrame
    df['NormalE'] = normal_e
    df['NormalN'] = normal_n
    df['NormalU'] = normal_u

    # Calculate azimuth and elevation
    df['Norm_Azimuth_rad'] = np.arctan2(df['NormalE'], df['NormalN']) % (2 * np.pi) # Normalize azimuth to [0, 2)
    df['Norm_Elevation_rad'] = np.arcsin(df['NormalU']) # Elevation in radians

    return df

def add_aim_vector_to_df(df):
    df = df.copy()

    # Compute differences in coordinates
    delta_x = df['TargetOffsetE'] - df['x']
    delta_y = df['TargetOffsetN'] - df['y']
    delta_z = df['TargetOffsetU'] - df['z']

    # Compute radial distance (magnitude of the vector)
    distance = np.sqrt(delta_x ** 2 + delta_y ** 2 + delta_z ** 2)

    # Compute azimuth (angle from the positive Y-axis in the XY-plane)
    azimuth_radians = np.arctan2(delta_x, delta_y) % (2 * np.pi)
    azimuth_degrees = np.degrees(azimuth_radians)

    # Compute elevation (angle from the XY-plane)
    elevation_radians = np.arctan2(delta_z, np.sqrt(delta_x ** 2 + delta_y ** 2))
    elevation_degrees = np.degrees(elevation_radians)

    # Assign new columns to the DataFrame
    df['Distance'] = distance
    df['Azimuth_rad'] = azimuth_radians
    df['Azimuth_deg'] = azimuth_degrees
    df['Elevation_rad'] = elevation_radians
    df['Elevation_deg'] = elevation_degrees

    return df

def unify_CalibrationTargetId(df):
    df = df.copy()
    df['TargetId'] = 3 # rechts
    df.loc[df['CalibrationTargetId'].isin([7, 1]), 'TargetId'] = 1 # Links unten
    df.loc[df['CalibrationTargetId'].isin([4, 5, 6]), 'TargetId'] = 2 # Links oben
    return df

def select_n_evenly_distant_points_kmeans( df:pd.DataFrame, n, columns = ['Sun_Azimuth_deg', 'Sun_Elevation_deg']):
    def kmeans(group: pd.DataFrame, n_val, columns):
        data = group[columns].values
        kmeans = KMeans(n_clusters=n_val, random_state=0, n_init='auto').fit(data)
        centroids = kmeans.cluster_centers_
        distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
        selected_indices = np.argmin(distances, axis=0)
        return group.iloc[selected_indices[:n_val]]

    df = df.copy()
    df = df.reset_index(drop=True)
    df = df.groupby('HeliostatId').apply(lambda group: kmeans(group, n, columns))
    df = df.droplevel(0)
    df = df.reset_index(drop=True)

    return df

def select_heliostats_based_on_calibration_count_and_target(df, minCount, minTargetCount, TargetId):
    df = df.copy()
    id_counts_non_target = df[df['TargetId'] != TargetId]['HeliostatId'].value_counts().reset_index()
    id_counts_non_target.columns = ['HeliostatId', 'NonTargetCount']
    id_counts_target = df[df['TargetId'] == TargetId]['HeliostatId'].value_counts().reset_index()
    id_counts_target.columns = ['HeliostatId', 'TargetCount']
    merged_counts = pd.merge(id_counts_non_target, id_counts_target, on='HeliostatId', how='inner')
    filtered_ids = merged_counts[(merged_counts['NonTargetCount'] >= minCount) & (merged_counts['TargetCount'] >=
          minTargetCount)]['HeliostatId']
    filtered_df = df[df['HeliostatId'].isin(filtered_ids)]

    return filtered_df

def sample_equal_clusters(data, n, y, azimuth_col='Sun_Azimuth_rad', elevation_col='Sun_Elevation_rad',
      heliostat_col='HeliostatId'):
    data = data.copy()
    data['selected'] = np.nan

```

```
    # Function to apply to each Heliostat group
    def sample_group(group):
        # Sort the group by azimuth and elevation
        group = group.copy()
        group['original_index'] = group.index
        group = group.sort_values(by=[azimuth_col, elevation_col])
        # Partition the group into 'n' equal-sized clusters
        group['cluster'] = (np.arange(len(group)) % n)
        # Now sample one point from each cluster for each set
        for set_number in range(y):
            sampled_group = group.groupby('cluster').apply(lambda x: x.sample(1))
            sampled_group = sampled_group.reset_index(drop=True)
            # Set the 'selected' column to the current set number using the original index
            data.loc[sampled_group['original_index'], 'selected'] = set_number
            # Remove the selected points from the group using their original indices
            group = group.drop(sampled_group['original_index'])
        return group

    # Group by HeliostatId and apply the sampling function to each group
    data.groupby(heliostat_col, group_keys=False).apply(sample_group)
    data['selected'] = data['selected'].astype(int)
    return data


def filter_heliostats_by_target_counts(df, Target1Count, Target2Count, Target3Count):
    df = df.copy()
    target_counts = df.groupby(['HeliostatId', 'TargetId']).size().unstack(fill_value=0)

    # Filter heliostats based on counts
    valid_heliostats = target_counts[
        (target_counts.get(1, 0) >= Target1Count) &
        (target_counts.get(2, 0) >= Target2Count) &
        (target_counts.get(3, 0) >= Target3Count)
        ].index

    # Return the filtered rows
    return df[df['HeliostatId'].isin(valid_heliostats)]

def prepare_df():
    path = '/Data/calib_data.csv'
    path_Heliostats = '/Data/Heliostatpositionen_xyz.csv'
    df_og = dl.load_dataset(path)
    df_heliostat_og = pd.read_csv(path_Heliostats, sep=';')
    df = df_og.copy()
    df_heliostat = df_heliostat_og.copy()
    df_heliostat['HeliostatId'] = df_heliostat['HeliostatId'].astype(int)
    df_heliostat = df_heliostat[['HeliostatId', 'InternalName', 'x', 'y', 'z']]
    df = unify_CalibrationTargetId(df)
    df = pd.merge(df, df_heliostat, on='HeliostatId', how='left')
    df = add_aim_vector_to_df(df)
    df = add_normal_vector_to_df(df)
    df = add_sun_vector_to_df(df)
    return df

def make_dataframe_with_one_target_as_val(df, target_id):
    df = df.copy()
    df = select_heliostats_based_on_calibration_count_and_target(df, 100, 5, target_id)
    n_heliostats = df['HeliostatId'].nunique()
    print(f'Found {n_heliostats} Heliostats with more than 100 Calibrations')
    df_calib = select_n_evenly_distant_points_kmeans(df.loc[df['TargetId'] != target_id], 100)
    df_target = select_n_evenly_distant_points_kmeans(df.loc[df['TargetId'] == target_id], 5)
    df_target['selected'] = -1
    df_calib = df_calib.sort_values(by='CreatedAt_unix', ascending=True)
    df_calib = sample_equal_clusters(df_calib, 25, 4)
    df_target['selected'] = -1
    df_filtered = pd.concat([df_calib, df_target], )
    df_filtered['selected'] = df_filtered['selected'] + 1
    return df_filtered


def sample_dataset(df, dataset_size, size_cluster):
    assert dataset_size % size_cluster == 0, 'Cluster size is no match for calibs per Heliostat'
    n_clusters = dataset_size // size_cluster
    df = select_n_evenly_distant_points_kmeans(df, dataset_size, ['Sun_Azimuth_rad', 'Sun_Elevation_rad'])
    sampled_df = sample_equal_clusters(df, size_cluster, n_clusters)
    return sampled_df
```

Listing A.3: **HeliostatDataModule.py**

```
import pandas as pd
import torch
import pytorch_lightning as pl
from torch.utils.data import DataLoader, TensorDataset
import Pipeline.Data_Processer as dpr

class HeliostatDataModule(pl.LightningDataModule):
    def __init__(self, dataframe:pd.DataFrame, batch_size, n_heliostats, features_pred, features_query, features_encoder,
          norm_method = 'minmax', norm_scope='Heliostat', eval=False):
        super().__init__()
        self.dataframe = dataframe
        self.batch_size = batch_size
```

```
        self.group_column = 'HeliostatId'
        self.n_heliostats = n_heliostats
        self.features_pred = features_pred
        self.features_query = features_query
        self.features_encoder = features_encoder
        self.norm_method = norm_method

        if norm_scope == 'Heliostat':
            self.global_norm_features = ['x', 'y', 'z', 'Distance', 'CreatedAt_unix', 'Month']
            self.group_norm_features = ['Norm_Azimuth_rad', 'Norm_Elevation_rad','Sun_Azimuth_rad', 'Sun_Elevation_rad',
                'Azimuth_rad', 'Elevation_rad','Sun_Azimuth_deg', 'Sun_Elevation_deg', 'Azimuth_deg',
                'Elevation_deg','Axis1MotorPosition', 'Axis2MotorPosition']
        else:
            self.global_norm_features = ['x', 'y', 'z', 'Distance', 'CreatedAt_unix', 'Month'] + ['Norm_Azimuth_rad',
                'Norm_Elevation_rad','Sun_Azimuth_rad', 'Sun_Elevation_rad', 'Azimuth_rad', 'Elevation_rad','Sun_Azimuth_deg',
                'Sun_Elevation_deg', 'Azimuth_deg', 'Elevation_deg','Axis1MotorPosition', 'Axis2MotorPosition']
            self.group_norm_features = []

    def setup(self, stage=None):
        sampled_df = self.dataframe.copy().reset_index()

        if self.norm_method == 'minmax':
            sampled_df = dpr.normalize_dataframe_minmax(sampled_df, 'HeliostatId',
                list(set(self.features_encoder)&set(self.group_norm_features)),
                list(set(self.features_encoder)&set(self.global_norm_features)), range(0,4))
            self.features_pred_norm_columns = [[f"{string}_min" for string in self.features_pred],[f"{string}_max" for string in
                self.features_pred]]
        else:
            sampled_df = dpr.normalize_dataframe_zscore(sampled_df, 'HeliostatId',
                list(set(self.features_encoder)&set(self.group_norm_features)),
                list(set(self.features_encoder)&set(self.global_norm_features)))
            self.features_pred_norm_columns = [[f"{string}_mean" for string in self.features_pred],[f"{string}_std" for string in
                self.features_pred]]

        df_val = sampled_df[sampled_df['selected'] == 0].copy()
        df_train = sampled_df[sampled_df['selected'] != 0].copy()

        self.df_train = df_train
        self.df_val = df_val

        train_tensors = self._make_tensors(self.df_train, [self.features_encoder, self.features_query, self.features_pred] +
            self.features_pred_norm_columns, self.group_column) #tuple with len(list) tensors(n_heliostats, len(df_train),
            features) with the specified features
        self.train_dataset = TensorDataset(*train_tensors)

        val_tensors = self._make_tensors(self.df_val, [self.features_encoder, self.features_query, self.features_pred] +
            self.features_pred_norm_columns, self.group_column) #tuple with len(list) tensors(n_heliostats, len(df_val),
            features) with the specified features
        mixed_val_tensors = self._make_tensors(self.df_train.loc[self.df_train['selected'] == 1], [self.features_encoder],
            self.group_column)
        val_tensors = val_tensors + mixed_val_tensors
        self.val_dataset = TensorDataset(*val_tensors)

        predict_tensors = self._make_tensors(self.df_val, [self.features_encoder, self.features_query, ['id']] +
            self.features_pred_norm_columns, self.group_column)
        predict_tensors = predict_tensors + mixed_val_tensors
        self.predict_dataset = TensorDataset(*predict_tensors)

    def _make_tensors(self, df: pd.DataFrame, feature_lists, group_column):
        if df.empty:
            return tuple([torch.zeros(self.n_heliostats, 818, 2)])
        # Group the data by the group_column
        groups = df.groupby(group_column)
        # List to store tensors for each feature list
        tensors = []
        for features in feature_lists:
            # List to store feature values for each group
            group_tensors = []
            for _, group in groups:
                select_values = group['selected'].unique()
                for selected in select_values:
                # Extract features and convert to tensor
                    feature_tensor = torch.tensor(group.loc[group['selected'] == selected][features].values, dtype=torch.float32)
                    group_tensors.append(feature_tensor)
            # Stack tensors along the first dimension (group dimension)
            stacked_tensor = torch.stack(group_tensors)
            # Add the stacked tensor to the list of tensors
            tensors.append(stacked_tensor)

        return tuple(tensors)

    def train_dataloader(self):
        return DataLoader(self.train_dataset, batch_size=self.batch_size, shuffle=True)

    def val_dataloader(self):
        return DataLoader(self.val_dataset, batch_size=self.n_heliostats, shuffle=False)

    def predict_dataloader(self):
        return DataLoader(self.predict_dataset, batch_size=self.n_heliostats, shuffle=False)
```

Listing A.4: **Helioformer.py**

```
import torch
import torch.nn as nn
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint, LearningRateMonitor
from pytorch_lightning.loggers import TensorBoardLogger
import Helioformer_model.utils.loss_functions as loss_functions
from Helioformer_model.blocks.learnable_feature_embedding_all import LearnedFeaturePositionalEncodingAll
from Helioformer_model.blocks.encoder_block2 import EncoderBlock
from Helioformer_model.blocks.decoder_block2 import DecoderBlock
from Helioformer_model.blocks.output_net_block import OutputBlock
import Helioformer_model.Data_Models.Transformer_DataModule_V14_Auto as DM
import Helioformer_model.utils.Schedulers as schedulers

data_dir = 'Helioformer/Checkpoints/'

class Encoder(nn.Module):
    def __init__(self, input_len, dim_model, num_heads, dim_feedforward, dropout, n_layers):
        super().__init__()
        self.embedding = LearnedFeaturePositionalEncodingAll(dim_model, input_len)
        self.layers = nn.ModuleList([EncoderBlock(dim_model=dim_model,
                                                  num_heads=num_heads,
                                                  dim_feedforward=dim_feedforward,
                                                  dropout=dropout)
                                     for _ in range(n_layers)])
        self.norm = nn.LayerNorm(dim_model)

    def forward(self, x):
        x = self.embedding(x)
        x = x.transpose(0, 1) # Transpose x to (seq_len, batch, features)
        for layer in self.layers:
            x = layer(x)
        return x

class Decoder(nn.Module):
    def __init__(self, input_len, dim_model, output_len, num_heads, dim_feedforward, dropout, n_layers, masking=False):
        super().__init__()
        self.masking = masking
        self.embedding = LearnedFeaturePositionalEncodingAll(dim_model, input_len)
        self.layers = nn.ModuleList([DecoderBlock(dim_model=dim_model,
                                                  dim_feedforward=dim_feedforward,
                                                  num_heads=num_heads,
                                                  dropout=dropout,
                                                  self_attn=masking)
                                     for _ in range(n_layers)])
        self.output_net = OutputBlock(dim_model, output_len, dropout)

    def forward(self, x, enc):
        x = self.embedding(x)
        x = x.transpose(0, 1)
        for layer in self.layers:
            x = layer(x, enc, tgt_mask=None)
        x = x.transpose(0, 1) # Revert transpose
        output = self.output_net(x)
        return output

class Transformer(nn.Module):
    def __init__(self, input_len_enc, input_len_dec, dim_model, output_len, num_heads, num_enc_layers, num_dec_layers,
                 dim_feedforward, dropout):
        super().__init__()
        self.Encoder = Encoder(input_len=input_len_enc, dim_model=dim_model, num_heads=num_heads,
                               dim_feedforward=dim_feedforward, dropout=dropout, n_layers=num_enc_layers)
        self.Decoder = Decoder(input_len=input_len_dec, dim_model=dim_model, output_len=output_len, num_heads=num_heads,
                               dim_feedforward=dim_feedforward, dropout=dropout, n_layers=num_dec_layers, masking=False)

    def forward(self, calibration, query):
        encoding = self.Encoder(calibration)
        x = self.Decoder(query, encoding)
        return x

class LightningTransformer(pl.LightningModule):
    def __init__(self,
                 epochs,
                 input_len_enc,
                 input_len_dec,
                 dim_model,
                 output_len,
                 num_heads,
                 num_enc_layers,
                 num_dec_layers,
                 dim_feedforward,
                 dropout,
                 norm_method,
                 predict_mode,
                 features_pred,
                 features_encoder,
                 features_query,
                 lossfunction,
                 encoder_len,
                 norm_scope,
                 lr=2e-4
```

```
                ):
        super().__init__()
        self.save_hyperparameters()
        self.model = Transformer(input_len_enc, input_len_dec, dim_model, output_len, num_heads, num_enc_layers,
                                 num_dec_layers, dim_feedforward, dropout)
        self.example_input_array = (torch.randn(1116, 20, input_len_enc), torch.randn(1116, 5, input_len_dec))
        self.criterion = nn.MSELoss() if lossfunction == 'MSE' else nn.L1Loss()
        self.helploss = nn.L1Loss()
        self.epochs = epochs
        self.lr = lr
        self.angle_deviation = loss_functions.AngularDeviationLoss(deg=False, denorm=False)
        self.norm_method = norm_method
        self.predict_mode = predict_mode
        self.norm_scope = norm_scope
        self.encoder_len = encoder_len

    def forward(self, calibration, query):
        # Forward pass through the Transformer model
        return self.model(calibration, query)

    def training_step(self, batch):
        # norm_tensor1 = mean or min, norm_tensor2 = std or max, depending on norm method
        calibration, query, target, norm_tensor1, norm_tensor2 = self._shuffle_batch(batch)
        calibration = calibration[:, :self.encoder_len, :]
        query = query[:, self.encoder_len:, :]
        target = target[:, self.encoder_len:, :]
        norm_tensor1 = norm_tensor1[:, self.encoder_len:, :]
        norm_tensor2 = norm_tensor2[:, self.encoder_len:, :]

        prediction = self.model(calibration, query)
        if self.norm_scope == 'Heliostat':
            prediction, target = self._denorm(prediction, target, norm_tensor1, norm_tensor2)
        loss = self.criterion(prediction, target)
        self.log('train_loss', loss, prog_bar=True, on_step=False, on_epoch=True)
        return loss

    def validation_step(self, batch):
        # norm_tensor1 = mean or min, norm_tensor2 = std or max
        calibration, query, target, norm_tensor1, norm_tensor2, mixed_calibration = batch
        mixed_calibration = mixed_calibration[:, :self.encoder_len, :]
        prediction = self.model(calibration, query)
        prediction, target = self._denorm(prediction, target, norm_tensor1, norm_tensor2)

        loss = self.criterion(prediction, target)
        self.log('val_loss', loss, prog_bar=True, on_step=False, on_epoch=True)

        if self.predict_mode == 'NV':
            deviation = self.angle_deviation(prediction, target) * 1000
            self.log('deviation', deviation, prog_bar=True, on_step=False, on_epoch=True)

        prediction = self.model(mixed_calibration, query)
        prediction, _ = self._denorm(prediction, prediction, norm_tensor1, norm_tensor2)
        loss_mixed = self.criterion(prediction, target)
        self.log('mixed_val_loss', loss_mixed, prog_bar=True, on_step=False, on_epoch=True)

        if self.predict_mode == 'NV':
            mixed_deviation = self.angle_deviation(prediction, target)
            mixed_deviation = mixed_deviation * 1000
            self.log('mixed_deviation', mixed_deviation, prog_bar=True, on_step=False, on_epoch=True)

        return loss

    def predict_step(self, batch):
        calibration, query, IDs_mixed, norm_tensor1, norm_tensor2, train_calibration = self._shuffle_batch(batch)
        train_calibration = train_calibration[:, :self.encoder_len, :]

        prediction_mixed = self.model(train_calibration, query)
        prediction_mixed, _ = self._denorm(prediction_mixed, prediction_mixed, norm_tensor1, norm_tensor2)

        calibration = calibration[:, :self.encoder_len, :]
        query = query[:, self.encoder_len:, :]
        IDs = IDs_mixed[:, self.encoder_len:, :]
        norm_tensor1 = norm_tensor1[:, self.encoder_len:, :]
        norm_tensor2 = norm_tensor2[:, self.encoder_len:, :]

        prediction = self.model(calibration, query)
        prediction, _ = self._denorm(prediction, prediction, norm_tensor1, norm_tensor2)

        return IDs, prediction, IDs_mixed, prediction_mixed

    def configure_optimizers(self):
        optimizer = torch.optim.RAdam(self.parameters(), lr=self.lr, weight_decay=1e-3, decoupled_weight_decay=True)
        scheduler = schedulers.LinearWarmupScheduler(optimizer=optimizer, max_iters=self.epochs)
        return {
            'optimizer': optimizer,
            'lr_scheduler': {
                'scheduler': scheduler,
                'interval': 'epoch', # 'step' means the scheduler is updated every step
                'frequency': 1
            }
        }

```

```
    def _denorm(self, prediction, target, norm_tensor1, norm_tensor2):
        if self.norm_method == 'minmax':
            denorm_target = target * (norm_tensor2 - norm_tensor1) + norm_tensor1
            denorm_prediction = prediction * (norm_tensor2 - norm_tensor1) + norm_tensor1
        else:
            denorm_target = target * norm_tensor2 + norm_tensor1
            denorm_prediction = prediction * norm_tensor2 + norm_tensor1
        return denorm_prediction, denorm_target

    def _shuffle_batch(self, batch):
        num_elements = batch[0].size(1)
        shuffle_idx = torch.randperm(num_elements)
        return tuple(tensor[:, shuffle_idx, :] for tensor in batch)


class LightningInputOptimizer(pl.LightningModule):
    def __init__(self,
                 encoder,
                 decoder,
                 parameters,
                 predict_mode,
                 lr=1e-3):
        super().__init__()
        self.predict_mode = predict_mode
        # Freeze encoder and decoder
        self.encoder = encoder
        self.decoder = decoder
        for param in self.encoder.parameters():
            param.requires_grad = False
        for param in self.decoder.parameters():
            param.requires_grad = False
        self.shuffleindex = [13, 8, 15, 23, 6, 10, 3, 0, 11, 16, 9, 19, 22, 4, 5, 20, 21, 17, 1, 18, 2, 12, 7, 14, 24]
        self.norm_method = 'sdasd'
        self.loss_to_monitor = 'loss'
        self.query_params = nn.Parameter(parameters)
        self.criterion = nn.L1Loss()
        self.helploss = nn.L1Loss()
        self.angle_deviation = loss_functions.AngularDeviationLoss(deg=False, denorm=False)
        self.angle_deviation_output = loss_functions.AngularDeviationLoss(deg=False, denorm=False, mean=False)
        self.lr = lr
        self.first = True

    def forward(self, calibration, query):
        encoding = self.encoder(calibration)
        decoded_output = self.decoder(query, encoding)
        return decoded_output

    def training_step(self, batch, batch_idx):
        calibration, query, target, ids, norm_tensor1_pred, norm_tensor2_pred, norm_tensor1_opt, norm_tensor2_opt =
              self._shuffle_batch(batch)
        calibration = calibration[:, :20, :]
        og_query = query[:, :, :2].clone()
        query[:, :, :2] = self.query_params
        model_output = self.forward(calibration, query)
        model_output_denorm, target_denorm = self._denorm(model_output, target, norm_tensor1_pred, norm_tensor2_pred)
        loss = self.criterion(model_output_denorm, target_denorm) * 10000
        query_params_denorm, og_query_denorm = self._denorm(self.query_params, og_query, norm_tensor1_opt, norm_tensor2_opt)

        if self.predict_mode == 'NV':
            cos_theta = self.angle_deviation(query_params_denorm, og_query_denorm)
            deviation = cos_theta * 1000
            self.log('nv_deviation', deviation, prog_bar=True, on_step=False, on_epoch=True)
            self.log('loss', loss, prog_bar=True, on_step=False, on_epoch=True)
        else:
            deviation = self.helploss(query_params_denorm, og_query_denorm)
            self.log('mp_deviation', deviation, prog_bar=True, on_step=False, on_epoch=True)
            self.log('loss', loss, prog_bar=True, on_step=False, on_epoch=True)

        if loss < 5 and self.predict_mode == 'NV':
            self.ids = ids
            self.output, _ = self._denorm(self.query_params, og_query, norm_tensor1_opt, norm_tensor2_opt)

        return loss

    def configure_optimizers(self):
        # Optimizer that optimizes the input calibration_params
        optimizer = torch.optim.RAdam([self.query_params], lr=self.lr)
        scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=20)
        return {
            'optimizer': optimizer,
            'lr_scheduler': {
                'scheduler': scheduler,
                'monitor': self.loss_to_monitor,
                'interval': 'epoch',
                'frequency': 1,
            }
        }
        return optimizer

    def _denorm(self, prediction, target, norm_tensor1, norm_tensor2):
        if self.norm_method == 'minmax':
            denorm_target = target * (norm_tensor2 - norm_tensor1) + norm_tensor1
```

```
            denorm_prediction = prediction * (norm_tensor2 - norm_tensor1) + norm_tensor1
        else:
            denorm_target = target * norm_tensor2 + norm_tensor1
            denorm_prediction = prediction * norm_tensor2 + norm_tensor1
        return denorm_prediction, denorm_target
    def _shuffle_batch(self, batch):
        return tuple(tensor[:, self.shuffleindex, :] for tensor in batch)


def train_Transformer(df,
                    lr,
                    features_pred,
                    features_query,
                    features_encoder,
                    n_heliostats,
                    batch_size,
                    max_epochs,
                    num_heads,
                    d_model,
                    dim_feedforward,
                    num_enc_layers,
                    num_dec_layers,
                    encoder_len,
                    predict_mode,
                    lossfunction,
                    gpus=0,
                    pretrained_path=None,
                    norm_method='zscore',
                    norm_scope='Heliostat'):

    data_module = DM.HeliostatDataModule(df,
                                    features_pred=features_pred,
                                    features_query=features_query,
                                    features_encoder=features_encoder,
                                    n_heliostats=n_heliostats,
                                    norm_method=norm_method,
                                    norm_scope=norm_scope,
                                    batch_size=batch_size,
                                    eval=False)

    logger = TensorBoardLogger("tb_logs11", name="Transformer", log_graph=True)

    # Set up checkpoints and learning rate monitoring
    checkpoint_callback = ModelCheckpoint(
        monitor='val_loss', # Monitor which loss
        dirpath=data_dir, # Directory to save checkpoints
        filename='best-checkpoint{epoch}', # Filename for the checkpoint
        save_top_k=1, # Save only the best model
        mode='min' # Mode 'min' to minimize the monitored metric
    )

    lr_monitor = LearningRateMonitor(logging_interval='step')

    if pretrained_path is not None:
        model = LightningTransformer.load_from_checkpoint(pretrained_path,
                                                    epochs=max_epochs,
                                                    dropout=0.0,
                                                    lr=lr,
                                                    lossfunction=lossfunction,
                                                    map_location=torch.device("cpu"))
    else:
        model = LightningTransformer(max_epochs,
                                input_len_enc=len(features_encoder),
                                input_len_dec=len(features_query),
                                dim_model=d_model,
                                output_len=len(features_pred),
                                num_heads=num_heads,
                                num_enc_layers=num_enc_layers,
                                num_dec_layers=num_dec_layers,
                                norm_method=norm_method,
                                predict_mode=predict_mode,
                                dim_feedforward=dim_feedforward,
                                dropout=0.1,
                                features_encoder=features_encoder,
                                features_pred=features_pred,
                                features_query=features_query,
                                encoder_len=encoder_len,
                                lossfunction=lossfunction,
                                norm_scope=norm_scope,
                                lr=lr)

    if gpus != 0:
        trainer = pl.Trainer(
            max_epochs=max_epochs,
            accelerator="gpu",
            devices=gpus,
            logger=logger,
            callbacks=[checkpoint_callback, lr_monitor],
            log_every_n_steps=1,
            check_val_every_n_epoch=5
        )
    else:
```

```
        trainer = pl.Trainer(
            max_epochs=max_epochs,
            accelerator="cpu",
            logger=logger,
            callbacks=[checkpoint_callback, lr_monitor],
            log_every_n_steps=1,
            check_val_every_n_epoch=5)

    torch.set_float32_matmul_precision('high')
    trainer.fit(model, data_module)
    version_number = trainer.logger.version
    best_model_path = checkpoint_callback.best_model_path
    model = LightningTransformer.load_from_checkpoint(best_model_path)
    prediction = trainer.predict(model, data_module)
    return prediction[0], version_number, best_model_path

def predict_Transformer(df, n_heliostats, pretrained_path, batch_size, gpus=0):
    model = LightningTransformer.load_from_checkpoint(pretrained_path,
                                                      map_location=torch.device('cpu'),
                                                      strict=False)

    data_module = DM.HeliostatDataModule(df,
                                         features_pred=model.hparams['features_pred'],
                                         features_query=model.hparams['features_query'],
                                         features_encoder=model.hparams['features_encoder'],
                                         n_heliostats=n_heliostats,
                                         norm_method=model.hparams['norm_method'],
                                         norm_scope=model.hparams['norm_scope'],
                                         batch_size=batch_size,
                                         eval=True)
    if gpus != 0:
        trainer = pl.Trainer(
            accelerator="gpu",
            devices=gpus,
        )
    else:
        trainer = pl.Trainer(
            accelerator="cpu",
        )

    prediction = trainer.predict(model, data_module)
    version_number = trainer.logger.version
    return prediction[0], version_number, model.hparams['features_pred']
```

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Essen, den 22. Februar, 2025