Thesis Dissertation

Data-driven Deep Reinforcement Learning for Agriculture

Constantinos Paphitis

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Data-driven Deep Reinforcement Learning for Agriculture

Constantinos Paphitis

Supervisor

Dr. Chris Christodoulou

A thesis submitted in partial fulfilment of the requirements for the award of a Bachelor's degree in Computer Science at the University of Cyprus

May 2024

Acknowledgements

Throughout the journey of my thesis project, I have been fortunate to receive an immense amount of support and guidance, which was pivotal in shaping the research and its outcomes. I would like to express my profound gratitude to Professor Chris Christodoulou and Dr. Vassilis Vassiliades for their invaluable contributions to my work. Their willingness to meet with me regularly online provided a framework of support that was crucial for my progress. They were always available to answer my questions promptly via email, providing clear, insightful responses that guided me through complex challenges and decision-making processes. I am deeply thankful for their mentorship and the significant role they played in my thesis project.

Summary

This thesis, titled "**Data-driven Deep Reinforcement Learning for Agriculture**" explores the application of advanced **Artificial Intelligence** (**AI**) techniques to optimize greenhouse management, specifically focusing on **Deep Reinforcement Learning** (**DRL**) methods. The research is motivated by the critical need to enhance food production in the face of a growing global population and the challenges posed by climate change. Greenhouses play a pivotal role in providing a controlled environment for crop cultivation, but their management involves complex decision-making to balance resource use and crop yield.

Greenhouses provide a stable environment for crop production, essential for meeting the food demands of an expected global population of nearly 10 billion by 2050. However, managing the greenhouse environment to optimize crop yield while minimizing resource use is challenging. Traditional methods rely heavily on human decision-making, which can be inefficient and lead to resource wastage. AI, particularly DRL, offers a solution by automating and optimizing these processes through continuous learning and adaptation based on real-time data from sensors.

The thesis builds on the **Autonomous Greenhouse Challenge** (**AGC**) **2nd Edition**, an international competition aimed at improving greenhouse vegetable production using AI. The competition tasked teams with developing AI algorithms to manage virtual greenhouses, balancing tomato yield and resource use. Data from this challenge, including environmental parameters and crop conditions, provides the foundation for this research.

The primary contribution of this thesis is the development of a deep reinforcement learning framework to optimize greenhouse operations. This involves creating an environment simulator based on historical data from the AGC. The simulator uses regression models to predict the impact of various actions on greenhouse conditions and crop parameters, enabling the training and evaluation of DRL algorithms without real-world trials.

The research focuses on two advanced DRL algorithms: **Proximal Policy Optimization** (**PPO**) and **REINFORCE**. These algorithms dynamically learn and optimize greenhouse

management strategies. PPO is known for its stability and efficiency, while REINFORCE, a policy gradient method, directly optimizes the policy function. Both algorithms are evaluated for their effectiveness in maximizing tomato yield and minimizing resource use.

The methodology involves several key steps, including data preprocessing, which entails cleaning and normalizing the dataset, handling missing values, and performing feature engineering to ensure the data is suitable for training models. An environment simulator is then developed using regression-based predictive models to simulate the greenhouse environment, predicting future states based on current conditions and actions. The DRL algorithms, PPO and REINFORCE, are implemented to train agents that interact with the environment simulator, learning optimal strategies to maximize rewards such as crop yield and quality.

The effectiveness of the DRL algorithms is evaluated through empirical testing within the simulated environment. The results indicate that DRL can significantly improve greenhouse management, leading to better crop yields and more efficient resource use compared to traditional methods. This research demonstrates the potential of DRL in transforming greenhouse agriculture by reducing reliance on human intervention and enhancing the sustainability and profitability of food production. The framework developed in this thesis can be adapted for other agricultural settings or broader domains requiring complex environmental control and resource management.

In conclusion, this thesis successfully applies DRL to greenhouse management and suggests future work could explore integrating additional environmental variables, testing other DRL algorithms, and applying the framework to different crops or agricultural environments. This research advances the field of AI in agriculture, providing a scalable approach to automating and optimizing food production systems.

Contents

Chapter 1: Introduction **1**
1.1 Problem Overview 1
1.2 Autonomous Greenhouse Challenge (AGC) 2nd Edition 2
1.3 Contribution 3

Chapter 2: Background **5**
2.1 Data Preprocessing 5
2.1.1 Min max scaling 6
2.1.2 Interpolation 6
2.1.3 IQR 7
2.2 Machine Learning 7
2.2.1 Supervised Learning 8
2.2.1.1 Regressors 8
2.2.1.2 Cross Validation 8
2.2.1.3 Metrics for Evaluation 9
2.2.2 Reinforcement Learning 11
2.2.2.1 Offline Reinforcement Learning 11
2.2.2.2 Deep Reinforcement Learning 12
2.2.2.3 Model-based Reinforcement Learning 12
2.3 Related previous work 13

Chapter 3: Methodology and Implementation **16**
3.1 Methodology 17
3.2 Data Overview 18
3.3 Data Preprocessing 20
3.3.1 Data Cleaning 20
3.3.2 Data Reduction 21
3.3.3 Data Expansion 22
3.3.4 Data Transformation 23

3.3.5 Feature Engineering 30
3.4 Definition of actions, state and reward 30
3.5 Environment Simulator 33
3.5.1 Regression-based Predictive Models 33
3.5.1.1 State Prediction Regression Models 36
3.5.1.2 Crop Parameters Prediction Regression Models 37
3.5.2 Creation of tf_agents py_environment 39
3.5.2.1 Initialization 39
3.5.2.2 Updating State 40
3.5.2.3 Updating Crop Parameters 42
3.5.2.4 Calculate Reward 42
3.5.2.5 Step function 43
3.6 Agent 44
3.6.1 PPO 44
3.6.2 REINFORCE 46
3.6.3 Training and evaluation 48

Chapter 4: Results **50**
4.1 Prediction Models 50
4.1.2 State Prediction Model 52
4.1.2 Crop Parameters Prediction Regression Model 55
4.1.3 Algorithm Justification 57
4.2 DRL algorithms 58
4.2.1 PPO 59
4.2.2 REINFORCE 63
4.2.3 Best algorithm 67

Chapter 5: Conclusion and Future Work **69**
5.1 Conclusion 69
5.2 Future work 71

Chapter 1

Introduction

1.1 Problem Overview 1

1.2 Autonomous Greenhouse Challenge (AGC) 2nd Edition 2

1.3 Contribution 3

1.1 Problem Overview

Greenhouses are pivotal for providing healthy and fresh food for our increasing global population, which is expected to reach nearly 10 billion by 2050. They play an important role in the quality and productivity of our food supply. The main benefit that comes with greenhouses is that they are irrespective of external weather, this means that they are unaffected by seasonal changes and unexpected weather conditions which have been getting worse due to climate change. This allows cultivation of crops throughout the year and ensures a steady supply of food.

However, the management of resources and climate control presents significant challenges. Bad control of these components can lead to wasteful practices and inefficient crop production. Thus, we cannot depend entirely on human decision making but we need a reliable system that will produce more crops with better quality using less resources.

To address these issues, Artificial Intelligence (AI) and advanced technologies can monitor with the help of sensors and optimize environmental parameters like temperature, humidity, CO2 levels and soil moisture, ensuring plant growth conditions with minimal resources like water, nutrients and energy**[1]**. This way, decision support systems can analyze data from

sensors and provide recommendations about which actions the farmers should make, or even develop systems that are fully automated and independent from human interaction.

There have been many attempts to integrate Artificial Intelligence (AI) into greenhouses**[2]**. In these attempts the aim was to make an AI system which minimizes resource use and maximizes crop yield and quality while handling the greenhouse management. Therefore, there have been numerous competitions around the world regarding AI in agriculture and one of them was the Autonomous Greenhouse Challenge (AGC) – 2nd Edition**[3]**.

1.2 Autonomous Greenhouse Challenge (AGC) 2nd Edition

The Autonomous Greenhouses 2nd Edition, hosted by Wageningen University & Research (WUR), was an international competition aimed at advancing greenhouse vegetable production through the use of artificial intelligence (AI). The hackathon was held from December 2019 to May 2020, with 21 participating teams. The teams were tasked with developing AI algorithms to manage virtual greenhouses, balancing tomato yield and resource usage like energy and water to maximize profit.

The dataset for the challenge was created by collecting data from a six-month cherry tomato production period in high-tech glasshouse compartments at WUR's research center in Bleiswijk, the Netherlands. The top 5 teams were selected for the growing experiment: AICU, DIGILOG, IUA.CAAS, The Automators, and Automatoes.

The selected teams implemented their AI strategies in real greenhouses at WUR's facilities in Bleiswijk, the Netherlands, over a six-month period that had actual cherry tomato crops. These teams remotely managed various greenhouse conditions such as temperature, light levels, CO2 concentrations, and cultivation-related parameters like plant density and stem density. The goal of the competition was to optimize the net profit of grown cherry tomato crops using minimal resources (water, energy, CO2, nutrients) while maximizing crop production and quality.

During the growing experiment, the AI algorithms could adjust greenhouse set points based on real-time data from the greenhouse environment. At the end of the competition, the net

profit of each team was assessed and used to evaluate the teams. The winning team, Automatoes, achieved the highest net profit of €6.86 per square meter.

1.3 Contribution

This thesis contributes to the field of greenhouse agriculture and artificial intelligence by advancing the capabilities of automated systems to manage greenhouse environments more effectively. Given the critical role that greenhouses play in food production amidst growing global demands and climatic challenges, the necessity for improved automation and decision-making systems is evident. My research directly addresses this need by implementing a deep reinforcement learning-based framework capable of simulating and optimizing greenhouse operations to enhance crop yield and resource efficiency.

My work extends the current understanding of AI applications in greenhouse management by developing a sophisticated environment simulator. This simulator uses historical data from the Autonomous Greenhouse Challenge to recreate the dynamic conditions of a real greenhouse. By integrating detailed regression models that predict the impact of various environmental parameters and management actions, the simulator provides a robust platform for testing and refining AI-driven strategies without the need for live trials.

The core of my contribution lies in the application of two advanced deep reinforcement learning algorithms, Proximal Policy Optimization (PPO) and REINFORCE. Unlike traditional approaches that might rely on static decision rules or supervised learning models, these algorithms offer a dynamic learning capability. They adapt and optimize decision-making processes based on the simulated feedback, enabling continuous improvement in strategies aimed at maximizing tomato yield and minimizing resource use.

Furthermore, I have empirically evaluated the effectiveness of these algorithms within the simulated environment. By plotting the cumulative rewards during both training phases and post-training evaluations in controlled test scenarios, the research assesses the practical viability of each algorithm. This not only demonstrates which algorithm performs better under specific conditions but also provides insights into how different strategies can be adjusted to improve greenhouse management.

The practical implications of this research are significant. By proving that deep reinforcement learning can effectively optimize greenhouse environments in simulation, this work paves the way for real-world applications where such AI systems can manage actual greenhouses. This contributes to reducing the reliance on human intervention, decreasing resource wastage, and potentially improving the overall profitability and sustainability of greenhouse agriculture.

Finally, my research contributes to both academic knowledge and practical applications by detailing the processes and methodologies involved in setting up, training, and evaluating AI systems within an agricultural context. It offers a comprehensive framework that other researchers and practitioners can replicate or adapt for similar challenges in different agricultural settings or even broader domains where environmental control and resource management are critical. This thesis not only advances the technical knowledge in applying AI to solve real-world problems in sustainable agriculture but also demonstrates a scalable approach to deploying intelligent systems in environments that are typically challenging to automate.

Chapter 2

Background

2.1 Data Preprocessing 5

2.1.1 Min max scaling 6

2.1.2 Interpolation 6

2.1.3 IQR 7

2.2 Machine Learning 7

2.2.1 Supervised Learning 8

2.2.1.1 Regressors 8

2.2.1.2 Cross Validation 8

2.2.1.3 Metrics for Evaluation 9

2.2.2 Reinforcement Learning 11

2.2.2.1 Offline Reinforcement Learning 11

2.2.2.2 Deep Reinforcement Learning 12

2.2.2.3 Model-based Reinforcement Learning 12

2.3 Related previous work 13

2.1 Data Preprocessing

Data preprocessing involves the preparation and transformation of raw data into a suitable format for the training of a model. It is a critical step for the functioning and performance of my regressor models. This process includes the cleaning of data, getting rid of irrelevant features, handling missing values, removing outliers, normalizing and encoding. A thorough data preprocessing can help reduce model overfitting, improve model accuracy and help the model generalize for potential new unseen data.

2.1.1 Min max scaling

Min max scaling is a very common normalizing technique in data preprocessing for the main reason that it scales all features to the range of [α, β], typically [0, 1]. This can be really helpful for data analyzation to get a better understanding of each feature's distribution and for outlier detection. Additionally, it helps the models come to a faster convergence because the weight adjustments during training will be smaller.

The formula for the min max scaling Is:

$$\mathrm{X}' = \alpha + \frac{\mathrm{X} - \mathrm{Xmin}\ \times\ (\beta - \alpha)}{\mathrm{Xmax} - \mathrm{Xmin}},$$

where **X** is the feature being scaled, **[α, β]** is the new range of the scaled feature **X'** and **Xmin** and **Xmax** are the minimum and maximum values of the feature **X**.

2.1.2 Interpolation

Interpolation is a mathematical technique used to estimate unknown values that fall within the range of a set of known data points. It is a popular technique in the data processing field to produce new data points within the range of a set of known data points. By applying interpolation, it passes through known points and computes the new values with an interpolating function.

The most common interpolating function is linear interpolation that assumes the change from one value to another is linear, for example in time series or other sequences that follow a linear pattern. The linear interpolation is described by the formula:

$$y = y_1 + \frac{(\mathrm{x} - \mathrm{x}_1)(\mathrm{y}_2 - \mathrm{y}_1)}{(\mathrm{x}_2 - \mathrm{x}_1)},$$

where **(x1, y1)** and **(x2, y2)** are known data points and **y** is the interpolated value at a new point **x**.

Interpolation is a good method to fill gaps in data or smooth the existing data, but it must be used carefully as it assumes that the unknown data points between two known ones can calculated with the chosen interpolated function, which in reality may not be true.

2.1.3 IQR

The Interquartile Range (IQR) is a measure of statistical dispersion, representing the difference between the **25th (Q1)** and **75th (Q3)** percentiles of the data. This way, it captures the middle 50% of the data, which constitutes the summary of the data's distribution without being influenced by the outliers.

The IQR is described by the simple formula:

$$\mathrm{IQR} = Q3 - Q1,$$

where values below $Q1 - 1.5 \times IQR$ and above $Q3 + 1.5 \times IQR$ are considered outliers.

The IQR is a popular technique because it is less sensitive to extreme outliers than other dispersion measures and also it does not assume a specific distribution of the data making it a versatile took for summarizing data and identifying outliers. Thus, IQR is a powerful tool for statistical analysis and for data preprocessing before training machine learning algorithms.

2.2 Machine Learning

Machine Learning (ML) falls under the broad category of AI, and it involves the study of algorithms and statistical models that computer systems perform without explicit instructions, relying on patterns and inference derived from data. Machine Learning first appeared back in the mid-20th century but has just recently gained popularity due to its rapid evolution which was caused by the increased computational power and expansion of data availability over the years. It includes a variety of techniques that are categorized to supervised learning, unsupervised learning and reinforcement learning.

2.2.1 Supervised Learning

Supervised learning is a category of ML techniques for models that learn to make predictions with a dataset that the expected output is known (labelled data), hence the name supervised. Therefore, supervised learning can be used only when both the input and output are known, and the model is trying to find a relationship between them. The algorithms tend to be relatively fast and very effective because they utilize all the information available to them and they try to minimize the error between the expected and the model output. However, it is up to us to prevent the models from overfitting to the training dataset or get stuck to a local minimum of the error function. This approach is primarily used to solve two types of problems: classification and regression.

2.2.1.1 Regression

Regression is a statistical method that belongs to the supervised learning category, and it involves fitting a line or curve that best captures the relationship between input and output values to predict a continuous dependent variable. The objective of the fitting is to minimize the loss function that we define by adjusting the parameters (weights). The fitting process continues until it reaches convergence, i.e. minimal difference of loss function between iterations or other criterion that we have set.

2.2.1.2 Cross Validation

Cross Validation is a popular evaluation technique that indicates the effectiveness of machine learning models, primarily the ones that use supervised learning algorithms. It ensures that the model does not overfit to the training set, but it can generalize to new unseen data. It involves dividing the data to k folds (k is set by us), where the model is trained on the k-1 subsets and the remaining set is used as the test set. This process is repeated k times so that each subset serves as a test set only once.

CV can be really helpful in cases where the dataset is too small and splitting the data to training and testing sets may be complicated and there is not enough data for a validation set or when bad splits might lead to overfitting. Therefore, in these cases, CV provides a more reliable estimate of model performance.

2.2.1.3 Metrics for Evaluation

To evaluate the performance of a machine learning model that uses a supervised learning algorithm, we calculate some metrics that indicate how good the predictions were in respect to the expected outputs.

For Regression Models, common metrics are:

1. Mean Absolute Error (MAE): This measures the average magnitude of the errors in a set of predictions, without considering their direction. It's the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight. It is described by the formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - y'_i|$$

2. Mean Squared Error (MSE): MSE is a measure of the quality of an estimator which is always non-negative, and values closer to zero are better. It's calculated as the average of the squares of the errors, i.e. the average squared difference between the estimated values and what is estimated described as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - y'_i)^2$$

3. Root Mean Squared Error (RMSE): RMSE is the square root of the mean of the squared errors. RMSE is a good measure of how accurately the model predicts the response, and it is the most appropriate when large errors are particularly undesirable. It is basically the root of the MSE metric:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - y'_i)^2}$$

4. R-squared (Coefficient of Determination): This metric provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of variance explained. The formula is:

$$R2 = 1 - \frac{\sum_{i=1}^{n}(y_i - y'_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2},$$

where $\bar{y}$ is the mean of the actual values y.

R-squared values vary from 0 to 1, where:

- 0 indicates that the model explains none of the variability of the response data around its mean.
- 1 indicates that the model explains all the variability of the response data around its mean.

Higher R-squared value means that it is a better fit and that the model can account for the variation in the dependent variable using the independent variables. For example, if the R-squared score is 0.80 it means that 80% of the variance in the dependent variable (output) is predictable from the independent variables (input). Therefore, an R-squared score close to 1 means it has a high level of correlation between the model's predictions and the actual data.

This metric is very useful for model comparison on the same dataset where the model with the highest R-squared score fits the dataset better than the others. However, R-squared can be misleading if there are outliers in the dataset or if the model is overfitted.

Comparing metrics:

The R-squared score is scale-independent whereas the other metrics are affected by the scale of each feature because for example an MSE or MAE that have a score of 100 can be considered either very high or very low, depending on the scale of the dataset.

However, R-squared should not be the only metric we take for our models because the variance around certain values might be very little. In this scenario, if there is a high concentration around a certain value and the model predicts values that are not far off the expected output then the MSE and MAE could be relatively low, but the R-squared score might be low indicating that the model does not explain much of the variance but simply because there isn't much variance to explain.

MAE is often used when the distribution of errors is uniform or when large outliers are likely but should not dominate the error metric. MSE and RMSE are used when large errors are particularly undesirable and should be heavily punished (squared). Thus, if the distribution is expected to be gaussian, RMSE and MSE might be more appropriate because a few large errors can dominate the overall error metric, much like how a few large deviations from the mean significantly affect the shape and characteristics of a Gaussian distribution.

2.2.2 Reinforcement Learning

Reinforcement Learning (RL)**[4]** is an area of ML just like supervised learning. Unlike supervised learning, the model does not have the correct answers, but the model agent interacts with an environment though actions and it gets immediate feedback which can be either positive or negative.

During training, the agent in each timestep decides what action it will take based on the current state of the environment, then the environment changes its state accordingly if needed and provides the agent with a reward. After receiving it, the agent updates its policy based on the reward and the new state of the environment.

Concluding, RL is an autonomous, self-teaching system that essentially learns through trial and error. The goal of the agent is to learn the optimal policy through experience that will maximize the total reward in the specific environment through many interactions.

2.2.2.1 Offline Reinforcement Learning

Offline Reinforcement Learning(RL)**[5]**, also known as Batch Reinforcement Learning, is a variant of RL that effectively leverages large, previously collected datasets for large-scale real-world applications. The fact that it uses static datasets means that it does not perform any interaction and exploration as opposed to the online RL. The advantage of using offline RL on a fixed dataset is that it mitigates potential risks and costs by not requiring any additional exploration and does not require collecting new data.

Since the dataset is fixed and the interaction with the greenhouses is impossible, online RL can not be applied. Instead, offline RL is a good approach to this problem since it does not require any more data collection or real-time experimentation with the greenhouses, it is

designed for dynamic decision making tasks and it can learn policies that consider long-term consequences.

2.2.2.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL)**[6]** combines two powerful areas in Machine Learning, Deep Learning and RL. Deep Learning is used in DRL to approximate functions such as the policy of choosing an action and the value function which estimates future rewards. DRL is more suitable than traditional RL for complex situations where the state spaces are high-dimensional (for example input from sensors) or when the action space is continuous and not limited and traditional RL methods cannot handle them.

However DRL requires large amounts of data and significant computational power for training so that it performs well and does not overfit. This means that the agent needs to extensively interact with the environment which might not be possible and very costly. In addition, DRL is more prone to overfit due to the lack of diversity in the environment or if the model it too complex.

There are three main approaches in which DRL algorithms approach learning:

1. **Value-Based methods** which focus on optimizing the value function.
2. **Policy-Based methods** that directly optimize the policy function, which methods are useful for continuous action spaces.
3. **Actor-Critic methods** which combine features of both policy-based and value-based methods.

2.2.2.3 Model-based Reinforcement Learning

Model-based reinforcement learning (RL)**[7]** is a subset of reinforcement learning techniques where the agent explicitly constructs a model of the environment which works as an environment simulator. Basically, the model learns how the environment responds to actions taken by the agent which includes:

- **State transitioning**: A model that predicts the changes on the state of the environment based on the actions and the previous state of the environment.

- **Reward function**: A model that predicts the reward received for the agent's action on a given state.

The primary advantage of model-based RL over model-free RL is that it requires fewer interactions with the actual environment to learn policies, whereas model-free need extensive interaction with it during training which can be costly and sometimes infeasible.

For model-based RL methods, sufficient initial interactions with the actual environment are typically necessary for the environment simulator to be built. Once it is built, the algorithms can interact solely with the environment simulator as it has inherited the actual environment's behaviour, making the process more efficient and less costly. However, it is crucial that the simulator is correctly representing the environment so that the policies learned are also effective when applied to the actual environment.

2.3 Previous work

Before I started developing my own model for the AGC 2nd Edition dataset, I have done research on implementations from past papers on the subject to help me structure my own methodology and get some inspiration.

Deep Reinforcement Learning for Agriculture: Principles and Use (2022)**[8]**:
In this paper, the objective was to train a DRL agent to mimic the teams' actions with Deep Q-Network on the AGC 2nd Edition dataset. Thus, an assumption was made that the teams have performed the best possible actions for each timestep.

For each team there is a dataset of 47,000 rows, since there are 5 teams the agent was trained for 5 episodes. In each episode there were 47,000 timesteps where the agent had to guess the action taken by the according team at the current timestep. If the prediction was equal to the team's action, then a positive reward was given to the agent, otherwise a punishment was given instead. Rewards and punishments were calculated by the difference of the prediction and actual action at the current timestep.

This approach utilizes popular DRL methods and has a straightforward reward function, but the agent cannot overcome the teams' performance because it tries to mimic them.

Data-driven Deep Reinforcement Learning for Agriculture – CYENS Internship report (2022)**[9]:**

Garazhian's goal in his work was to outperform the winning team of the AGC 2nd Edition dataset by enhancing production levels, but completely ignoring the cost and the amount of the resources the algorithm uses. For the experiments many offline RL algorithms were used to train the model and then they were compared based on the percentage of improvement of the reward function over the winning team.

For the training, the extracted data used was the GreenhouseClimate.csv which included various parameters of the greenhouse that were captured by sensors with a 5-minute interval and the CropParameters.csv that consisted of weekly crop parameters which indicated the growth of the stem. In the data preprocessing, the raw data from the sensors was averaged and aggregated to match the weekly data of the crop parameters. For the evaluation of models, the data for the winning team ("Automatoes") was excluded from the training set.

The input for the RL algorithms was separated to two sets of action and state. The criteria for this separation is if a factor was directly under our control then the factor is considered action, otherwise it is considered as part of the state. Additionally, set points were completely excluded from the data since realized points were included.

As for the reward, Garazhian encountered the problem of reward sparsity. Lab results of the tomatoes and their quality testing were gathered every two weeks. Therefore, including these in the reward function would delay learning, hinder exploration and potentially have even less samples because it is difficult to delegate reward to a series of actions. The solution to this problem was to consider including crop parameters to the reward function as they are associated with the crop's wellness, and they are highly correlated with the tomato's quantity and quality but also have a 1-week interval instead of 2.

The results of his work have shown 4-5% improvement in cumulative weighted rewards (over 16 timesteps) compared to the best-performing team. However, it is questionable if the prediction of the crop parameters is valid, if the weights of the reward function are correct and if the algorithms had overfit to the small weekly dataset.

Deep reinforcement learning for improving competitive cycling performance (2022)**[10]**:
This paper is about developing a model with AI algorithm that recommends actions to the competitive cyclists that will correct their posture while cycling. For this system, a combination of model-based RL and DRL.

For the creation of the dataset, sensors were placed on cyclists' back creating a dataset of 14 different bicycle rides varying in duration, parameters of the cyclist (age, weight and height), bike parameters and many other factors. For practical reasons, recommendations could not take place every second, so the dataset was scaled down to every 5 seconds by averaging.

After gathering all the data needed, the data was processed, new features were added, and all features were analysed as to how correlated they were with the speed of the cyclist. Once it was decided which features would be used, regressor models were built to predict target variables such as speed and heart rate and other target variables.

The environment simulator consists of multiple predictive models and when the agent takes an action it creates the next state. Based on the state, the environment gives the appropriate reward which allows for the model to train and learn the optimal observation-action pair. The goal of the agent is to optimize the ride's average speed without straining the cyclist.

From this work, I have inherited the idea of creating an environment simulator and using model-based RL methods with a DRL agent for policy learning.

Chapter 3

Methodology and Implementation

3.1 Methodology 17
3.2 Data Overview 18
3.3 Data Preprocessing 20
3.3.1 Data Cleaning 20
3.3.2 Data Reduction 21
3.3.3 Data Expansion 22
3.3.4 Data Transformation 23
3.3.5 Feature Engineering 30
3.4 Definition of actions, state and reward 30
3.5 Environment Simulator 33
3.5.1 Regression-based Predictive Models 33
3.5.1.1 State Prediction Regression Models 36
3.5.1.2 Crop Parameters Prediction Regression Models 37
3.5.2 Creation of tf_agents py_environment 39
3.5.2.1 Initialization 39
3.5.2.2 Updating State 40
3.5.2.3 Updating Crop Parameters 42
3.5.2.4 Calculate Reward 42
3.5.2.5 Step function 43
3.6 Agent 44
3.6.1 PPO 44
3.6.2 REINFORCE 46
3.6.3 Training and evaluation 48

3.1 Methodology

Methodology Introduction:

The main purpose of this research is to develop an advanced decision-making system for greenhouse management based on DRL and model-based RL. Several studies have explored the application of DRL in agricultural settings, particularly focusing on irrigation scheduling to enhance water use efficiency**[11]**. The primary function of this system is to maximize the production of tomato crops under controlled environments and to outperform human-operated systems in terms of efficiency and output. The basic approach consists of building an environment simulator, which is a virtual environment where different DRL models can interact during the training process and simulation. This type of environment can be allowed in discrete action spaces of any complexity.

Data Preprocessing for Environment Simulator:

The first step in the development of an effective environment simulator is to prepare the necessary data. In the current study, the dataset for the Autonomous Greenhouse Challenge has been cleaned, preprocessed, and analysed to ensure that the data collected is both relevant and accurate for the simulation. The preprocessing of the data included the cleaning of the information collected, removal of missing values, and correction of anomalies through interpolation. The process was vital since a complete and relevant dataset helps in generating accurate models through the simulation. Feature engineering was another critical aspect of this process, and it involves the creation of new variables based on selected factors. All the new variables were numerical in nature and were added to provide a better perspective on the flow of time and simulate the seasonality.

Creation of the Environment Simulator:

This imitates the actual conditions within a greenhouse utilizing the pre-compiled data, thus allowing for the training and optimizing of the decision-making processes of the RL agents. Two regression models were included in the simulator, one for forecasting future states of the greenhouse's environment, and the other for future crop parameter value. The models receive current state and action from the agent and provide predictions. This component is crucial to the simulator as it enables it to accurately recreate the impact of various actions on the health of crops and resource utilization. This was then combined into a DRL setup that

allowed RL agents to engage with the dynamic and responsive virtual environment. This inclusion is a significant step in training the agents in varying conditions while at the same time experiencing real-time decision-making within a simulated setting.

Training and Evaluation of DRL Models:

In the experiments two main DRL algorithms were utilized, Proximal Policy Optimization (PPO) and REINFORCE. PPO is stable and efficient in optimizing policy networks that determine the course of action based on the state of the environment. On the other hand, REINFORCE provides a different approach to learning, focusing on policy gradients and, as such, allows study into RL learning behaviour under the same conditions. Training is carried out with the simulator to provide feedback in the form of rewards based on the performance impacts on yield or resource use. Both algorithms were evaluated by plotting the cumulative rewards during training, as well as during policy assessments in a post-training test environment.

Results Comparison:

This theoretical effectiveness of PPO over REINFORCE was further explored through a comparative analysis to examine which of the two alternatives positively influenced optimized greenhouse management. The comparative analysis was vital in determining the more efficient learning and decision-making algorithm within the simulated environment, and theoretically, it is PPO.

3.2 Data Overview

The data from the AGC 2nd Edition contains a folder for each team, a folder named “Weather”, which has data about the weather outdoors and it was collected by the weather station, a folder named “Reference” that included data represented by a group of Dutch commercial growers and was used to compare with the AI-controlled compartments and a pdf named “Economics”. The pdf explained how the overall profit is calculated by giving crops a price estimate based on the tomato’s category (A full price, B half price and the Unsellable category) and deducting the costs of the resources consumed. As the outdoor conditions were mutual to all, but also irrelevant to the performance of each team, the “Weather” folder was ignored as well as the “Reference” folder and worked only with the

teams' folders. Furthermore, the pdf was not taken into account since the goal of this thesis is to improve crop's production and quality and not to maximize the net profit. Inside each team's folder there are the following csv files:

- **CropParameters**: Contains characteristics of the crop that were observed such as stem growth, stem thickness etc.
- **Resources**: Includes data about the resources used daily in the greenhouse such as water, electricity and CO2.
- **GreenhouseClimate**: Indoor climate, status of actuators and irrigation
- **Production**: how many kg per m2 crop production of class A (valued with full price) and class B (valued with half price) and other parameters about the crop harvested.
- **TomQuality**: Assessment of tomatoes based on different parameters like flavour, weight, acid etc.
- **LabAnalysis**: Results from lab tests on tomatoes.
- **GrodanSens**: Metrics from sensors for slab water content, slab temperature and electrical conductivity.

Since the goal of the thesis is just to improve crop production and quality and not to minimize resources cost, the Resources csv file will not be used for now but it could be used in future work. Moreover, Production csv file is useful to calculate net worth which also doesn't align with the thesis' goal. TomQuality and LabAnalysis were too sparse to include to my dataset since their record interval was 2 weeks. Lastly, in the GrodanSens csv file there are some metrics of the water which are independent of the greenhouse and our main focus is to create an AI system for an automated greenhouse. Thus, the only csv files that were used for the final dataset are **GreenhouseClimate** and **CropParameters**.

The GreenhouseClimate.csv consists of 50 features and 47810 entries. The entries have a 5-minute interval and the competition took 24 weeks, by doing the calculations this should indeed give over 47k rows of raw data. This dataset has been separated to states and actions, the actions are the parameters that were completely under our control (and served as actions) and the rest were considered as part of the state. This way we avoid overfitting and we can proceed with RL.

The CropParameters.csv contains five crop characteristics out of which three are used in the reward function: Stem thickness, Stem growth and Cumulative number of new trusses on the stem. All three parameters are correlated with the plant's health and quality and tomato quantity. Unfortunately, there are only 24 records of these parameters as they were taken weekly. The two features that were not included in the reward function were registered by the teams and for most weeks there were no changes in the values.

Both GreenhouseClimate.csv and CropParameters.csv dataset descriptions can be found in the **Appendix A**.

3.3 Data Preprocessing

Data Preprocessing is a crucial part for building the regressor models for the environment simulator. It directly impacts the accuracy and efficiency of the predictive models. It is important that the environment simulator must not only be predictive but reliable too, making data preprocessing an indispensable part of the process. Preprocessing for State Prediction model in Appendix B1 and for Crop Parameters Prediction model in **Appendix B2**.

3.3.1 Data Cleaning

The first step of data preprocessing is cleaning the data. This involves handling missing values and dealing with errors and outliers in the data. After a thorough review of the GreenhouseClimate.csv, it was observed that there was a set of records had missing values across all features, so they were removed immediately. The rest of the missing values have been filled with the Interpolation method that is described in section **2.1.2**. The reason why Interpolation was applied is due to the fact that the data in the GreenhouseClimate.csv is a timeseries of greenhouse parameters that were recorded every 5 minutes. Thus, it is safe to assume that the relation between sequential values is linear and an easy way to fill these gaps was with a linear interpolation function.

Another method that could potentially be used to fill the gaps was by taking the average of neighboring values. However, in cases where there are many sequential values missing in a feature, averaging can be problematic for two reasons. The first reason is that if there are a lot of sequential values missing, then the averaging method would take two datapoints that are far from each other and produce a value that would not be realistic. The second reason is

that averaging would assign the same value to all the sequential missing values which may not accurately represent the variability of the data.

After the filling of the missing values, a post-interpolation cleaning was done to ensure that the calculated values were in the correct format that was expected for each feature. For example, some features had percentage values and others had binary values.

As for addressing the errors and outliers, a methodical approach was adopted to mitigate their impact on the model's performance. Firstly, with the help of Interquartile Range (IQR) method in section **2.1.3**, datapoints that deviate significantly from the central tendency of the data were pinpointed. By calculating the 75th percentile (Q3) and the 25th percentile (Q1) of the data, outliers were identified as any datapoints lying beyond 3 times the IQR above Q3 or below Q1. A smaller factor than 3 was not as effective because it would change the variance of the data a little too much. Instead of deleting the outliers, which could lead to a loss of valuable information and could mess up the sequence of the timeseries, a less aggressive adjustment was applied:

$$Lower\ bound = Q1 - 3\ \times IQR$$
$$Upper\ bound = Q3 + 3\ \times IQR$$

By defining these bounds, the outliers that exceeded them were adjusted by being assigned the corresponding upper or lower bound value.

This method proved effective for managing outliers in the dataset, as it allowed for adjustments without distorting the underlying variance and patterns, which is crucial for building a reliable and accurate environment simulator.

3.3.2 Data Reduction

The purpose of this step is to reduce the volume of data in the GreenhouseClimate.csv, without changing the integrity and variance of the original data. There are many reasons to why the data of this dataset was reduced:

a. The GreenhouseClimate.csv, as already mentioned, has 5-minute intervals between records which results to having over 47,000 rows of data for each team, whereas

CropParameters.csv has 1-week intervals and only 24 rows and some rows even have missing values. This would cause the problem of having sparse rewards and would be difficult to merge 24 rows with 47,000.

b. Recording every 5 minutes in a greenhouse is redundant since there cannot be much change in during those 5 minutes for the parameters to change significantly.

c. Raw data from sensors is often prone to containing outliers due to sensor malfunctions, environmental inference, or other anomalies that could interfere with the measuring of data. Thus, reducing the data with averaging could smoothen the data and reduce outliers.

d. Less data means faster and more meaningful learning for the environment simulator model, requiring less computational resources.

For these reasons, it was decided to reduce the GreenhouseClimate.csv dataset, but not over-reduce it because that would lead to underfitting and poor performance. Therefore, instead of having 5-minute intervals between records, the data was aggregated by averaging 12 sequential values to create each new record, resulting in hourly intervals. The final dataset had around 4000 entries for each team, and this is verified by multiplying 24 weeks with 168 hour per week and this gives us around 4000 hours.

3.3.3 Data Expansion

Unfortunately, CropParameters.csv has only 24 records which are weekly for each team. Down sampling GreenhouseClimate.csv to weekly records is not a good practice since this would leave us with a very small dataset to train the environment simulator.

Since CropParameters.csv contains parameters that contribute to the reward function, leaving the dataset as is would cause a credit assignment problem because one reward would correspond to 168 actions (168 hours per week). To avoid having this problem, the dataset was merged to the GreenhouseClimate.csv by attaching the values of CropParameters.csv to the records with the same timestep and then interpolated all the missing values between every known datapoint with linear Interpolation (**2.1.2**).

Since the parameters indicate how much a plant grows, it is safe to assume that the relation between sequential values of the parameters is linear.

3.3.4 Data Transformation

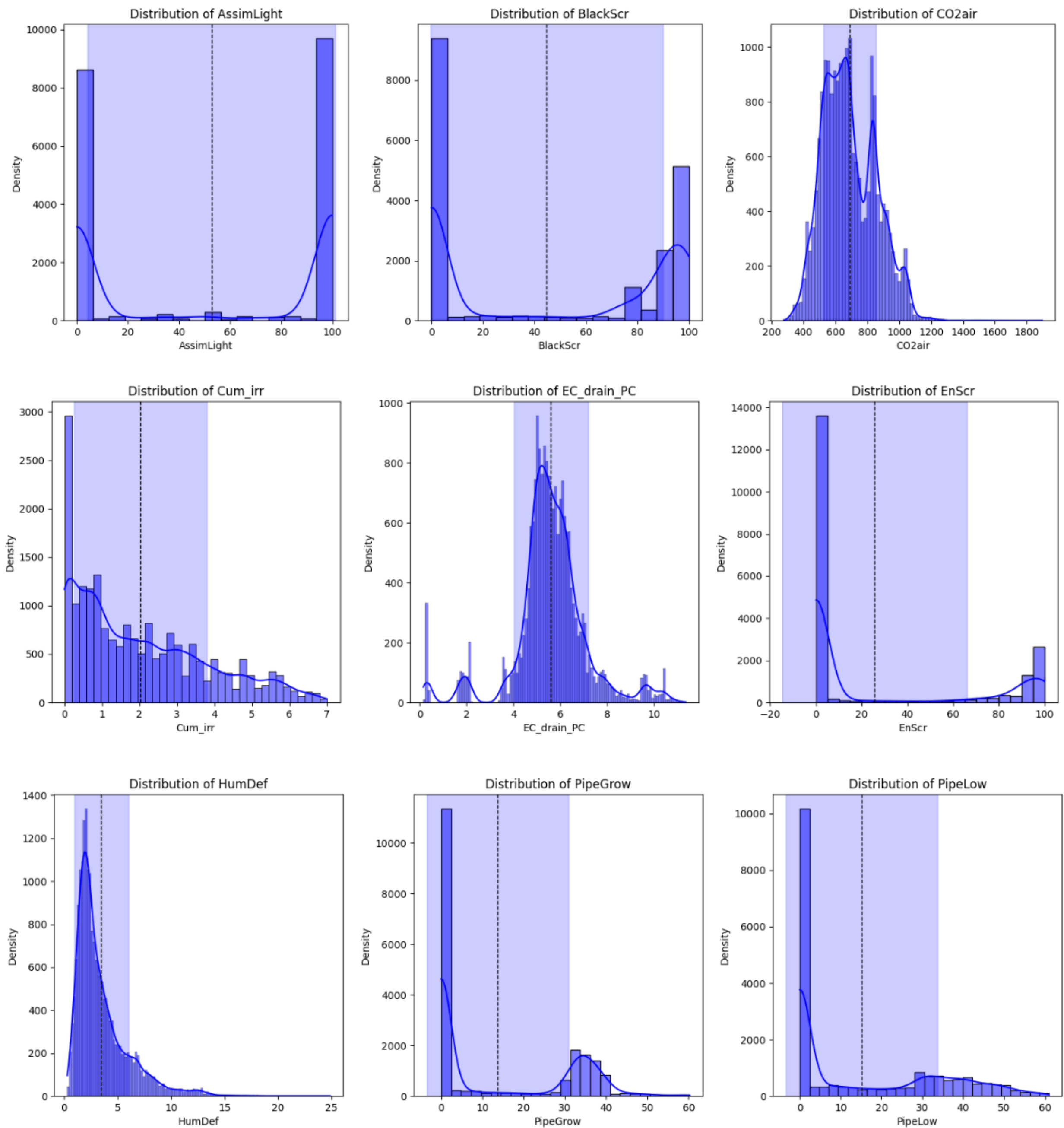
The next step of data preprocessing is to transform the data by normalizing it. Normalization is important for regression models as many of them assume that all features are numerically equivalent in scale and if they are not the model might not treat all features fairly. Some models even require normalization as a prerequisite for them to perform properly like for example Support Vector Machines. Moreover, many regression model algorithms converge faster on normalized datasets for the reason that the weight updates tend to be smaller and the oscillations narrower.

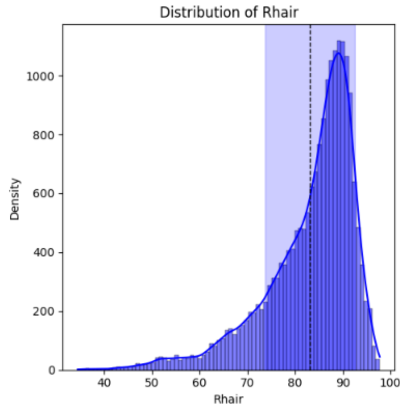
Before proceeding with the normalization, it was decided that the 5 separate datasets, one for each team, are merged. By doing that, the normalization would be the same for all teams and specifically in the Min Max scaling, the maximum normalized value would be assigned to the maximum value of all datasets as well as for the minimum too. So the 5 datasets were merged vertically and a new csv file was created called lens.csv that contains the row index of the end of each team's dataset in the new merged dataset.

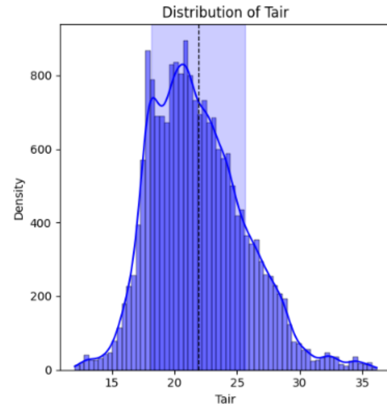
The normalization technique chosen was Min-Max scaling (**2.1.1**) and the new fixed range of the normalized dataset is [0, 1]. One of the pros of using Min-Max scaling is that the range is fixed within boundaries that we set, providing a better understanding of the scale between different variables. This allows us to recognize when a value is at its maximum or minimum, but also in the case of crop parameters if it has even surpassed the dataset's maximum. Another pro is that Min Max scaling is independent of the data's distribution which was an important since the features had different distributions.

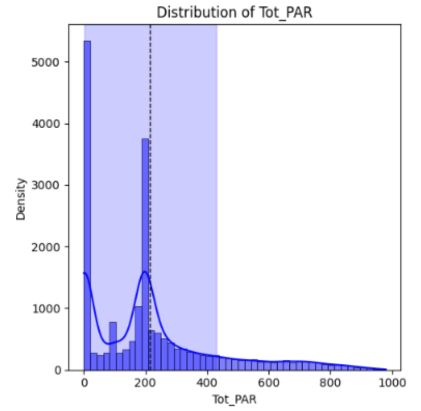
Another normalization technique that had been experimented with is Zero Mean scaling (standardization), which did not yield as effective results. One of the reasons the Zero Mean scaling was not as efficient could be because it works better with data that approximate a normal distribution. Additionally, the other method was preferred for the fact that Zero Mean scaling did not have set the normalized data to a fixed range and some values could even become negative and that would make the problem more complicated.

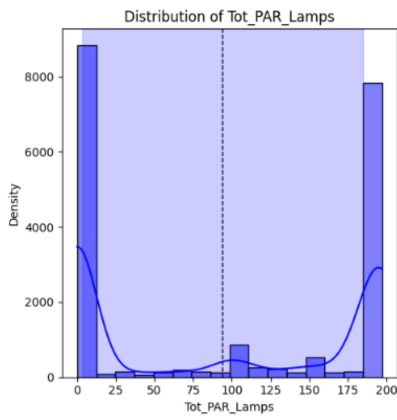
The distribution of the data can be shown in the following plots:

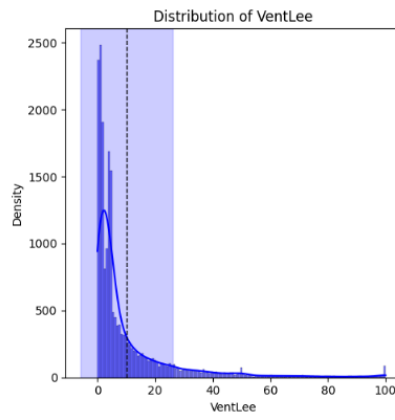


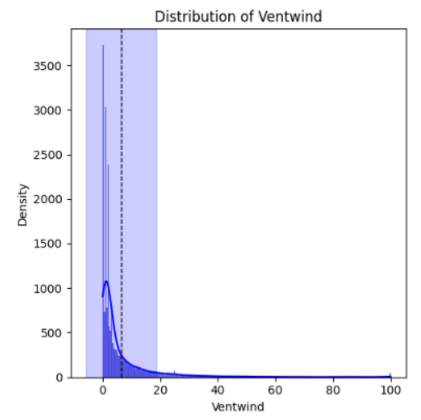

Distribution of Rhair
Density
Rhair

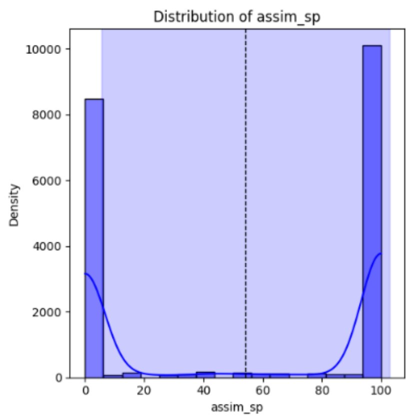

Distribution of Tair
Density
Tair

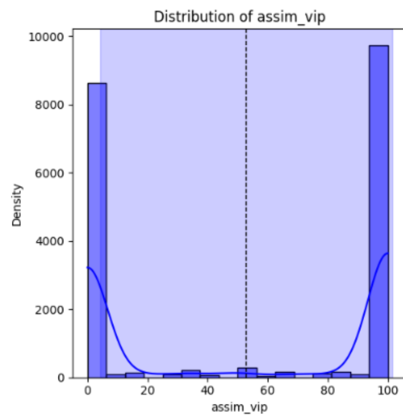

Distribution of Tot_PAR
Density
Tot_PAR

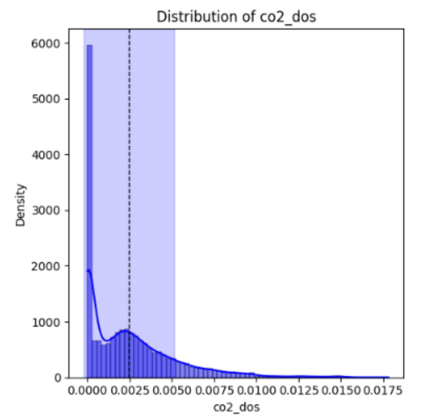

Distribution of Tot_PAR_Lamps
Density
Tot_PAR_Lamps

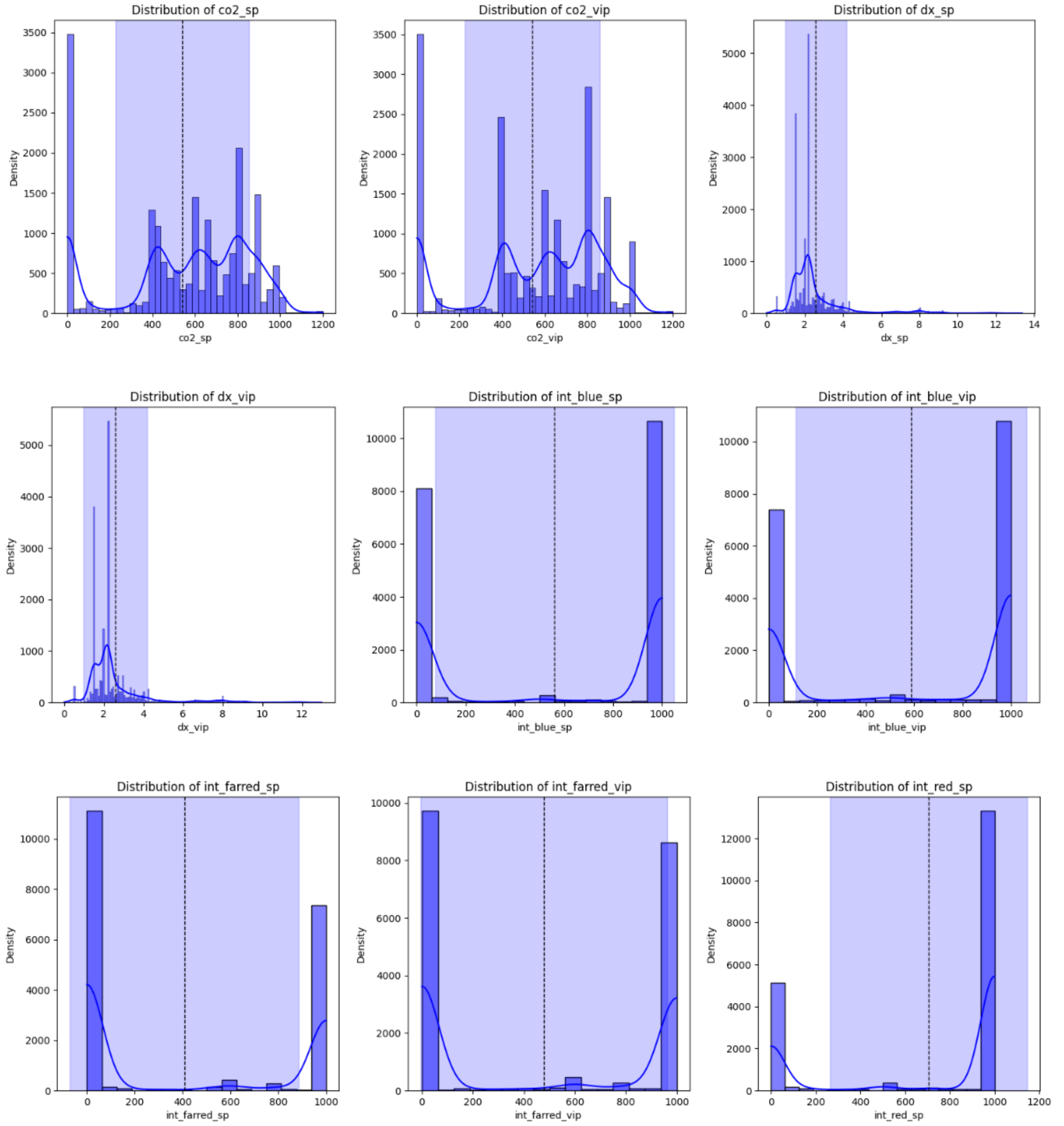

Distribution of VentLee
Density
VentLee

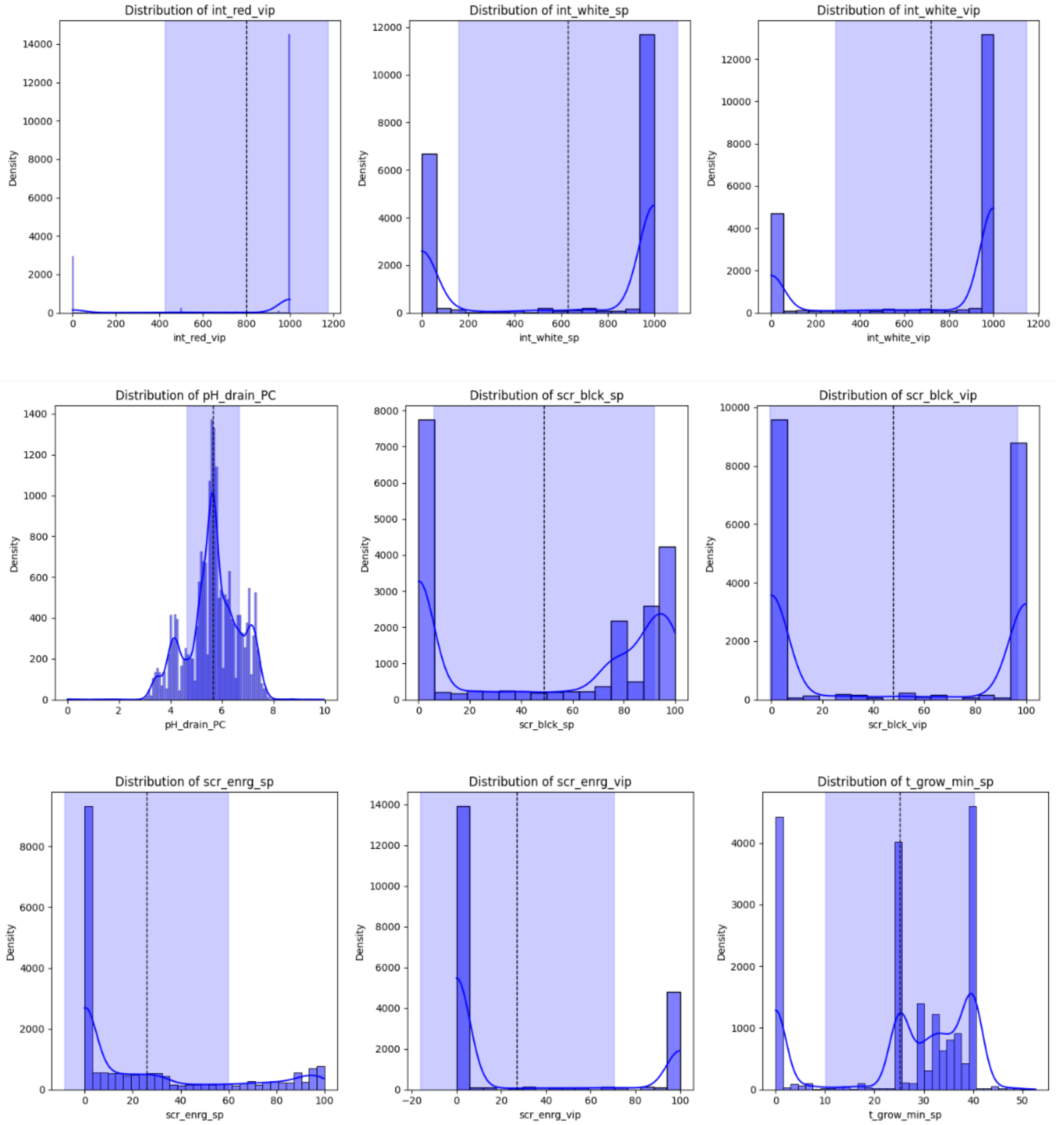

Distribution of Ventwind
Density
Ventwind

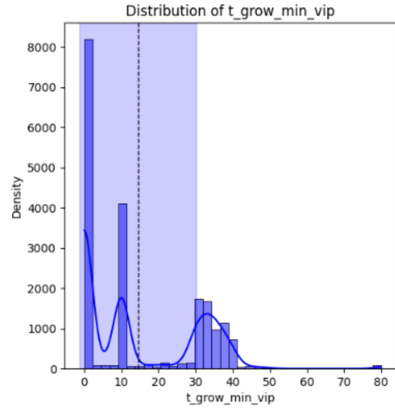

Distribution of assim_sp
Density
assim_sp

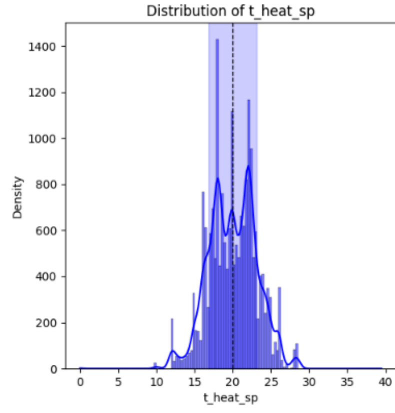

Distribution of assim_vip
Density
assim_vip

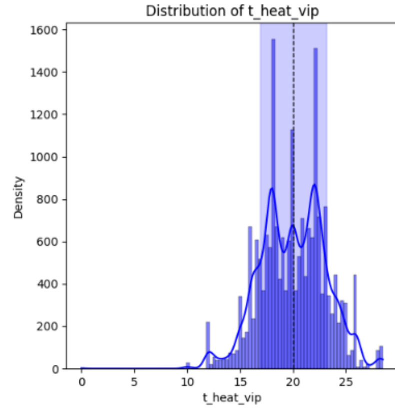

Distribution of co2_dos
Density
co2_dos

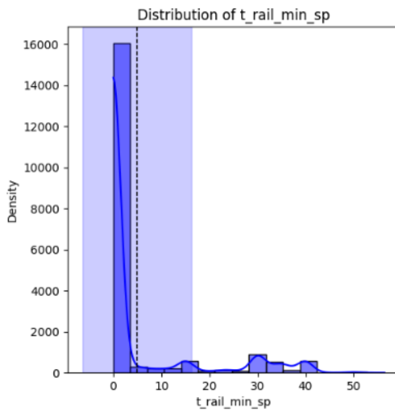

Distribution of co2_sp
Distribution of co2_vip
Distribution of dx_sp
Distribution of dx_vip
Distribution of int_blue_sp
Distribution of int_blue_vip
Distribution of int_farred_sp
Distribution of int_farred_vip
Distribution of int_red_sp
Density
co2_sp
co2_vip
dx_sp
dx_vip
int_blue_sp
int_blue_vip
int_farred_sp
int_farred_vip
int_red_sp

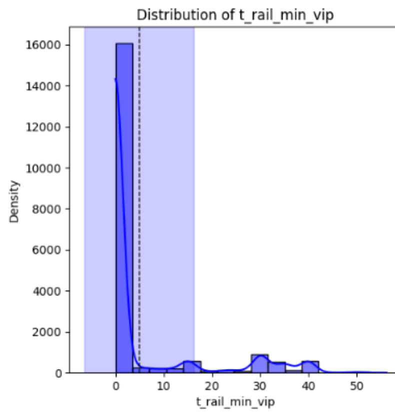

Distribution of int_red_vip
Distribution of int_white_sp
Distribution of int_white_vip
Distribution of pH_drain_PC
Distribution of scr_blck_sp
Distribution of scr_blck_vip
Distribution of scr_enrg_sp
Distribution of scr_enrg_vip
Distribution of t_grow_min_sp
Density
int_red_vip
int_white_sp
int_white_vip
pH_drain_PC
scr_blck_sp
scr_blck_vip
scr_enrg_sp
scr_enrg_vip
t_grow_min_sp

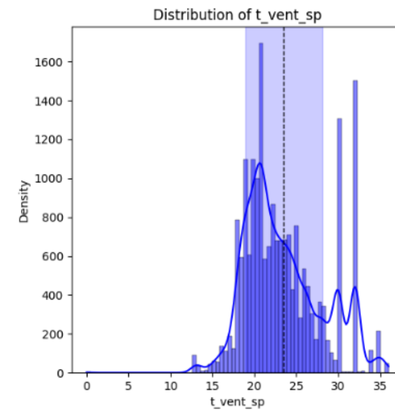

Distribution of t_grow_min_vip
Density
8000
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60
70
80
t_grow_min_vip

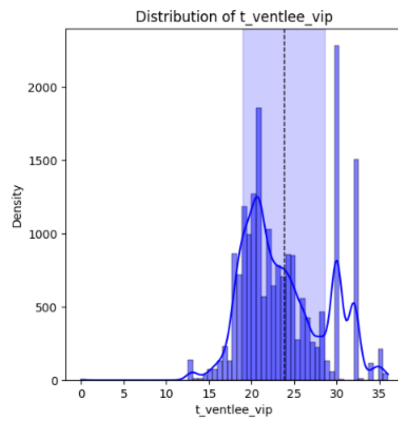

Distribution of t_heat_sp
Density
1400
1200
1000
800
600
400
200
0
0
5
10
15
20
25
30
35
40
t_heat_sp

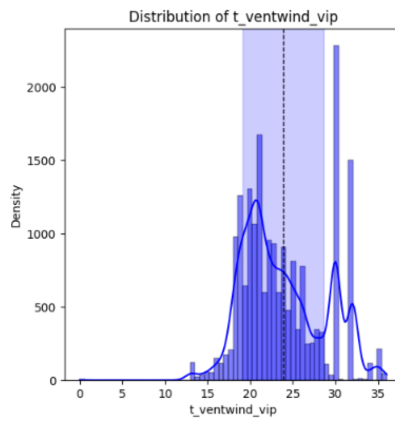

Distribution of t_heat_vip
Density
1600
1400
1200
1000
800
600
400
200
0
0
5
10
15
20
25
t_heat_vip

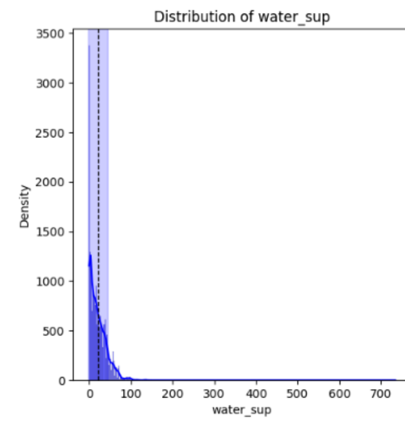

Distribution of t_rail_min_sp
Density
16000
14000
12000
10000
8000
6000
4000
2000
0
0
10
20
30
40
50
t_rail_min_sp


Distribution of t_rail_min_vip
Density
16000
14000
12000
10000
8000
6000
4000
2000
0
0
10
20
30
40
50
t_rail_min_vip


Distribution of t_vent_sp
Density
1600
1400
1200
1000
800
600
400
200
0
0
5
10
15
20
25
30
35
t_vent_sp


Distribution of t_ventlee_vip
Density
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
t_ventlee_vip


Distribution of t_ventwind_vip
Density
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
t_ventwind_vip


Distribution of water_sup
Density
3500
3000
2500
2000
1500
1000
500
0
0
100
200
300
400
500
600
700
water_sup

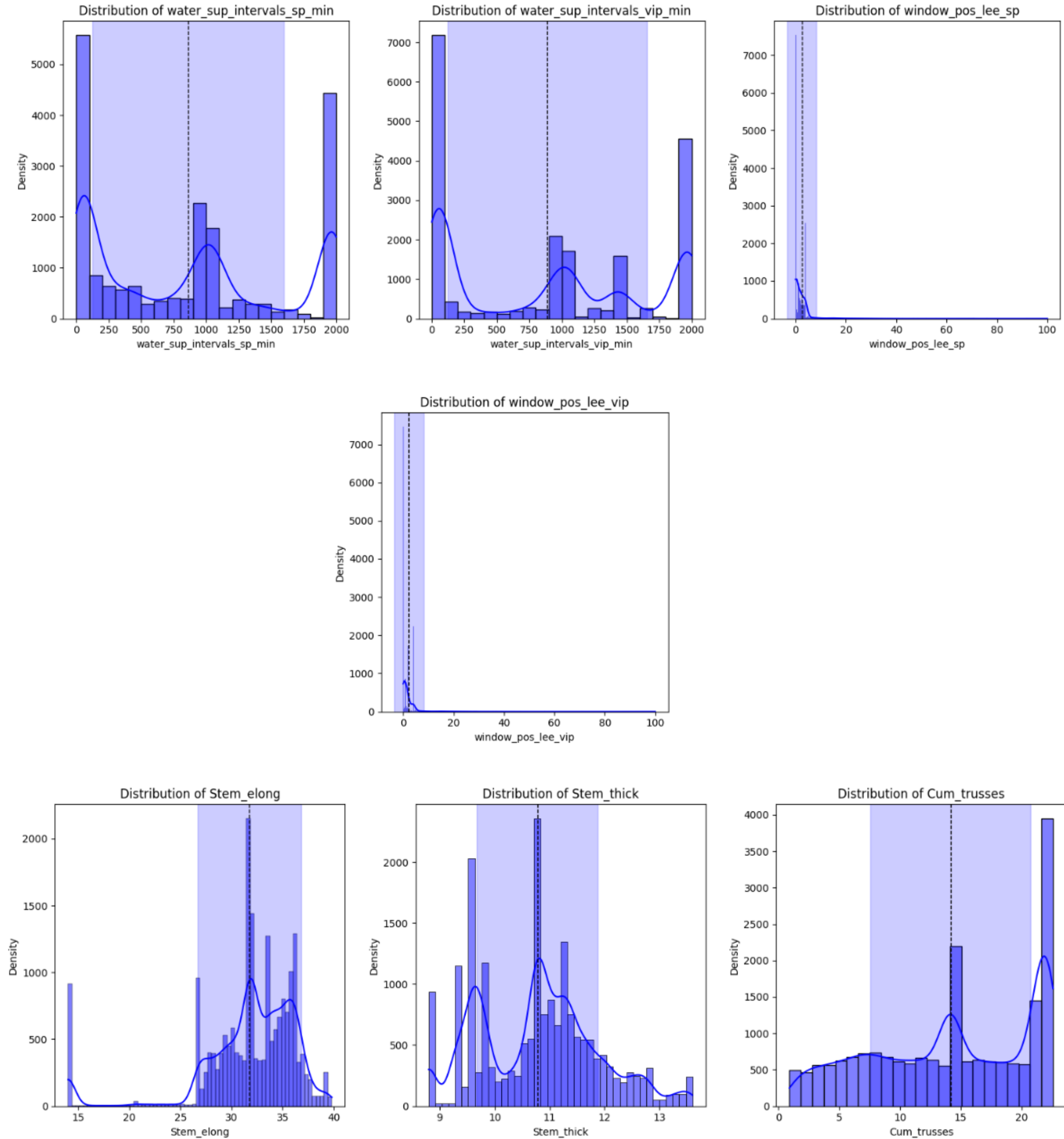


Figure 3.1: Data Distributions for every feature

From the plots it is observed that some features follow gaussian distribution such as Tair and EC_drain_PC but many other features are skewed, others are binary, others are flat and etc.

Therefore, each feature has its own distribution, and many do not follow normal distribution, thus Zero Mean scaling was not effective for this problem.

3.3.5 Feature Engineering

This process involves creating new features or transforming existing ones to uncover patterns in the data that may not be immediately apparent. It has the power to significantly influence the accuracy of regression models by providing additional context to the model or by making the relationships between input and output more explicit.

Looking at the GreenhouseClimate.csv, there was no data for the time of the day the records were made. Even though there is a 'Time' column, it is just an indicator for the timestep each record was made, and it was only useful for merging the CropParameters.csv with the GreenhousClimate.csv. Hence, a feature associating the hour of the day that the raw data was taken would be really beneficial for the model because performing some actions at a specific time of the day could have a better impact rather than another time.

To create this feature, a new column 'hour' was added to the existing dataset of each team and it counted from 0 to 23 to reflect on the 24 hours that a day has. This way the model could associate a specific hour value to certain actions and find a relationship between the hour of the day and the effectiveness of these actions.

3.4 Definition of actions, state and reward

To implement any RL model, actions, state and reward need to be defined first since the RL agent chooses an action each timestep on a current state and then it receives the new state and a feedback from the environment which can be either positive or negative. In the GreenhouseClimate.csv, there are multiple greenhouse parameters from which some of them are completely under human control and some are influenced by other parameters and environmental factors. That being said, it was decided that parameters under the teams' control will be declared as actions and the rest as state.

Here is the list of state parameters:

- **Tair**: No control of greenhouse temperature
- **Rhair**: No control of greenhouse humidity

- **CO2air**: Depends from co2_dos
- **HumDef**: similar to Rhair
- **PipeLow**: No control of the temperature in the rail pipes
- **PipeGrow**: No control of the temperature in the crop pipes
- **Tot_PAR**: Total inside PAR depends on various factors such as outdoor PAR, cover transmissivity, the operation and transmissivity of energy and blackout screens, and the PAR output from artificial lighting sources like LEDs and HPS lamps that are not all under the teams' control.
- **Tot_PAR_Lamps**: Par sum from HPS and LED lamps depends on the lamps
- **EC_drain_PC**: Result of several actions taken previously, Electrical Conductivity of the drainage water suggests either the efficiency of nutrient use or potential issues like over-fertilization.
- **pH_drain_PC**: Represents the pH level of the drainage water in a greenhouse indicator of the current condition of the water's acidity.

Also, all realized points are considered part of the state because these values represent the actual measurements or outcomes resulting from the actions (setpoints) implemented in the greenhouse environment.

List of actions:

- **VentLee**: Open and close Leeward vents
- **Ventwind**: Open and close Windward vents
- **AssimLight**: Switch on and off HPS lamps
- **EnScr**: Open and close energy curtain
- **BlackScr**: Open and close black curtain
- **co2_dos**: Adjust the CO2 concentration in the greenhouse with dosages
- **Water_sup**: Reflects the total irrigation time
- **Cum_irr**: Cumulative number of litres of irrigation

Additionally, all setpoints are considered actions because setpoints are predetermined values to which a system is intended to control.

Therefore, one action is considered a set of values for each of these parameters chosen by the agent for the environment simulator.

As for the reward, since the Stem_elong, Stem_thick and Cum_trusses align with the crop's wellness and tomato quality and quantity, a reward function is needed that involves the three parameters. Since high values for these parameters indicate that the crop is growing, the goal is to maximize these values. Consequently, the reward function can simply be the weighted sum of the three.

The following table represents the final values (unnormalized) of the crop parameters for each team:

Teams:	**Automatoes**	**AICU**	**Digilog**	**IUACAAS**	**TheAutomators**
Stem_elong	26.8	14	33.7	36.3	32.2
Stem_thick	8.8	9.8	9.6	9.3	9.6
Cum_trusses	22.6	22	21.5	21.7	22.2

Figure 3.2: Crop parameter values of the last record for each team

Knowing that the winner of the competition was the team 'Automatoes', it is evident that Cum_trusses had a significant impact for the team's success since it is the only parameter that the 'Automatoes' recorded the highest value compared to all other teams. Ergo, the weight of this parameter should be higher than the other two.

Recall that the features are normalized with Min Max scaling to the fixed range of [0, 1]. Inspired by the reward function of the paper (**Garazhian, 2023**)**[9]**, the reward function is a weighted sum but not of the three parameters but of the difference between the new value and the value before the action was applied. The reason is that this gives clearer feedback for the model's action but also because the reward function can take negative values too since the crop could shrink and that can be taken as a punishment.

The weights for the differences of Stem_elong, Stem_thick and Cum_trusses are 0.2, 0.2 and 0.6 respectively. The purpose of assigning these specific values to the weights is that the

weights sum up to 1 and the difference of Cum_trusses had the biggest weight. In the environment simulator, the crop parameters are clipped to match the range of [0, 2], where values over 1 mean that they are over the highest value in the teams' dataset and 2 is the double of the maximum value. Based on this fact, by summing up the weights up to 1 means that the output of the reward function is within the range of [-2, 2] and the cumulative reward is also within these bounds.

In conclusion the reward function is described by the formula:

$$reward = 0.2(Stem_elong' - Stem_elong) + 0.2(Stem_thick' - Stem_thick) + 0.6(Cum_trusses' - Cum_trusses)$$

3.5 Environment Simulator

The environment simulator is a crucial component for the model-based RL system, particularly in this project where only an offline dataset was provided and no direct access to the greenhouses was possible, which can be very constraining. An accurate simulator ensures that the policies being learned with it are viable and effective in real greenhouses conditions, thus it is important that it reflects the dynamics of a real greenhouse.

3.5.1 Regression-based Predictive Models

Regression-based predictive models are particularly suited for implementing the environment simulator in the context of greenhouse management. This choice is driven by the nature of the data involved which consists of continuous numerical values that describe various environmental conditions within the greenhouse.

In the model-based RL system, regression models play a pivotal role by predicting the next state of the greenhouse environment and key crop parameters. The accuracy of these predictions directly impacts the effectiveness of the environment simulator to mirror real greenhouse conditions.

The first step before building the model is to do feature selection. The model is expected to predict many parameters, the removal of features must be done carefully since a feature can be unrelatable for most parameters, but it could possess a strong relationship with a select few. It was observed that many set points had nearly identical values with the realized points

Besides having both set points and realized points could potentially be redundant, it could also cause confusion to the model and allow it to focus on the other parameters. This could also lead to overfitting as it would not generalize since the actions would be closely related to state parameters that the model was trying to predict. To examine the relationship between set points and realized points, a correlation map was plotted:

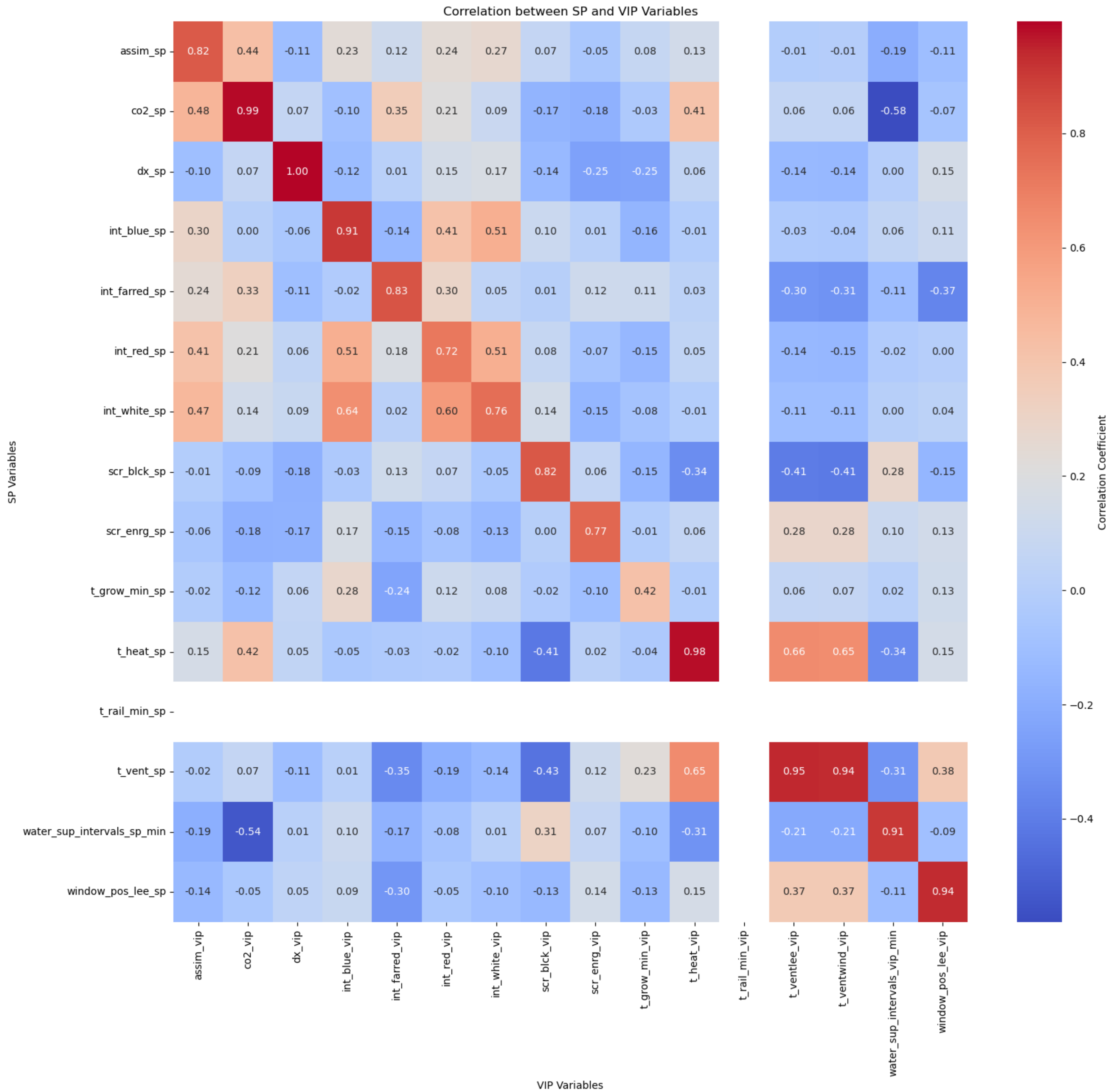


Figure 3.3: Correlation map for setpoints and realized points

Set points are found on the y axis and realized points on the x axis. It is evident that indeed the set point parameters are highly correlated to the corresponding realized point parameters shown as red in the correlation map. However, there is one exception which is

t_grow_min_sp that has only 0.42 correlation with t_grow_min_vip. Thus, it was decided to remove realized points from the state except t_grow_min_vip and keep all the set points as actions. The reason why set points were chosen rather than realized points is for the fact that by having more actions gives more flexibility to the agent to learn and experiment. In addition, the realized points are only dependent from the set points so keeping them as part of the state could potentially create false relationships between other parameters.

After testing the models without the realized points, the performance improved dramatically and the model was simplified without losing any valuable information.

3.5.1.1 State Prediction Regression Models

The objective of this model (Appendix B3) is to predict the next state parameters based on the action parameters and the current state parameters, i.e. the input of the model is the action and the current state, and the output is the next state.

$$\boldsymbol{Action}[\boldsymbol{t}] + \boldsymbol{State}[\boldsymbol{t}] => \boldsymbol{State}[\boldsymbol{t}+\mathbf{1}]\text{, where } \boldsymbol{t} \text{ is the timestep.}$$

The input dataset X is ready from the data processing step since a record contains both action and state, but we need to create the expected output dataset Y that will contain the corresponding expected next state for each record in X which is basically next record's values that are part of the state and not the action. As already mentioned, the dataset is a merge of all 5 teams' datasets with lens.csv containing the indexes of which row each team sequence ends.

Iterating through the **lens.csv** indexes in the X dataset, for each team's sequence called current_sequence (a subset of X), a copy of the sequence containing only the columns that are involved in the state set is made called current_targets and then it is shifted by one timestep, thus creating the expected next state for each record of the current_sequence.

After shifting the current_targets, the last row becomes Nan, thereby it is dropped. As for the current_sequence, the last row does not have an expected output since it is the end of the sequence, so the last row is dropped here too.

At the end of this process, we have 5 current_targets for each team that are merged to a new dataset Y to match the input dataset X and now that both X and Y are created, the model training can begin.

Since the dataset is small and having a separate validation set would mean less data for training, the models were trained and evaluated with the 10-fold CV method (**2.2.1.2**). With the CV method, a model is trained from scratch with each fold and then the average of each metric is provided for evaluation.

The type of models that CV was applied to were:

- Random Forest Regressor
- Multi-Output Gradient Boosting Regressor
- Multi-Output Support Vector Regressor
- K-Nearest Neighbors Regressor
- Linear Regressor
- Elastic Net

And the metrics used to compare these types of models are:

- MAE
- MSE
- RMSE
- R2

After evaluating the types of models, the 10 models, that were trained with CV, of the best performing type of model were saved and used in the environment simulator for predicting the next state.

3.5.1.2 Crop Parameters Prediction Regression Model

The objective of this model (**Appendix B4**) is to predict the next values of the crop parameters based on the action parameters the current state parameters and the current crop parameter values. So the input is the action, current state and the current crop parameters and the output is the next crop parameter values.

$$\boldsymbol{Action}[\boldsymbol{t}] + \boldsymbol{State}[\boldsymbol{t}] + \boldsymbol{Crop\ Parameters}[\boldsymbol{t}] => \boldsymbol{Crop\ Parameters}[\boldsymbol{t} + \mathbf{1}],$$

where $\boldsymbol{t}$ is the timestep.

The reason why the State and Crop Parameters prediction models were not combined to one as follows:

$$\boldsymbol{Action}[\boldsymbol{t}] + \boldsymbol{State}[\boldsymbol{t}] + \boldsymbol{Crop\ Parameters}[\boldsymbol{t}] => \boldsymbol{Crop\ Parameters}[\boldsymbol{t} + \mathbf{1}] + \boldsymbol{State}[\boldsymbol{t} + \mathbf{1}]$$

is because the next state (State[t+1]) has no relation to the current values of the crop parameters (Crop Parameters[t]). By combining these models, it would potentially create false relationships between the current crop parameters and the next state which would be wrong and not realistic. Creating false dynamics between parameters would lead to the failure of reflecting real greenhouse conditions which is the main goal of an environment simulator.

The input dataset X is the same with the dataset of the other model's X dataset but it also includes the interpolated crop parameters as mentioned in the Data Expansion section (**3.3.3**). As for the Y dataset, it now includes the expected next crop parameters, not the expected next state of the next record. In this case, the methodology for creating the Y dataset is the same as the State Prediction Regression Model but instead of current_targets being the state parameters shifted by one timestep, it is now the crop parameters shifted by one timestep. The last row is also dropped on both current_targets and current_sequence (subset of X) and by merging the current_targets the Y dataset is created.

Same as the State Prediction Regression Model, CV was used for model training and the model types trained were:

For the model training, CV was used for the same model types as for the State Prediction Regression Model and evaluated with the same metrics.

After evaluating the types of models, the 10 models, that were trained with CV, of the best performing type of model were saved and used in the environment simulator for predicting the next crop parameters.

3.5.2 Creation of tf_agents py_environment

The creation of the environment simulator, **tf_agents py_environment** is explored serving as the foundation for implementing the environment simulator (**Appendix B5**). The library is specifically designed to support the dynamic and efficient execution of reinforcement learning methods. Additionally, it is modular and friendly to TensorFlow, making it possible to integrate the simulator model with the framework and leverage the TensorFlow environment for better data transmission and processing methods. The library is ideal for developing complex reinforcement learning models and methods because such frameworks require strong data flow and computation efficiencies. Furthermore, the library comes with various available algorithms and environments making it possible to use preexisting components in the development process. Subsequently, the tf_agents library is designed with a compilation of custom environments that ensure the environment requirements can be met with little effort in the customization. Hence, the tf_agents library is robust and flexible for creating the simulator.

3.5.2.1 Initialization

The creation of a TensorFlow Agents (tf_agents) PyEnvironment**[12]** involves defining the environment's action and observation specifications, setting boundaries for these specifications, and initializing the environment's state along with the crop parameters which are expected to increase during training. In our case, the environment is designed to simulate a greenhouse system, where various parameters are controlled, some directly by the agent as they are part of the action and some indirectly as part of the state.

The **action specification** defines what actions the agent can take within the environment. The specifications involve the greenhouse parameters that are considered as part of the action as discussed in the **3.4** section. So, every action is considered a selection of values by the agent for each action parameter within the set boundaries in the specifications. The boundaries for each parameter are set to [0, 1] and this is because knowing that the data was normalized with Min Max scaling (**2.1.1**), the values in the normalized dataset are in the range of [0, 1] where 0 is the minimum and 1 the maximum. Since the goal for the environment simulator is to mimic the behavior of the real greenhouse environment, these boundaries were chosen to ensure that the simulated greenhouse environment remains within

realistic and manageable ranges. This also prevents the generation of unrealistically large or small values during simulation while maintaining consistency with the dataset.

As for the **observation specification**, it involves the parameters that are part of the state as discussed in the **3.4** section and just like the action specifications, the set of parameters is bounded in the range of [0, 1] for the same exact reason.

After defining the specifications, the environment's state and crop parameters are initialized using the **reset()** function which is also used at the start of each new episode. Specifically, the function initializes some environment variables and sets the greenhouse environment's initial state and initial crop parameter values, i.e. stem elongation, stem thickness, and cumulative trusses, to values near the minimum observed in the dataset. By setting the crop parameters near their lower bounds, the increase in cumulative reward in the subsequent iterations becomes more evident and easier to discriminate. In terms of learner's experience, the selected initial configuration makes the agent more attuned to its perceptions concerning the ramifications its actions have on the environment, which considerably speeds up knowledge acquisition. Moreover, beginning from the same source state and crop parameter values as the teams did, it promotes comparison and carries over the continuity of past experiments to the present study. Having this base configuration, we can evaluate the learner's performance against benchmarks from previous studies. In addition, initializing the crop parameters from their minimum or near-minimum limits presumes a vast potential for growth from the very beginning, as the system boundary is uniform.

Additionally, other variables in the class are initialized to enable managing the overall operation of the environment. Some of the most notable variables are the weights of the reward function, that determine which crop parameter is more important in the reward calculation, the timestep counter, which is initialized to 0, the list of cumulative reward of each episode, which will be plotted after the end of the training and the file path for the models. Once the model path is initialized, the predictive models developed which are referenced in section **3.5.1** are loaded to memory.

3.5.2.2 Updating State

In this section, we discuss the mechanism through which the environment updates its state in

response to an action taken by the agent, a crucial aspect of the simulation's dynamic nature. As detailed in section **3.5.1.1**, the system employs an ensemble of 10 models, that were trained with CV, to predict the subsequent state of the environment. Each model in this ensemble is designed to take the current state and the agent's recent action as inputs and output a predicted new state.

Given the inherent uncertainties and complexities of predicting environmental responses, the ensemble method enhances the robustness and accuracy of the state prediction. Instead of relying on a single model's output, which may be prone to specific biases or errors, the ensemble approach aggregates the predictions from all 10 models. This is achieved by averaging the outputs for each state parameter across the models. For example, if the state parameters include factors such as temperature, velocity, and position, each model will provide a prediction for these variables, and the average of these predictions will form the composite new state.

The averaging process helps in smoothing out anomalies and reducing the influence of any single model's potential predictive inaccuracies. The resulting averaged parameters are then compiled into a unified state vector, which is fed back into the environment simulator. This updated state vector provides a more reliable and stable basis for the environment, reflecting the collective inference of multiple predictive insights, and thus supports the agent in making more informed decisions in the subsequent steps.

This method not only leverages the strengths of multiple predictive models but also mitigates their individual weaknesses, ensuring that the environment's evolution is both realistic and grounded in a comprehensive analytical approach. The use of an ensemble also aligns with best practices in machine learning, where model averaging is known to improve prediction performance over using single models.

Once the new state is predicted, the state parameters are clipped to [0, 1] as the observation specifications require and the hour feature is increased modularly by 1/24 so it is also within the bounds of [0, 1] and a dictionary is created that includes the predicted state and the new hour. This dictionary replaces the old state dictionary with the update method, thus updating the state successfully for the next timestep.

3.5.2.3 Updating Crop Parameters

Besides updating the state of the environment, the simulator needs to update the crop parameters too, which are key targets that the agent aims to optimize through its actions. which the agent aims to maximize with its actions. Similarly with the state updating, the environment needs to predict the new crop parameters first with the ensemble of 10 models as described in the section **3.5.1.2**.

After obtaining the averaged predicted crop parameters, the values are clipped to [0, 2]. The reason why the crop parameters are clipped to fit in this specific range is because it doesn't make sense for the stem to shrink more than the minimum values of the whole dataset which is 0 since the dataset was normalized with Min Max scaling. As for the upper bound, it is set to 2 because by setting the upper limit to 2, the model allows for the possibility of the crops achieving up to double the maximum value observed in the dataset. This setting not only enables the agent to potentially outperform the existing benchmarks but also remains within realistic growth expectations under optimal but achievable greenhouse conditions. This strategic clipping ensures that the crop parameters remain within a feasible range, fostering both realism in simulation behavior and ambition in the agent's performance objectives.

The predicted crop parameters are then used to calculate the differences between the predicted and the old crop parameters for the reward function and then a dictionary is created with the predicted crop parameters and the differences. This new dictionary of crop parameters will replace the old, thus updating them so the model can move on to calculate the reward for the current timestep.

3.5.2.4 Calculate Reward

The environment besides updating its state, it is also responsible to provide immediate feedback for the agent's action. By updating the crop parameters and taking the differences from their old values, the environment is able to calculate the immediate reward for each timestep and the cumulative reward of the episode.

The computation of the immediate reward is fundamental because it guides the agent's learning process, influencing how it adjusts its actions to maximize future rewards. By

assigning a numerical value to each action's outcome, the reward helps the agent discern beneficial actions from detrimental ones within the context of the given environment.

Furthermore, the environment also computes the cumulative reward, which is the total reward accumulated over an episode. Plotting this cumulative reward is extremely useful for several reasons. One good reason is that the cumulative reward provides a straightforward metric to evaluate the agent's performance over time. By observing the total rewards per episode, it can be determined if the agent’s attempts to maximize rewards are getting better as the episodes go on, worse or neither. Moreover, plotting the cumulative reward can help detect issues in the learning algorithm, too high or too low learning rates and inadequate exploration. Finally, this method helps identify when the learning process is converging to optimal policy and determine if the number of episodes was sufficient or not.

The reward function is viewed in the appendix where each crop parameter difference is first multiplied with its corresponding weight and then summed giving a scalar value. This value is then added to the cumulative reward variable and then returned to the agent as a reward. Both the immediate and cumulative rewards are within the range of [-2, 2] for the reason discussed in the section **3.4**.

3.5.2.5 Step function

In the implementation of a reinforcement learning environment using tf_agents, the step function is a core component that dictates the progression of each timestep and ultimately, each episode within the environment. This function is invoked whenever the agent takes an action, and its execution represents a single iteration of the agent-environment interaction loop.

The function begins by receiving an action, which the agent decides based on its current policy. The action is assumed to be a dictionary where keys correspond to the defined action specifications.

The environment simulator in the step function first updates the crop parameters based on the agent’s action and the current state. Next, the simulator updates its state after considering the action's impact which also is assumed to be a dictionary where keys correspond to the defined observation specifications.

Once the timestep is finished, the timestep class variable is incremented and if this variable does not exceed 4000 timesteps, then the environment simulator returns the state and the reward calculated. Otherwise, if the timestep is over 4000, then the cumulative reward of the episode is appended to the list of cumulative rewards and the episode ends returning the last state and reward to the agent. The reason why 4000 is the timestep limit is because the dataset of teams, after it was aggregated to hourly recordings, had 4000 records each (24 weeks multiplied by 168 hours weekly). Hence, one episode is set to 4000 timesteps to match the duration of the teams' experiment for future comparison.

3.6 Agent

The agent operates within a TensorFlow Agents (tf_agents) framework designed to mimic a complex greenhouse. The primary role of the agent is to optimize the crop growth parameters, by taking appropriate actions based on the state of the environment. Through continuous interaction with the environment simulator described in section **3.5**, the agent learns by receiving immediate feedback in the form of rewards that evaluate the impact of its actions on crop health and productivity. This iterative training process involves the agent executing actions, the environment responding with new state updates and rewards, and the agent updating its policy based on this feedback. Over time, the agent refines its strategy to maximize cumulative rewards, thereby learning to make decisions that yield the most beneficial outcomes.

The agent is trained with two algorithms in this thesis, PPO and REINFORCE.

3.6.1 PPO

Proximal Policy Optimization (PPO)**[13]** is a cutting-edge algorithm in the realm of Deep Reinforcement Learning (DRL). Developed by OpenAI, PPO has gained significant traction due to its simplicity, stability, and remarkable performance across various tasks.

PPO falls under the category of Policy-Based methods in DRL (**2.2.2.2**). Unlike value-based methods that focus on optimizing the value function or actor-critic methods that combine features of both value-based and policy-based methods, PPO directly optimizes the policy function. This characteristic makes it particularly suitable for environments with continuous action spaces.

This algorithm is known for its stability during training meaning that the policy updates are controlled, preventing the network from diverging or becoming unstable. Also, PPO utilizes training samples which lead to faster convergence and requires fewer samples compared to some other deep reinforcement learning algorithms. Furthermore, it is well-suited for environments with continuous action spaces, like the one for this project, as it directly optimizes the policy function. Finally, maintains a balance between exploration and exploitation, crucial for learning in dynamic environments. By controlling the magnitude of policy updates, it ensures that the agent explores new strategies while not deviating too far from previously learned policies.

The policy network, which is commonly implemented as a neural network, receives the current state of the environment and generates a probability distribution for the available actions. PPO then interacts with the environment to collect experiences, which contain state-action pairs, and the corresponding rewards. The experiences are essential in assessing how sophisticated the policy is in maximizing the expected rewards. PPO then proceeds to improve the policy by updating its parameters. This is dissimilar to value-based approaches that consist of optimizing the value functions because PPO directly improves the policy function. This is done through policy-based optimization where the innovation of PPO is its constrained-based form of optimization that ensures stability. Rather than allowing for an unconstrained policy update, PPO clips the changes to the policy to ensure that they do not change significantly. The policy network training uses a gradient-based optimization technique such as stochastic gradient descent or the Adam optimizer. These two techniques minimize the objective, thus updating the policy in each iteration. PPO ensures that learning is stable and efficient in an environment with a continuous action space.

Implementation:

The training setup for the PPO agent (**Appendix B6**) comprises two central components, the actor and the value networks. These networks are designed with the **actor_distribution_network** and **value_network** classes made available by TensorFlow Agents. Both the actor and the value networks incorporate a custom combiner, which processes the inputs by concatenating all relevant tensors to ensure that the features from

different sources can be effectively merged before they are fed into the fully connected layers with 100 units each. This setup allows the networks to thoroughly process the complex interactions between the states and the actions in the environment.

The optimization of the network is handled using an **Adam optimizer** with a learning rate of 0.01 to ensure effective and efficient gradient descent during the training. The agent itself is configured to use various configurations, including observation and reward normalization and generalized advantage estimation over ten training epochs. These features enhance the agent's performance by ensuring a more stable and converged learning process.

The data necessary to train the agent is recorded and stored in a **TFUniformReplayBuffer** to allow batch learning from the experience. A **DynamicEpisodeDriver** is used to capture data into the buffer as the agent interacts with the environment over multiple episode sequences. This setting not only allows the agent to have enough data for learning but also to refine its policy in a thorough state-action-reward experience review. This setup of the components allows for an effective training of the PPO agent, promoting optimal learning and adaptability in the greenhouse environment simulation.

3.6.2 REINFORCE

REINFORCE**[14]**, referred to as the Monte Carlo Policy Gradient, is one of the core algorithms of this field and has been critical in the evolution and exploration of policy-based methods in DRL (**2.2.2.2**) due to its simplistic implementation and theoretical characteristics.

Unlike value-based methods, which seek to optimize the value function, or the actor-critic family that balances optimization with both the policy and the value function, REINFORCE belongs to purely policy-based methods. This means it aims to optimize the policy function entirely, ignoring the value one. Such characteristics make REINFORCE particularly effective in cases where the sequence of decision-making steps is essential and when the environment is highly stochastic.

REINFORCE is unique in that it operates on complete episodes for learning, making it an “on-policy” method. This implies that it is not dependent on prior or different policy trajectories and always uses the episodes under its current policy to update its parameters. Its

major strengths are that the simple algorithm can precisely optimize policies in an unbiased manner. It does this by focusing on the final output of policy execution, the total of rewards accumulated. This learning signal is consistent and directly correlated to the objective of maximizing returns, hence making it very reliable especially when the reward signal of the environment is clear and the episodes are clearly defined. It secondly accommodates high-dimensional action spaces and policy structures regardless of complexity, which is actually the norm in real-world applications. Its other critical aspect is that it inherently supports learning and exploration since exploration is made on sampled policy improvement, which satisfies the vital need to discover where new policies highly lead to improved performances.

The operational mechanics of REINFORCE are based on the generation and evaluation of episode trajectories. The REINFORCE algorithm has a policy network at its heart, often implemented via a neural network that receives the current environmental state and outputs a distribution of probabilities over possible actions. This distribution then plays a critical role in guiding the agent's actions in the environment. During the episode, the agent interacts with the environment under this policy, recording all state-action pairs and noting the immediate rewards. Once the episode concludes, the total discounted reward for each action is calculated and returned. These returns are then used to calculate gradients, which adjust the policy's parameters to increase the probability of actions that yield higher returns. The key aspect here is that the policy is updated after the episode is completed, ensuring that each update is informed by an entire sequence of actions and outcomes, reinforcing the connection between actions taken and their long-term results.

This episodic update method contrasts sharply with algorithms that update based on single steps or pairs of actions and rewards, allowing REINFORCE to leverage all the information from fully explored sequences. By updating the policy parameters using full returns, REINFORCE aligns directly with the objective of maximizing cumulative rewards, making it effective in episodic environments. The policy parameters are typically computed via gradient ascent, often using simple but powerful optimizers such as stochastic gradient descent. Although this approach isn't always the quickest or least variant, it excels in environments where the quality of an entire sequence of actions determines success.

Implementation:

The REINFORCE setup (**Appendix B7**) includes an actor network, implemented using the **actor_distribution_network**. A key feature of this network is the custom combiner layer, which concatenates all input tensors into a single flattened vector. This is crucial for processing the diverse and high-dimensional state inputs. The actor network consists of two fully connected layers with 100 and 50 neurons, respectively, designed to capture the complex interrelationships within the environmental data and facilitate precise action selection based on the current state.

In terms of agent optimization and initialization, the REINFORCE agent employs an **Adam optimizer** with a learning rate of 0.001 and features return normalization, which adjusts the rewards to have zero mean and unit variance, significantly enhancing the learning stability. The agent's initialization includes setting up a TensorFlow variable train_step_counter, which helps monitor and control the training process.

Training data is collected and stored in a **TFUniformReplayBuffer**, which holds trajectories that include states, actions, and rewards. This data is crucial for updating the agent's policy. Data collection is managed by a **DynamicEpisodeDriver**, which orchestrates the interaction between the agent and the environment, ensuring a consistent and efficient collection of experiences. This driver executes the policy over multiple episodes, enabling the agent to explore and learn from a diverse array of environmental scenarios.

3.6.3 Training and evaluation

Both algorithms during training are set to execute a set number of episodes, specifically 100 episodes in this scenario. Each episode encapsulates a complete interaction cycle between the agent and the environment for 4000 timesteps, reflecting a full set of actions from start to finish. Unfortunately, due to the lack of computational power and time the algorithms could not train for more episodes.

At the start of each episode, the **DynamicEpisodeDriver** executes the agent's current policy to interact with the environment. This driver ensures the capture of complete trajectories, which include sequences of states, actions, and rewards, all stored in the **TFUniformReplayBuffer**. Following the conclusion of an episode, these trajectories are

retrieved from the buffer for training purposes. The agent processes this data to update its policy, aiming to refine its decision-making skills based on the outcomes of the whole episode. The session ends with the calculation and recording of training loss, providing a clear metric of the agent's progression in learning throughout the episode. Then, the replay buffer is cleared to accommodate fresh data from upcoming episodes.

After training, the cumulative rewards collected during each episode are plotted. This visualization serves as a tool to evaluate the effectiveness of the agent’s evolving policy across episodes, showcasing the learning trends and the agent’s increasing capability to maximize rewards.

Once training is complete, the agent's performance is assessed through an evaluation phase to test the effectiveness of the learned policy. This phase uses a similar environmental setup to maintain consistent testing conditions. The evaluation involves running the agent through 10 separate episodes in the test environment. The agent navigates through the environment based on its learned policy, and the actions taken, along with the resultant state transitions and rewards, are recorded. The total reward for each episode, representing the agent’s performance, is calculated by summing up all received rewards.

The cumulative rewards for each evaluation episode are then plotted. This chart is crucial as it reflects the agent’s ability to apply its learned policy to new scenarios, indicating how well the training has generalized. It also highlights the consistency of the agent’s performance and pinpoints areas that may require further refinement.

Chapter 4

Results

4.1 Prediction Models 50
4.1.2 State Prediction Model 52
4.1.2 Crop Parameters Prediction Regression Model 55
4.1.3 Algorithm Justification 57
4.2 DRL algorithms 58
4.2.1 PPO 59
4.2.2 REINFORCE 63
4.2.3 Best algorithm 67

4.1 Prediction Models

To build the State Prediction Regression Model and the Crop Parameters Regression Prediction Model, a lot of regression algorithms were cross validated for comparison. The algorithms were:

- Random Forest Regressor
- Gradient Boosting
- Support Vector Regressor
- K-Nearest Neighbours
- Linear Regression
- Elastic Net

Random Forest Regressor:

Random Forest is an ensemble model that constructs multiple decision trees during training and averages the predictions of each tree. This method excels in regression tasks as it

effectively manages overfitting without significantly increasing bias errors. It is particularly useful in high-dimensional spaces.

Gradient Boosting Regressor:

Gradient Boosting is an ensemble learning technique that builds models sequentially, with each model aiming to correct the errors of its predecessor. This approach optimizes a loss function to generate predictions and is known for its high accuracy, even with complex datasets where relationships between parameters are nonlinear.

Support Vector Regressor (SVR):

SVR utilizes principles similar to Support Vector Machines (SVM), a well-regarded method in classification tasks. SVR focuses on maximizing the number of instances between two boundaries while minimizing margin violations, making it resistant to outliers and effective in high-dimensional spaces.

K-Nearest Neighbors (KNN):

The KNN algorithm is a form of instance-based or lazy learning where computations are deferred until classification. It predicts outcomes by analyzing the behaviors of a case's nearest neighbors, averaging their attributes. This method is straightforward and useful where predictions can be directly derived from nearby data points in the feature space.

Linear Regression:

Linear Regression is a fundamental and traditional method in statistics and machine learning for predicting a response variable from one or more predictors using a linear relationship. It is especially effective when the relationships within the data are linear.

The Gradient Boosting Regressor and the Support Vector Regressor do not support multi-output regression. To solve this problem, they were wrapped with the meta-estimator '**MultiOutputRegressor**' that basically constructs one regressor per target. This assumes that the outputs are not related to each other and each regressor is fitted independently on a single output.

When it comes to the complete assessment of regression models designed to predict state and crop parameters, it is essential to use a varied of performance evaluation during CV, as input

features have different and non-relatable natures. Given that the input data has distributions as distinct as Gaussian, skewed, flat, and random ones as shown in the section 3.3.4, different statistical methods should be applied to ensure a universally applicable assessment of model fitness and effectiveness. For this reason, three metrics were selected for comparing the algorithms: **Mean Absolute Error** (**MAE**), **Mean Squared Error** (**MSE**), and **R-squared** (**R2**). Each of these metrics provides a unique perspective on model performance, as detailed in Section **2.2.1.3**. MAE offers insights into the average magnitude of the errors without considering their direction, making it a straightforward indicator of average error. MSE, which squares the errors before averaging, emphasizes the impact of larger errors and is particularly useful for identifying models that might be prone to significant prediction errors. R-squared measures how much of the variation in the dependent variables (outputs) can be explained by the independent variables (inputs), indicating how well the model fits the data. Together, these metrics provide a well-rounded assessment of model accuracy, fit, and robustness, and an informed comparison across the regression algorithms employed in the project.

4.1.2 State Prediction Regression Model

Before Cross-Validating, a predefined set produced by the **train_test_split** function from Scikit-learn, which allocated 80% of the data to the training set and 20% to the testing set, with a fixed random state to ensure reproducibility of the split, was used to train the models for each algorithm. After training, the models were used to make predictions on the dataset, which were then utilized to calculate the metrics **MAE**, **MSE**, **RMSE** and **R2**.

The results were:

Model	MAE	MSE	RMSE	R2
Random Forest Regressor	0.030566	0.004200	0.064805	0.928661
Gradient Boosting Regressor	0.037728	0.006782	0.082352	0.909448
Support Vector Regressor	0.057036	0.008510	0.092252	0.865166
K-Nearest Neighbors	0.038598	0.007360	0.085789	0.887993
Linear Regression	0.052392	0.011923	0.109194	0.854127
Elastic Net	0.192762	0.059946	0.244839	-0.000242

Figure 4.1: Metric table for the training of State Prediction Regression Model with every model type using the train_test_split

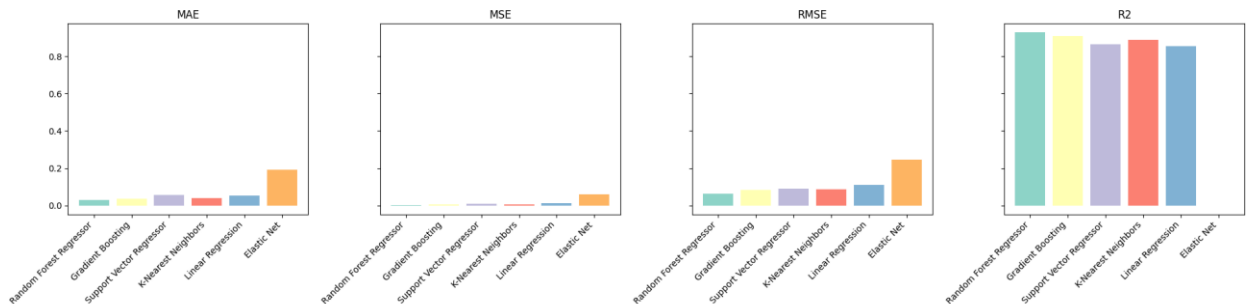


Figure 4.2: Metric plot or the training of State Prediction Regression Model with every model type using the train_test_split

Overall, the results show that most models performed relatively well, with **Random Forest Regressor** and **Gradient boosting Regressor** showing the best results by having relatively high R2 values and low error metrics. Given that both models are based on ensemble methods where multiple decision trees make a decision and thus outperform individual models, it can be assumed that the data contains complex and nonlinear relationships which the model could capture. However, the **Elastic Net** has a very low level of performance with closer to zero

R2 and significantly higher error values. This can be due to the linear nature of the model that cannot capture the complexity of the dataset or due to multicollinearity in which the Elastic Net method was penalized, and therefore difference.

However, a further evaluation with CV was applied for several reasons. One of them is to detect overfitting that negatively impacts the performance of the model on new data. Furthermore, CV provides a more generalized performance metric than a single train-test split. Finally, CV maximizes the use of available data by allowing every observation to be used for both training and validation, and as such, the model training can benefit from the entire dataset.

By applying 10-fold CV the results were:

Model	MAE	MSE	R2
Random Forest Regressor	0.080887	0.016240	0.272739
Gradient Boosting Regressor	0.045091	0.008538	0.806296
Support Vector Regressor	0.084574	0.016443	0.236989
K-Nearest Neighbors	0.100600	0.027678	-0.255842
Linear Regression	0.057751	0.012808	0.757539
Elastic Net	0.197382	0.062745	1.475431

Figure 4.3: Metric table for the training of State Prediction Regression Model with every model type using CV with 10 folds

It is clear that **Gradient Boosting Regressor** stands out as the top performer with an R2 of 0.806, the lowest MSE at 0.008538, and the lowest MAE of 0.045091. Its superior performance suggests strong predictive capabilities and an excellent fit to the dataset, making it the preferred choice for predicting state and crop parameters in the simulator.

Linear Regression, surprisingly, shows strong performance with an R2 of 0.757539 and moderate error rates, indicating a good fit for datasets with linear characteristics. However,

its slightly higher error metrics compared to Gradient Boosting Regressor suggest some limitations in handling complex nonlinear relationships within the data.

4.1.2 Crop Parameters Prediction Regression Model

Similarly with the State Prediction Regression Model, a predefined set produced was used to train the models for each algorithm. After training, the models were used to make predictions on the dataset, which were then utilized to calculate the metrics MAE, MSE, RMSE and R2.

The results were:

Model	MAE	MSE	RMSE	R2
Random Forest Regressor	0.000522	1.064815e-05	0.003263	0.999848
Gradient Boosting Regressor	0.001659	5.093134e-06	0.002257	0.999911
Support Vector Regressor	0.047985	3.112529e-03	0.055790	0.943564
K-Nearest Neighbors	0.020611	1.459654e-03	0.038205	0.972726
Linear Regression	0.000636	7.546073e-07	0.000869	0.999986
Elastic Net	0.191737	6.149200e-02	0.247976	-0.001020

Figure 4.4: Metric table for the training of Crop Parameters Prediction Regression Model with every model type using the train_test_split

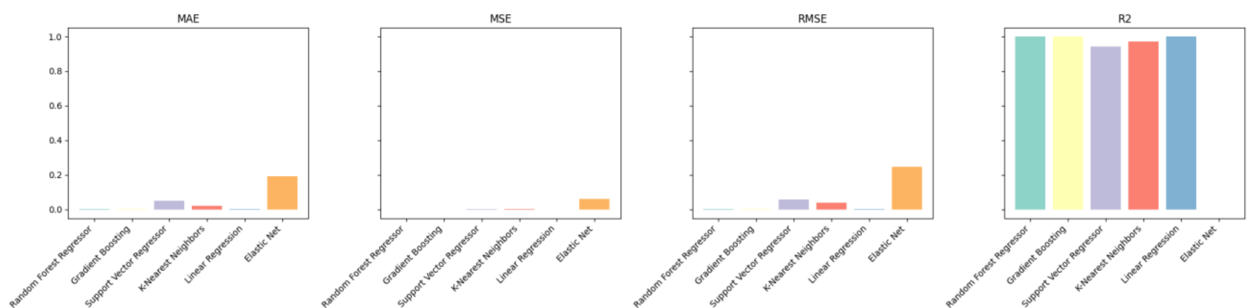


Figure 4.5: Metric plot or the training of Crop Parameters Prediction Regression Model with every model type using the train_test_split

All the models seem to perform exceptionally well with a very high R2 score and minimal MSE and MAE. This time the **Linear Regression** had the lowest MSE and high R2 value close to 1 and the **Random Forest Regressor** the lowest MAE**.** The results This suggest an almost perfect prediction with minimal error, indicating that the linear model is surprisingly effective, potentially due to a linear or nearly linear relationship between features and the target in the dataset used for this split. The **Elastic Net** performed poorly once again, with a negative R2 indicating that the model performs worse than a simple horizontal line mean model.

For the same reasons as for the State Prediction Regression Model, CV was applied for further evaluation.

By applying 10-fold CV the results were:

Model	MAE	MSE	R2
Random Forest Regressor	0.037598	0.003679	0.763458
Gradient Boosting Regressor	0.003859	0.000128	0.996679
Support Vector Regressor	0.074762	0.008725	0.567697
K-Nearest Neighbors	0.142893	0.040026	-0.935228
Linear Regression	0.000817	0.000048	0.998314
Elastic Net	0.206359	0.069178	-1.886438

Figure 4.6: Metric table for the training of Crop Parameters Prediction Regression Model with every model type using CV with 10 folds

Gradient Boosting Regressor and **Linear Regression** clearly stand out with extremely high R2 values of 0.996679 and 0.998314, respectively, indicating that both models are highly predictive of the crop parameters with minimal error (MSE and MAE).

Even though Linear Regression has slightly better metrics, it is important to note that the expected output that the model tries to predict is the three crop parameters which have been

linearly interpolated during the data expansion in the section **3.3.3**. Real-world data might exhibit non-linear behaviors not captured through interpolation, whereas Gradient Boosting can adapt to potential non-linearities better than Linear Regression. Therefore, choosing Gradient Bosting Regressor algorithm appears to be a strategic decision based on its performance, adaptability, and robustness.

4.1.3 Algorithm Justification

Multi-output **Gradient Boosting Regression** was selected to train both predictive models as it performed the best in both cases. It is specifically designed to handle scenarios where multiple output predictions are required from the same set of inputs.

In the context of a greenhouse, where outputs (e.g., humidity levels, temperature, CO2 levels) might not only be interdependent but also vary widely in their nature and distribution, a multioutput framework is essential. This model trains individual gradient boosting regressors for each output parameter but does so within a coherent framework that manages all outputs simultaneously. This capability is crucial for maintaining consistency across predictions of different parameters, each of which may influence the others.

Also, Gradient Boosting is inherently robust to a variety of input distributions due to its decision-tree-based approach. This makes them particularly adept at managing features with different distributions such as Gaussian, skewed, or flat, as typically seen in greenhouse data sets where factors like humidity, CO2 levels, and light intensity can vary in distribution.

Gradient Boosting Regression constructs a strong predictive model by sequentially adding trees that correct the residuals of the previous trees. This iterative correction allows the model to effectively capture complex non-linear relationships between features and outputs. In greenhouse environments, the interactions between different environmental factors and their effects on plant growth can be highly non-linear and influenced by multiple interacting factors. Gradient Boosting’s capacity to model these relationships makes it highly effective for such applications.

Additionally, Gradient Boosting includes several mechanisms for regularization, such as the number of trees, depth of trees, and learning rate, which help prevent overfitting.

4.2 DRL algorithms

In this section, we explore the application of two prominent DRL algorithms: **Proximal Policy Optimization** (**PPO**) and **REINFORCE**. These algorithms represent two different approaches to solving reinforcement learning problems, each with unique characteristics and strengths as described in the section **3.6**.

However, theoretically, REINFORCE is not as robust or efficient as PPO especially in environments that require high stability and reliable policy behaviour over time. REINFORCE is a basic policy gradient method that makes a direct proportional update to the policy based on the gradient of the expected returns. The direct nature of the update can introduce several weaknesses, particularly in complex learning problems.

Its updates are often characterized by high variance due to the fact that calculating the gradient requires a full trajectory, and so the output of the gradient calculation at any single time steps depends heavily on the rewards and actions from start to finish. High variance is often a sign of unstable training because policies may fluctuate and fail to converge to optimality.

Similarly, because the algorithm requires a full trajectory to make updates, it can be inefficient in terms of computational complexity. In cases where rollout trajectories are expensive or slow to collect, the variance in policy can increase. As a result, many episodes are required to make a learning update sufficiently reliable when using REINFORCE. This often makes the approach much slower to converge than more advanced methods and also less sample-efficient.

Whereas PPO addresses many of the limitations seen in REINFORCE through several key improvements. It introduces a clipped surrogate objective, limiting the policy update at each iteration. This procedure acts as an upper bound and prevents the software from making implausibly large updates that might destabilize the policy. However, it ensures the updates are sizable enough to meaningfully improve the learning rather than undermining the strengths of an already decent policy.

In the upcoming sections of the thesis, I will present empirical results demonstrating these theoretical expectations. The experiments are designed to compare the performance of

REINFORCE and PPO based on the cumulative reward. The results are expected to validate the theoretical advantages of PPO over REINFORCE, providing a practical perspective on why PPO is often favoured in applications requiring robust and efficient learning. The effectiveness and efficiency of these algorithms were evaluated through two main strategies:

- Cumulative Reward During Training: This metric is one of the most important in that it quantifies the actual performance of each policy over time. Since the key end for most RL applications is to maximize the accumulated reward, the cumulative reward during training provides an insight into how quickly each of the algorithms learn to approach this goal. By actually plotting the dynamics of the accumulated rewards throughout the training, we will be able to understand which of the algorithms learns more stably and quickly. Therefore, this figure will provide the information needed to practically apply the model.
- Cumulative Reward in a Test Environment: Following the training phase, both algorithms were further tested in a new test environment for ten episodes. By measuring the total rewards accumulated in each episode of this test phase, we assess the robustness and reliability of the policies.

Each algorithm as mentioned in the section **3.6.3** was trained for 100 iterations of 1 episode each, 4000 timesteps per episode.

4.2.1 PPO

The PPO was trained with the Adam optimizer. In the experiments, three different learning rates (LRs) were used for the Adam optimizer: 0.01, 0.001 and 0.0001. For each learning rate, two plots are provided: A plot of the cumulative reward for each of the 100 iterations during the training and a plot of policy evaluation which the policy is applied on a new test environment for 10 episodes.

Learning Rate: 0.01

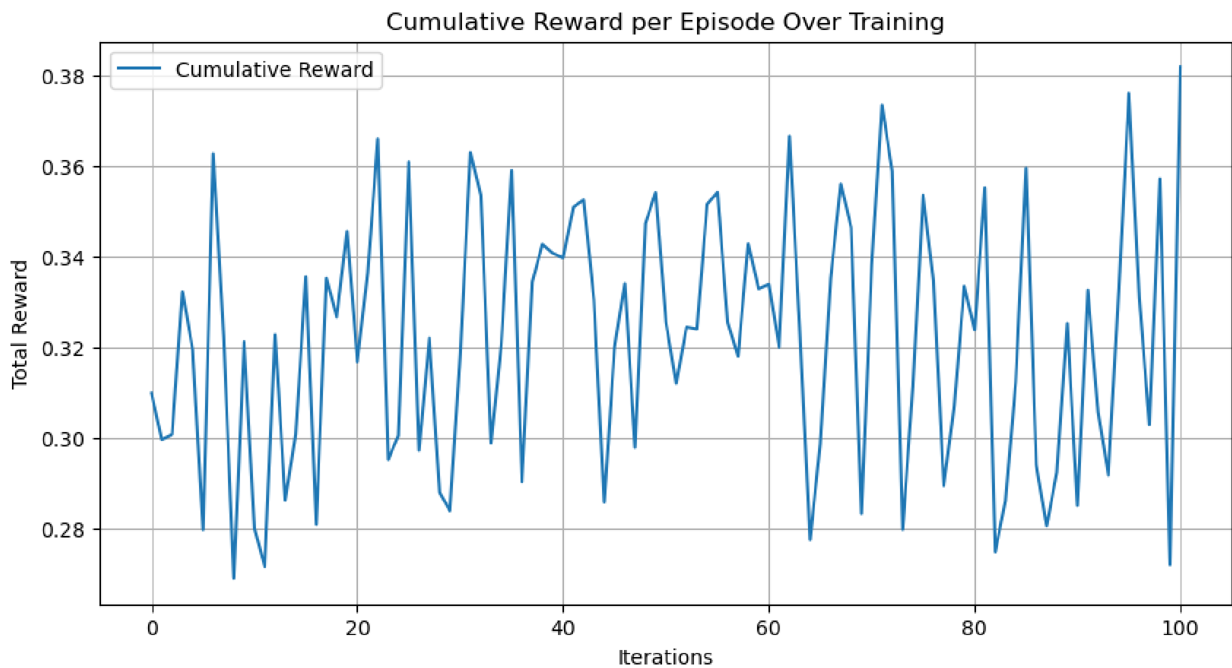


Figure 4.7: Cumulative reward for PPO training with LR 0.01

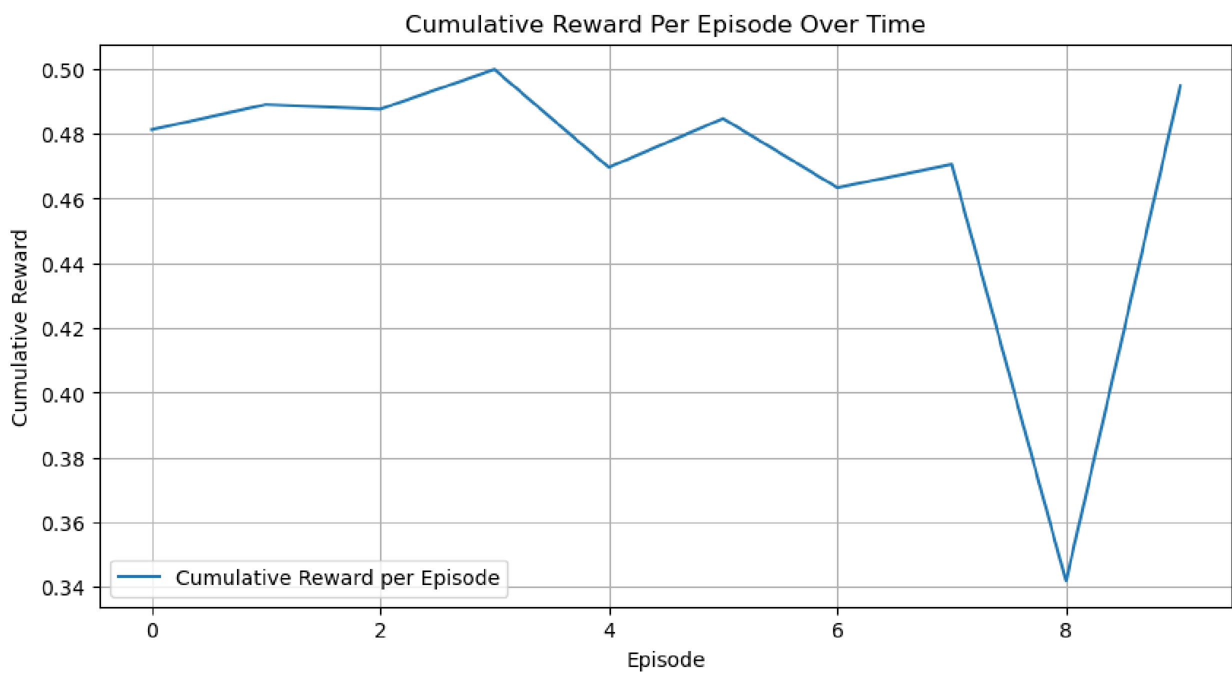


Figure 4.8: Cumulative reward for PPO evaluation trained with LR 0.01

Learning Rate: 0.001

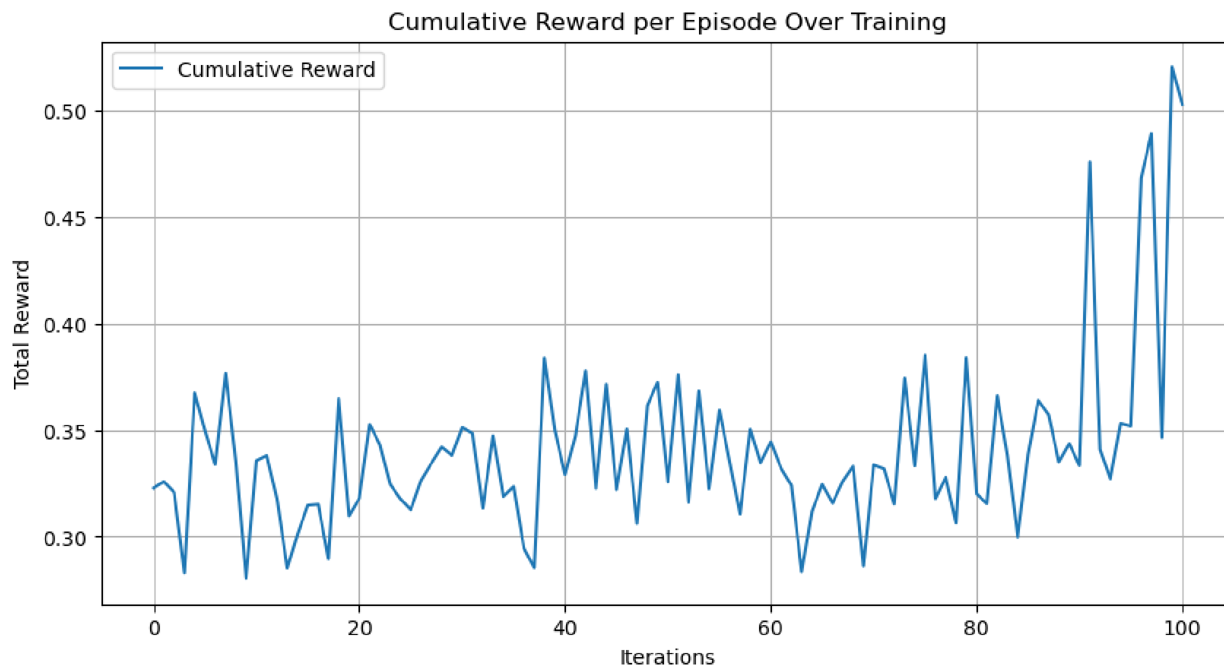


Figure 4.9: Cumulative reward for PPO training with LR 0.001

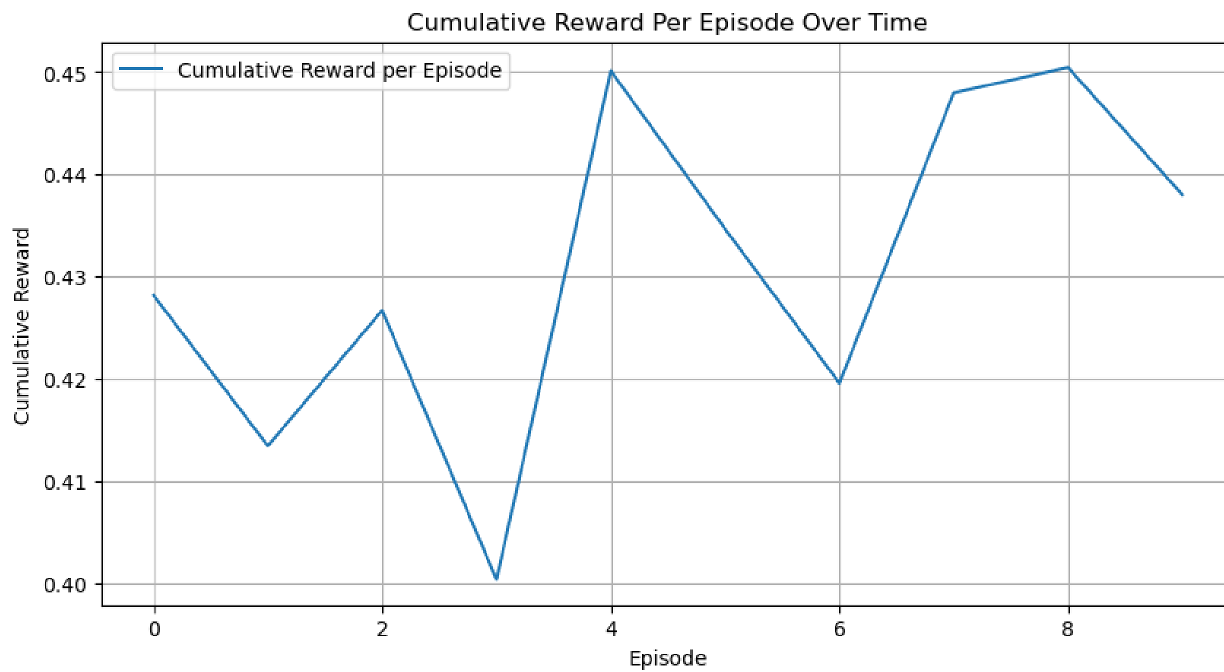


Figure 4.10: Cumulative reward for PPO evaluation trained with LR 0.001

<u>Learning Rate: 0.0001</u>

Figure 4.11: Cumulative reward for PPO training with LR 0.0001

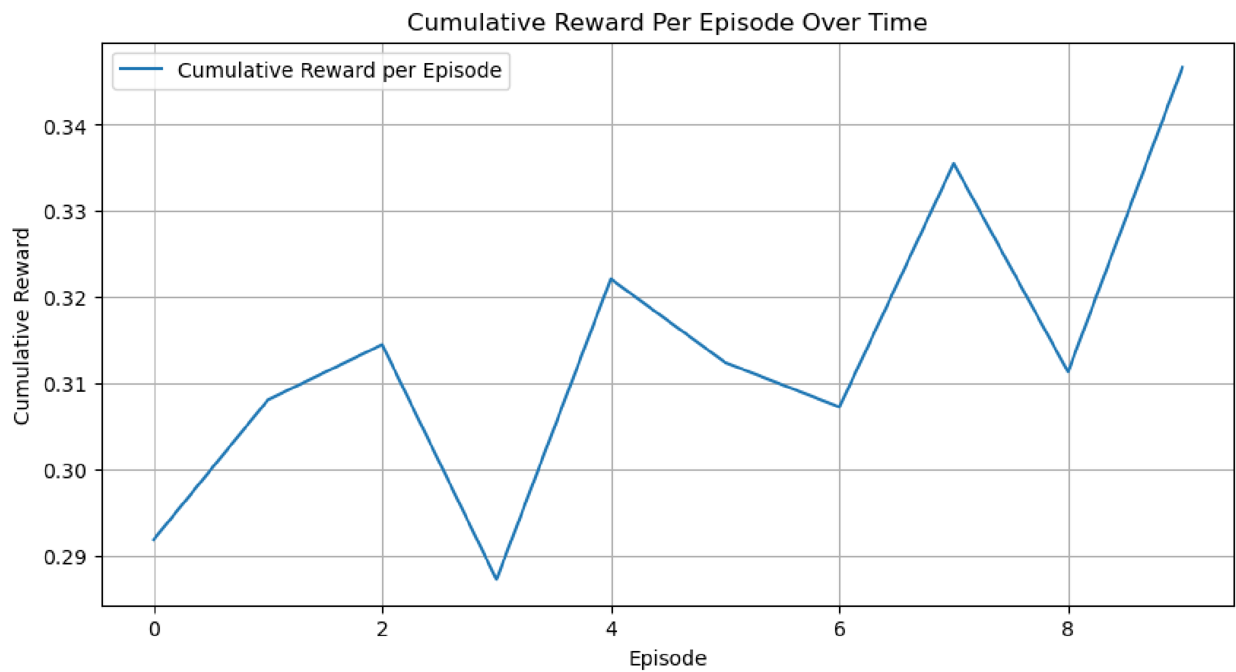


Figure 4.12: Cumulative reward for PPO evaluation trained with LR 0.0001

Discussion:

After observing the graphs it is clear that the best learning rate is 0.001 since the learning rate is gradually increasing, specifically in the first graph the cumulative reward has overcome 0.50 at in the last iterations. Additionally, the policy with 0.001 Adam optimizer learning rate has achieved cumulative rewards in the range of [0.40, 0.45].

A lower learning rate does not show significant improvement as the learning is too slow and not many changes are applied in each iteration. Similarly, a higher learning rate does not seem to be effective either since the cumulative reward during the training is too noisy due to the rapid big changes. However, with a bigger learning rate the evaluation had surprisingly good evaluation results except one episode that had 0.34 cumulative reward. This could mean that the model adapted to certain aspects of the environment, but its volatility is evident where the cumulative reward dropped in that one episode.

4.2.2 REINFORCE

The REINFORCE was also trained with the Adam optimizer. In the experiments, the same learning rates as in PPO were used for the optimizer and the training was 100 iterations and the evaluation 10.

Learning Rate: 0.01

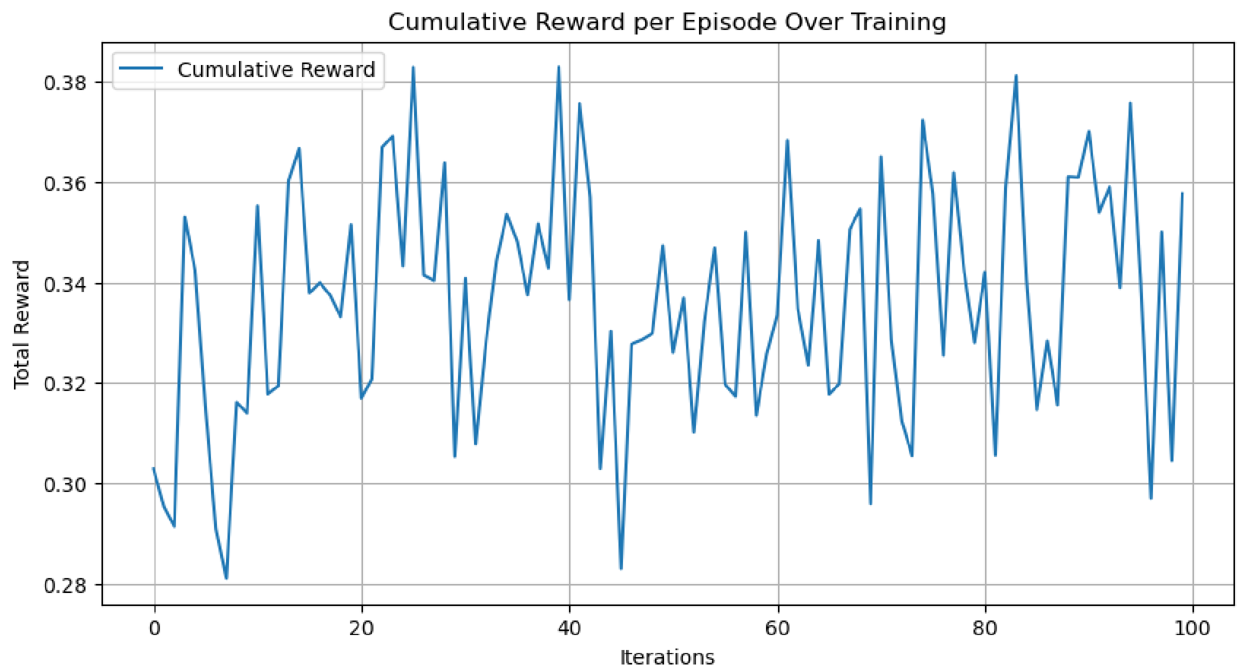


Figure 4.13: Cumulative reward for REINFORCE training with LR 0.01

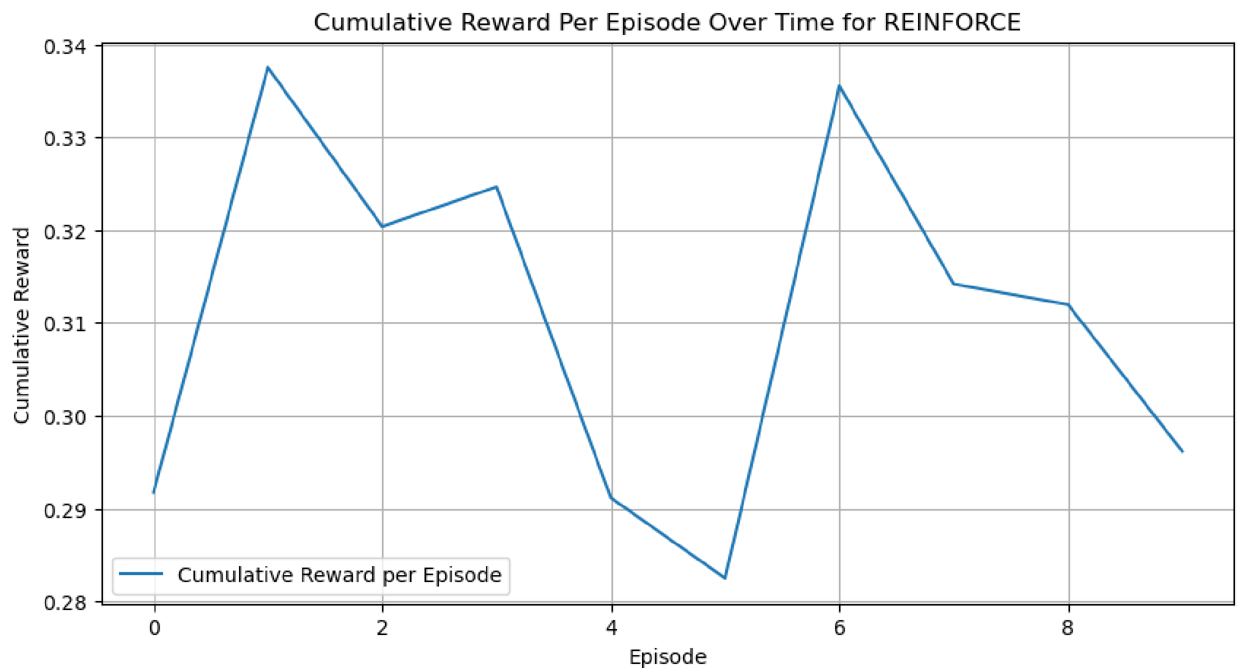


Figure 4.14: Cumulative reward for REINFORCE evaluation trained with LR 0.01

Learning Rate: 0.001

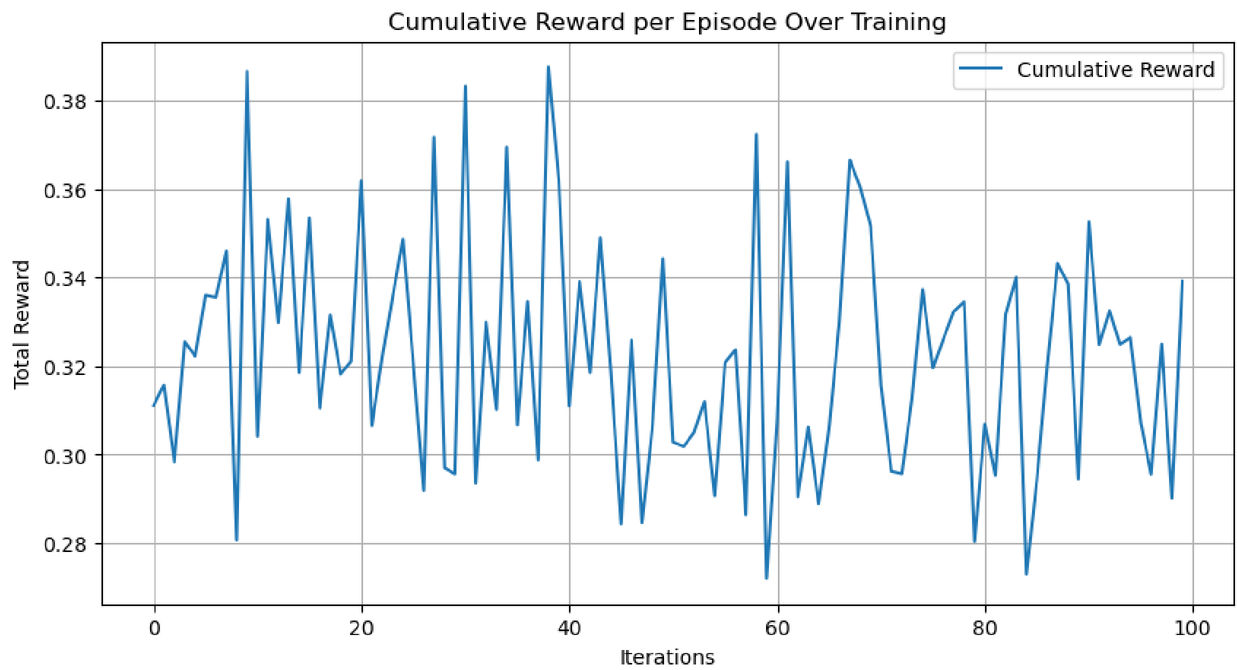


Figure 4.15: Cumulative reward for REINFORCE training with LR 0.001

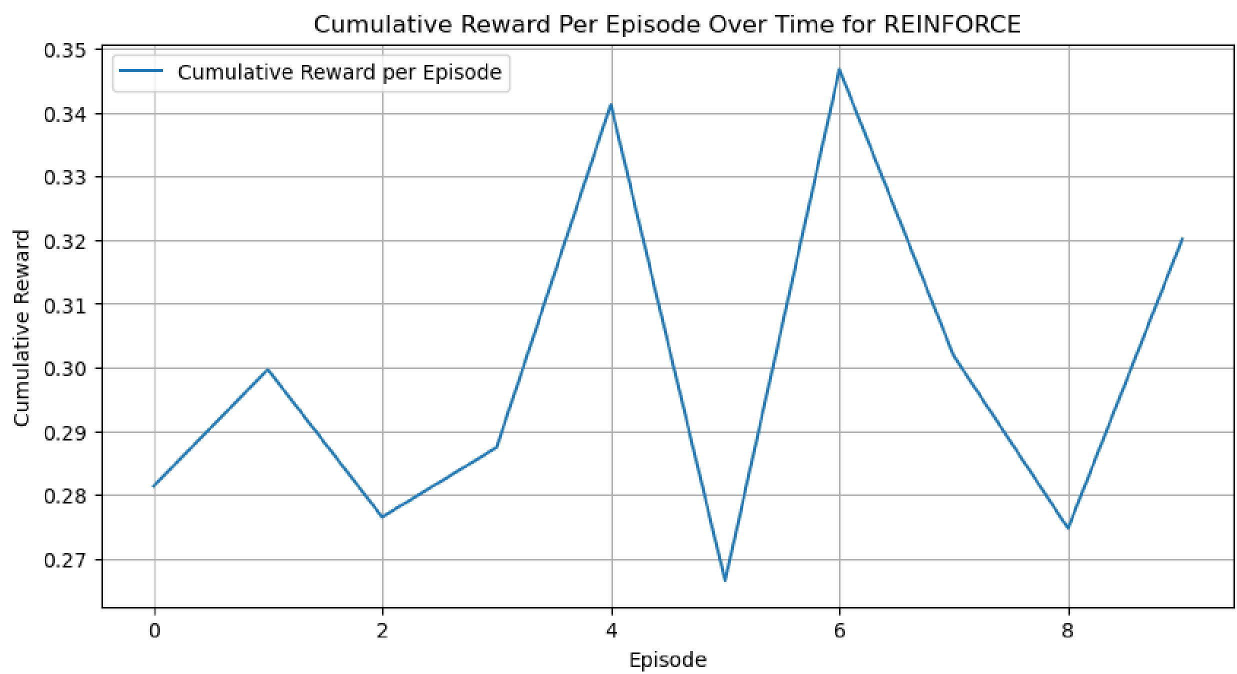


Figure 4.16: Cumulative reward for REINFORCE evaluation trained with LR 0.001

Learning Rate: 0.0001

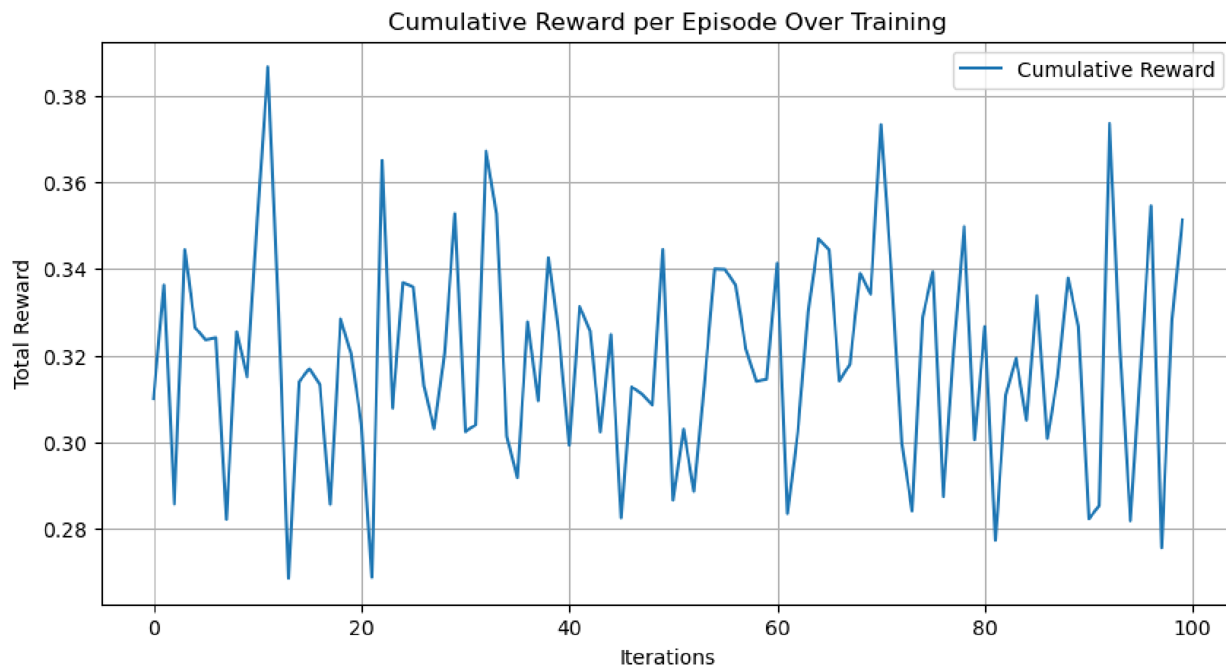


Figure 4.17: Cumulative reward for REINFORCE training with LR 0.0001

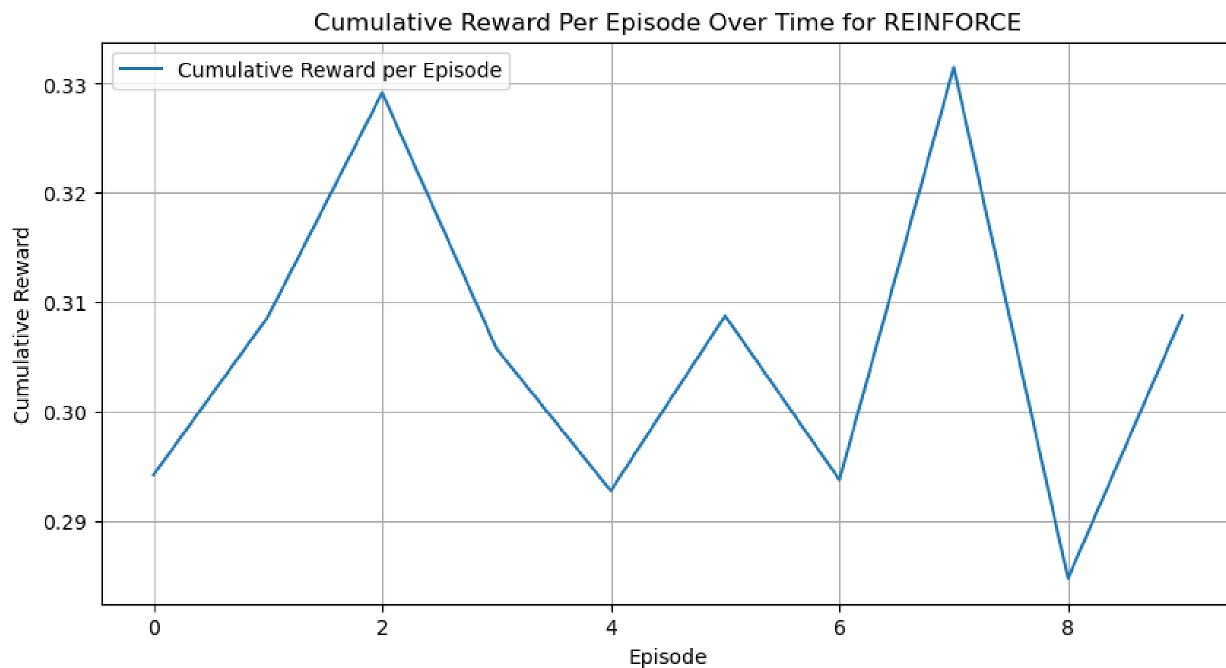


Figure 4.18: Cumulative reward for REINFORCE evaluation trained with LR 0.0001

Discussion:

In the exploration of the REINFORCE algorithm, the impact of different learning rates on model performance was distinctly less favourable compared to the results observed with the PPO algorithm. Across various learning rate settings, REINFORCE consistently struggled to achieve the efficiency and effectiveness demonstrated by PPO. Despite experimenting with multiple rates, none yielded particularly strong outcomes within the scope of 100 training iterations.

The analysis underscores that while REINFORCE can operate with a bigger learning rate, it does not inherently translate to better performance but rather to an increased range of outcomes, which sometimes captures higher peaks in cumulative rewards. This variability, paired with generally lower efficiency compared to PPO, suggests that REINFORCE might require additional iterations to truly optimize and stabilize its policy learning. In scenarios where training duration can be extended, and model stability is less critical, a higher learning rate for REINFORCE might prove more advantageous, albeit still trailing the performance capabilities and consistency offered by PPO.

4.2.3 Best algorithm

The graphs in Chapter 4 indicate that PPO with a 0.001 learning rate is the best performing model due to its stable and consistently improving cumulative rewards. Specifically, Figure 4.9 and Figure 4.10 show that the cumulative rewards for PPO with a 0.001 learning rate exceed 0.50 in the final iterations of training and achieve a range of [0.40, 0.45] during evaluation.

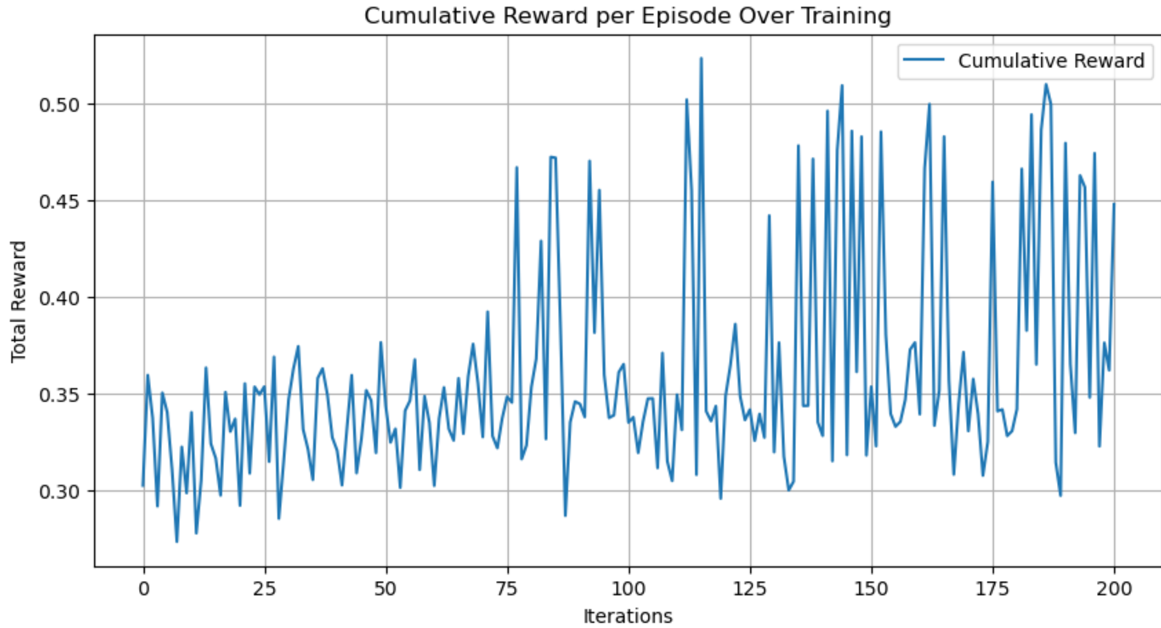


Figure 4.19: Cumulative reward for PPO training for 200 episodes with LR 0.001

After running the best algorithm, PPO, with the optimal learning rate, we observe significant learning progress as evidenced by Figure 4.19. The cumulative reward graph indicates a clear upward trend over the 100 episodes, suggesting that the agent is increasingly improving its performance. During training, the PPO algorithm continuously explores various actions to discover the most rewarding strategies. This exploration can lead to fluctuations in performance as the agent sometimes tries suboptimal actions to ensure it is not missing out on potentially better policies. Additionally, the complexity and variability of the greenhouse environment, combined with the stochastic nature of the learning process, contribute to these oscillations.

Chapter 5

Conclusion and Future Work

5.1 Conclusion 69

5.2 Future work 71

5.1 Conclusion

This research confirms that Deep Reinforcement Learning (DRL) can effectively address complex problems within controlled agricultural environments, providing robust tools for optimizing greenhouse operations. By leveraging DRL, we demonstrate that these techniques are not only theoretically sound but also practically applicable in real-life settings. This application of DRL has the potential to revolutionize traditional farming methods by enabling more precise control over environmental factors and resource utilization, significantly contributing to sustainable agriculture. Implementing such AI-driven systems can lead to increased crop yields, reduced resource waste, and enhanced economic viability, making food production more sustainable and efficient as global agricultural demands continue to grow.

In this study, the use of an environment simulator was particularly beneficial given the constraints specific to the problem at hand. Direct interaction with the greenhouses was not feasible, and the available dataset for training was relatively small, limiting the potential for extensive empirical testing. Furthermore, the application of Deep Reinforcement Learning (DRL) typically requires large amounts of data to effectively capture the complex dynamics of the environment it seeks to optimize. The environment simulator addressed these challenges by providing a virtual platform where the DRL algorithms could be intensively tested and refined. This approach allowed for the simulation of numerous scenarios to

evaluate the performance and impact of various strategies without the need for direct, real-world experimentation, thus conserving resources and enabling more iterative enhancements in a controlled and cost-effective manner.

In this project, cross-validation (CV) was chosen over a traditional train-test split due to the limited size of the dataset. CV allows for a more robust evaluation by utilizing the entire dataset for both training and validation, ensuring that every data point is used for both purposes across different iterations. This method reduces the risk of model overfitting and provides a more comprehensive understanding of the model's performance, which is crucial in situations where data are not abundant. Opting to train and average predictions across 10 models further enhanced the reliability and stability of the results. This approach averages out the noise and potential biases of individual models, leading to more generalized and robust predictions.

Choosing multi-output Gradient Boosting for regression tasks was particularly effective because of its capability to handle the intricacies and non-linear relationships inherent in agricultural datasets. Gradient Boosting builds models sequentially, improving with each iteration by focusing on the hardest to predict instances, which makes it adept at handling complex and varied data structures. This characteristic was crucial for predicting multiple dependent variables simultaneously, which is often required in agricultural modeling to account for the various interacting parameters that influence crop growth and resource usage. The strength of Gradient Boosting in capturing complex interactions within the data made it the superior choice compared to other models, which might not effectively handle the multi-dimensional outputs or might not adapt as efficiently to the evolving nuances of the dataset.

Among the learning algorithms evaluated, Proximal Policy Optimization (PPO) was identified as the superior choice over REINFORCE, which was an expected outcome based on the specific characteristics of each method. The learning rate of 0.01 for PPO was found to be optimal, balancing rapid convergence with stability to prevent significant performance oscillations. Other learning rates tested included more aggressive rates such as 0.001 and 0.01, but these either resulted in slower convergence or too much instability in policy updates. The use of the Adam optimizer with PPO further enhanced its performance by efficiently

handling the gradient descent, optimizing the learning process to achieve more reliable and robust policy improvements. This combination of a carefully tuned learning rate and a sophisticated optimizer ensures that PPO not only learns effectively but also adapts consistently to the complexities of the simulated agricultural environment.

In summary, the integration of DRL into greenhouse management through the use of advanced simulation tools and precisely selected predictive models represents a significant stride in agricultural technology. The strategic use of Proximal Policy Optimization (PPO) and meticulous calibration of its learning rate illustrate the potential of AI to significantly enhance the efficiency and sustainability of food production globally. Despite the limited span of only 200 iterations (episodes), the results clearly demonstrate that the agent is effectively learning, as evidenced by the increase in cumulative rewards as the episodes progress. Furthermore, the crop parameters have shown consistent improvement, reinforcing the effectiveness of the applied AI strategies. At the conclusion of the last episode, both the actions taken and the state of the environment exhibit a positive alignment with the correlation map of the features with the crop parameters. This alignment further substantiates the appropriateness of the chosen AI methods and their implementation, highlighting their capability to optimize agricultural outputs in real-world scenarios.

5.1 Future work

Given the constraints on time and computational resources during the development of this system, there are several key areas for improvement that could enhance the model's performance and its applicability in real-world scenarios:

- Enhance Prediction Models: For future work, it is therefore critical to carry out systematic hyperparameter tuning of the conducted Gradient Boosting algorithms in order to refine the prediction models. Specifically, all major parameters including the learning rate, number of trees, tree depth, minimum samples split, and maximum features must be adjusted meticulously to suit the characteristics of agricultural data. These changes have the potential to drastically improve the accuracy and efficiency of the models and make sure they remain simple enough to prevent overfitting. This

targeted activity is essential to guarantee that the models not only make precise predictions but also remain strong and effective across all potential case scenarios.

- Explore Non-Policy Based Algorithms: Exploring a broader array of Deep Reinforcement Learning (DRL) algorithms, including those outside of policy-based approaches, could significantly enhance the understanding and management of greenhouse environments. Deep Q-Learning, an extension of the traditional Q-learning algorithm, employs deep neural networks to approximate the Q-value function, which could be particularly effective in stable environments where the value of actions does not change drastically. This method allows for a robust assessment of action outcomes, facilitating solid decision-making. Additionally, incorporating advanced actor-critic methods like Soft Actor-Critic (SAC), which optimizes a trade-off between entropy (exploration) and reward (exploitation), could offer a balanced approach to learning. SAC, in particular, is designed to function well in environments with continuous action spaces, making it suitable for the nuanced control required in agricultural settings. Each of these DRL categories addresses learning and decision-making from a different angle, potentially leading to more effective strategies for maximizing crop yield and resource efficiency.
- Increase Number of Episodes: Running more episodes could help in achieving a more complete convergence of the learning algorithms, as the cumulative reward was still showing an upward trend at the conclusion of 200 episodes. Extended training might allow the model to explore a wider range of strategies and refine its approach, potentially surpassing the performance of the teams' systems.
- Expand Data Sources and Crop Parameters: Expanding the dataset by utilizing more CSV files and additional crop parameters could significantly enhance the model's performance by providing a richer, more varied dataset. This approach not only improves the robustness and accuracy of the model by covering more variability in the input data but also ensures a broader representation of possible greenhouse conditions and crop behaviours. Such expansion would enable a more detailed simulation of real-world scenarios, thereby increasing the generalizability and reliability of the model predictions.

- Adjust Reward Function Weights: Modifying the weights of the reward function could substantially impact the learning outcomes by aligning the model's focus more closely with specific agricultural goals. This adjustment allows for a tailored approach where certain crop parameters can be prioritized according to their importance, thus directly influencing the strategic decisions made by the AI in real-time scenarios.
- Experiment with alternative Optimizers: While the Adam optimizer is renowned for its efficiency and adaptive learning rates, exploring alternative optimization algorithms such as SGD or RMSprop could uncover additional improvements in model performance.

References

[1] Growlink (no date). Embracing the Power of IoT and AI in Greenhouse Farming. (Accessed: 02 January 2024) https://blog.growlink.com/embracing-the-power-of-iot-and-ai-in-greenhouse-farming

[2] Hemming, S., de Zwart, F., Ellings, A., Righini, I., and Petropoulou, A. (2019). Remote control of greenhouse vegetable production with artificial intelligence-greenhouse climate, irrigation, and crop production, Sensors, 19(8), 1807. https://doi.org/10.3390/s19081807

[3] Heroseo (2020). Autonomous Greenhouse Challenge (AGC) - 2nd edition, Kaggle. (Accessed: 02 January 2024) https://www.kaggle.com/datasets/piantic/autonomous-greenhouse-challengeagc-2nd-2019

[4] GeeksforGeeks (no date). What is Reinforcement Learning? (Accessed: 02 January 2024) https://www.geeksforgeeks.org/what-is-reinforcement-learning/

[5] Levine, S., Kumar, A., Tucker, G. and Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on Open problems, arXiv preprint, arXiv:2005.01643.

[6] GeeksforGeeks (no date). A Beginner's Guide to Deep Reinforcement Learning. (Accessed: 02 January 2024) https://www.geeksforgeeks.org/a-beginners-guide-to-deep-reinforcement-learning/

[7] Kalra, R. (no date). Understanding Model-Based Reinforcement Learning. (Accessed: 02 January 2024) https://medium.com/@kalra.rakshit/understanding-model-based-reinforcement-learning-b9600af509be

[8] Gandhi, R. (2022). Deep Reinforcement Learning for Agriculture: Principles and Use Cases. In: Reddy, G.P.O., Raval, M.S., Adinarayana, J., Chaudhary, S. (eds), Data Science in Agriculture and Natural Resource Management. Studies in Big Data, Singapore: Springer, vol 96, pp. 75-94.

[9] Garazhian, S. (2023). Data-driven Deep Reinforcement Learning for Agriculture, CYENS Internship Report.

[10] Demosthenous, G., Kyriakou, M., and Vassiliades, V. (2022). Deep reinforcement learning for improving competitive cycling performance. Expert Systems With

Applications, 203, 117311. (Accessed: 02 January 2024) https://doi.org/10.1016/j.eswa.2022.117311

[11] Han, D. et al. (2023). Deep reinforcement learning for irrigation scheduling using high-dimensional sensor feedback, arXiv preprint, arXiv:2301.00899.

[12] TensorFlow (no date). TF Agents: PyEnvironment. (Accessed: 02 January 2024) https://www.tensorflow.org/agents/api_docs/python/tf_agents/environments/PyEnvironment

[13] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms, arXiv preprint, arXiv:1707.06347.

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

Appendix A: Dataset

A1 GreenhouseClimate.csv

(All taken with a 5-minute interval)

A1.1 Indoor climate, status of actuators and irrigation

Column Heading	Parameter Description	Unit	Comments
Tair	Greenhouse Air temperature	°C	-
Rhair	Greenhouse relative humidity	%	-
CO2air	CO_2 greenhouse	ppm	-
HumDef	Greenhouse humidity deficit	g/m^3	-
VentLee	Leeward vents opening	% [0 to 100]	-
Ventwind	Windward vents opening	% [0 to 100]	-
AssimLight	HPS lamps status (on-off)	% [0 or 100]	-
EnScr	Energy curtain opening	% [0 to 100]	-
BlackScr	Blackout curtain opening	% [0 to 100]	-

PipeLow	Rail pipe Temperature (Lower circuit)	°C	-
PipeGrow	Crop pipe Temperature (Growth circuit)	°C	-
co2_dos	CO2 dosing	kg/ha hour	Computed CO2 dosage and calibrated by monthly CO2- meter readings.
Tot_PAR	Total inside PAR (Sun + HPS + LED)	μmol/m^2 s	Computed based on outdoor PAR, cover transmissivity (0.5), operation and transmissivity of energy (0.75) and blackout screens (0.02), PAR from LED and HPS.
Tot_PAR_Lamps	PAR sum from HPS and LED lamps	μmol/m^2 s	Computed based on lamps' operation and measured PPFD contribution of HPS (100 μmol/m^2 s) and LED (Blue = 11, Red = 49, Farred = 0, White = 37 μmol/m^2 s) when set at maximum range of LED proportional control (=1000)
EC_drain_PC	Drain EC	dS/m	-
pH_drain_PC	Drain pH	[-]	-
Water_sup	Cumulative number of minutes of irrigation in a day	minutes	This cumulation is reset to 0 at midnight.

Cum_irr	Cumulative number of litres of irrigation in a day	L/m^2 day	Conversion from minutes to liter. This cumulation is reset to 0 at midnight.

A1.2 Climate and irrigation setpoints

Column Heading	**Parameter Description**	**Unit**	**Comments**
co2_sp	CO2 setpoint	ppm	-
dx_sp	Humidity deficit setpoint	g/m^3	-
t_rail_min_sp	Rail pipe minimum temperature setpoint	°C	-
t_grow_min_sp	Crop pipe minimum temperature setpoint	°C	-
Assim_sp	Assimilation lighting setpoint (HPS lamp)	% [0 or 100]	-
scr_enrg_sp	Energy curtain setpoint	% [0 to 100]	-
scr_blck_sp	Blackout curtain setpoint	% [0 to 100]	-
t_heat_sp	Heating temperature setpoint	°C	-
t_vent_sp	Ventilation temperature setpoint (leeward vents)	°C	-
window_pos_lee_sp	Lee side window position minimum	%	-

	setpoint (leeward vents)		
water_sup_int_sp_min	Water supply interval time setpoint	minutes	Interval time between the last and next irrigation turn
int_blue_sp	Intensity set of blue spectrum channel (LED lamps)	[0 to 1000] range of proportional control	LED light control was coupled with HPS control ,that is LED lamps can only be used when the HPS-lamps are switched on as well.
int_red_sp	Intensity set of red spectrum channel (LED lamps)	[0 to 1000] range of	As above
int_farred_sp	Intensity set of far-red spectrum channel (LED lamps)	[0 to 1000] range of proportional control	As above
nt_white_sp	Intensity set of white spectrum channel (LED lamps)	[0 to 1000] range of proportional control	As above

A1.3 VIP (realized setpoints)

Column Heading	Parameter Description	Unit	Comments
co2_vip	CO2 VIP	ppm	-
dx_vip	Humidity deficit VIP	g/m^3	-
t_rail_min_vip	Rail pipe minimum temperature VIP	°C	-

t_grow_min_vip	Crop pipe minimum temperature VIP	°C	-
Assim_vip	Assimilation lighting VIP (HPS lamp)	% [0 or 100]	-
scr_enrg_vip	Energy curtain VIP	% [0 to 100]	-
scr_blck_vip	Blackout curtain VIP	% [0 to 100]	-
t_heat_vip	Heating temperature VIP	°C	-
t_vent_vip	Ventilation temperature VIP (leeward vents)	°C	-
window_pos_lee_vip	Lee side window position minimum VIP (leeward vents)	%	-
water_sup_int_vip_min	Water supply interval time VIP	minutes	Interval time between the last and next irrigation turn
int_blue_vip	Intensity set of blue spectrum channel VIP (LED lamps)	[0 to 1000] range of proportional control	LED light control was coupled with HPS control ,that is LED lamps can only be used when the HPS-lamps are switched on as well.
int_red_vip	Intensity set of red spectrum channel VIP (LED lamps)	[0 to 1000] range of	As above
int_farred_vip	Intensity set of far-red spectrum channel VIP (LED lamps)	[0 to 1000] range of proportional control	As above

nt_white_vip	Intensity set of white spectrum channel VIP (LED lamps)	[0 to 1000] range of proportional control	As above

A2 Crop Parameters
(All taken with a 1-week interval)

Column Heading	Parameter Description	Unit	Comments
Stem_elong	Stem growth per week	cm/week	This refers to 10 sample stems (average value). Data no later than 22 April (because tomato plants were topped).
Stem_thick	Stem thickness	mm	This refers to 10 sample stems (average value) Data no later than 22 April (because tomato plants were topped)
Cum_trusses	Cumulative number of new set trusses on the stem.	number/stem	This refers to 10 sample stems (average value). A truss is considered as “set” Manual registration when at least 5 flowers are set.
stem_dens	Stem density	Stems/m^2	-
Plant_dens	Plant density	Plants/m^2	-

Appendix B: Code

B1 Data Processing code for State Prediction Model

Importing libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
import os
from sklearn.model_selection import train_test_split
import xgboost as xgb
from sklearn.metrics import mean_absolute_error
import pickle
import os
from scipy.stats import iqr
```

Functions to clean and process a team's dataset

```
repository_url = "https://github.com/CEAOD/Data.git"
!git clone {repository_url}
path = '/content/Data/GH_Tomato/AutonomousGreenhouseChallenge2019'

def get_folders_list(path):
  # get all items from path
  items = os.listdir(path)
  # if item is folder, store in folders list
  folders = [item for item in items if os.path.isdir(os.path.join(path,
item))]
  print("Folders in the directory:")
  print(folders)
  return folders

def interpolate_nan(series):
    # Fill NaN values by interpolating
    series_interpolated = series.interpolate(method='linear',
limit_direction='both')
    return series_interpolated


def get_prepare_data(team_name):
  print("@team name is:  " + team_name)
  main_path = "/content/Data/GH_Tomato/AutonomousGreenhouseChallenge2019"
  path = main_path + "/" + team_name



  df_GreenhouseClimate = pd.read_csv(path + "/GreenhouseClimate.csv")
  print("coluns of df_"+ team_name+ "_GreenhouseClimate: \n",
df_GreenhouseClimate.columns)
  df_GreenhouseClimate = df_GreenhouseClimate.astype(float)
  print("number of rows:", df_GreenhouseClimate.shape[0])
```

```
  df_GreenhouseClimate =
df_GreenhouseClimate.dropna(subset=["AssimLight"]) #delete 71 features
which are common null in many features.
  df_GreenhouseClimate =
df_GreenhouseClimate.dropna(subset=["int_blue_sp"]) # delete 14 value.
  # if Cum_irr is more than 10 it is outlier (since it is not logical and
it is an unit related error.)
  df_GreenhouseClimate['Cum_irr'] =
df_GreenhouseClimate['Cum_irr'].apply(lambda x: x / 10 if x > 7 else x)

  df_GreenhouseClimate = df_GreenhouseClimate.reset_index(drop=True)


  # Fill NaN values for each column in the DataFrame
  for col_i in df_GreenhouseClimate.columns:
      df_GreenhouseClimate[col_i] =
interpolate_nan(df_GreenhouseClimate[col_i])

  # POST-INTERPOLATION cleaning
  percentage_features = ['VentLee', 'Ventwind', 'EnScr', 'BlackScr',
'scr_enrg_sp', 'scr_blck_sp']
  for feature in percentage_features:
      df_GreenhouseClimate[feature] =
df_GreenhouseClimate[feature].clip(lower=0, upper=100)

  # Features that are binary should be 0 or 100 as described in the
ReadMe pdf
  binary_features = ['AssimLight', 'assim_sp', 'assim_vip',
'scr_enrg_vip', 'scr_blck_vip']
  for feature in binary_features:
      df_GreenhouseClimate[feature] =
df_GreenhouseClimate[feature].apply(lambda x: 0 if x < 50 else 100)

  # Ensure irrigation volumes are not negative
  df_GreenhouseClimate['Cum_irr'] =
df_GreenhouseClimate['Cum_irr'].clip(lower=0)
  df_GreenhouseClimate = df_GreenhouseClimate.reset_index(drop=True)


  nan_counts = df_GreenhouseClimate.isnull().sum()
  print("None counts after process:", nan_counts.sum())


  x_raw = df_GreenhouseClimate
  x_time = x_raw['%time']
  x_raw = x_raw.drop(columns=['%time'])

  x_train = []
  time_values = []  # To store the time values for each group

  for iter in range(0, len(x_raw), 12):
      selected_rows = x_raw[iter:iter+12]
      selected_rows = selected_rows.tail(12)
      average_row = selected_rows.mean()  # Calculate the mean for the
selected rows
      x_train.append(average_row)
```

```
      # Calculate the corresponding hour and add it to time_values
      current_time = (iter // 12) % 24
      time_values.append(current_time)

  # Create a DataFrame from the averaged rows
  df_x_train = pd.DataFrame(x_train)

  # Add the 'time' column to the DataFrame
  df_x_train['hour'] = time_values

  #print("len x_train", len(x_raw))
  return df_x_train
```

Merging each team's dataset

```
directory_path =
'/content/Data/GH_Tomato/AutonomousGreenhouseChallenge2019'
team_names = get_folders_list(directory_path)
team_names.remove('Weather')
team_names.remove('Reference')


x_new_list = []
lens = []

for team_iterator in range(0, len(team_names)):
  if (team_iterator == 0):
    x = get_prepare_data(team_names[team_iterator])
    lens.append(len(x))
  else:
    x_new = get_prepare_data(team_names[team_iterator])
    x = pd.concat([x, x_new], axis=0, ignore_index=True)
    lens.append(len(x))
x = x.fillna(x.mean())

x.isnull().sum()
```

Distribution plotting and normalizing

```
def plot_distributions_grid(df, rows=6):
    # Calculate the number of columns needed based on the number of rows
specified
    cols = int(np.ceil(len(df.columns) / rows))

    # Initialize the figure with subplots
    fig, axes = plt.subplots(rows, cols, figsize=(5*cols, 5*rows))
    axes = axes.flatten()  # Flatten the axes array for easy iteration

    for i, column_name in enumerate(df.columns):
        # Select the current axis for plotting
        ax = axes[i]
        sns.histplot(df[column_name], kde=True, color='blue', ax=ax)
        mean = df[column_name].mean()
        std_dev = df[column_name].std()

        # Highlight the standard deviation area around the mean
        ax.axvline(mean, color='k', linestyle='dashed', linewidth=1)
```

```python
        ax.axvspan(mean - std_dev, mean + std_dev, alpha=0.2,
color='blue')

        # Set the title and labels
        ax.set_title(f'Distribution of {column_name}')
        ax.set_xlabel(column_name)
        ax.set_ylabel('Density')

    # Hide any unused axes
    for i in range(len(df.columns), len(axes)):
        fig.delaxes(axes[i])

    plt.tight_layout()
    plt.show()

# Call the function to plot distributions for all features in a grid
plot_distributions_grid(x)

from sklearn.preprocessing import MinMaxScaler
import pandas as pd
import numpy as np

def normalize_dataframe(df):
    scaler = MinMaxScaler()
    df_normalized = pd.DataFrame(scaler.fit_transform(df),
columns=df.columns)
    return df_normalized, scaler

# Assuming 'x' is your original DataFrame
x_normalized, scaler = normalize_dataframe(x)

# Compute and print max, min, and IQR for all features
def compute_stats(df):
    stats = pd.DataFrame(index=['Max', 'Min', 'IQR'], columns=df.columns)
    for column in df.columns:
        stats.loc['Max', column] = df[column].max()
        stats.loc['Min', column] = df[column].min()
        stats.loc['IQR', column] = iqr(df[column])
    return stats

stats = compute_stats(x)
normalized_stats = compute_stats(x_normalized)
print(x_normalized.shape)

normalized_stats
```

Outliers' adjustment

```python
def adjust_outliers_less_aggressive(column):
    # Determine the IQR
    Q1 = column.quantile(0.25)
    Q3 = column.quantile(0.75)
    IQR_value = iqr(column)

    # Calculate the outlier thresholds using a wider range
    lower_bound = Q1 - 3 * IQR_value
    upper_bound = Q3 + 3 * IQR_value
```

```
    # Adjustment values just outside the less aggressive outlier
thresholds
    lower_adjustment = Q1 - 2.99 * IQR_value
    upper_adjustment = Q3 + 2.99 * IQR_value

    # Apply adjustments
    column = np.where(column < lower_bound, lower_adjustment, column)
    column = np.where(column > upper_bound, upper_adjustment, column)

    return column

# Apply the less aggressive outlier adjustment
x_adjusted = x_normalized.apply(adjust_outliers_less_aggressive)
print(x_adjusted.shape)
```

```
# Call the function to plot distributions for all features in a grid
plot_distributions_grid(x_adjusted)

# Re-plot the distributions for all features in a grid
plot_distributions_grid(x_adjusted)

# Re-compute and print max, min, and IQR for all features
adjusted_stats = compute_stats(x_adjusted)
adjusted_stats
```

Saving the dataset

```
x_adjusted.to_csv(folder_path + '/GreenhouseClimate_X_adjusted.csv',
index=False)
```

B2 Data Processing code for Crop Parameters Prediction Model

The only difference from B1 is the get_prepare_data() function

```
def get_prepare_data(team_name):
  print("@team name is:  " + team_name)
  main_path = "/content/Data/GH_Tomato/AutonomousGreenhouseChallenge2019"
  path = main_path + "/" + team_name

  df_CropParameters = pd.read_csv(path + "/CropParameters.csv")
  df_CropParameters = df_CropParameters.astype(float)
  print("columns of df "+team_name+" CropParameters: \n",
df_CropParameters.columns)
  #crop parameters are sampled weekly.
  y_raw = df_CropParameters[['%Time', 'Stem_elong', 'Stem_thick',
'Cum_trusses']]
  y_raw = y_raw.dropna()
  y_raw = y_raw.reset_index(drop=True)
  # y_time = y_raw['%Time']
  # y_raw = y_raw.drop(columns=['%Time'])
  # y_time = y_time.reset_index(drop=True)
```

```
  df_GreenhouseClimate = pd.read_csv(path + "/GreenhouseClimate.csv")
  print("coluns of df_"+ team_name+ "_GreenhouseClimate: \n",
df_GreenhouseClimate.columns)
  df_GreenhouseClimate = df_GreenhouseClimate.astype(float)
  print("number of rows:", df_GreenhouseClimate.shape[0])

  df_GreenhouseClimate =
df_GreenhouseClimate.dropna(subset=["AssimLight"]) #delete 71 features
which are common null in many features.
  df_GreenhouseClimate =
df_GreenhouseClimate.dropna(subset=["int_blue_sp"]) # delete 14 value.
  # if Cum_irr is more than 10 it is outlier (since it is not logical and
it is an unit related error.)
  df_GreenhouseClimate['Cum_irr'] =
df_GreenhouseClimate['Cum_irr'].apply(lambda x: x / 10 if x > 7 else x)

  df_GreenhouseClimate = df_GreenhouseClimate.reset_index(drop=True)

  null_counter = df_GreenhouseClimate.isnull().sum()
  max_null_len = null_counter.max() # maximum count of missing values in
any column

  df_GreenhouseClimate = df_GreenhouseClimate.reset_index(drop=True)

  # Fill NaN values for each column in the DataFrame
  for col_i in df_GreenhouseClimate.columns:
      df_GreenhouseClimate[col_i] =
interpolate_nan(df_GreenhouseClimate[col_i])

  # POST-INTERPOLATION cleaning
  percentage_features = ['VentLee', 'Ventwind', 'EnScr', 'BlackScr',
'scr_enrg_sp', 'scr_blck_sp']
  for feature in percentage_features:
      df_GreenhouseClimate[feature] =
df_GreenhouseClimate[feature].clip(lower=0, upper=100)

  # Features that are binary should be 0 or 100 as described in the
ReadMe pdf
  binary_features = ['AssimLight', 'assim_sp', 'assim_vip',
'scr_enrg_vip', 'scr_blck_vip']
  for feature in binary_features:
      df_GreenhouseClimate[feature] =
df_GreenhouseClimate[feature].apply(lambda x: 0 if x < 50 else 100)

  # Ensure irrigation volumes are not negative
  df_GreenhouseClimate['Cum_irr'] =
df_GreenhouseClimate['Cum_irr'].clip(lower=0)
  df_GreenhouseClimate = df_GreenhouseClimate.reset_index(drop=True)

    nan_counts = df_GreenhouseClimate.isnull().sum()
  print("None counts after process:", nan_counts.sum())

  x_raw = df_GreenhouseClimate
    # Convert %time columns in both DataFrames to datetime format if not
already
  df_GreenhouseClimate['%time'] = pd.to_datetime(x_raw['%time'])
```

```
  y_raw['%Time'] = pd.to_datetime(y_raw['%Time'])

  # Merge x_raw with y_raw on the time column
  df_merged = pd.merge(df_GreenhouseClimate, y_raw, how='left',
left_on='%time', right_on='%Time')

  # Interpolate the missing values in the columns that came from y_raw
  df_merged[['Stem_elong', 'Stem_thick', 'Cum_trusses']] =
df_merged[['Stem_elong', 'Stem_thick',
'Cum_trusses']].interpolate(method='linear')

  # Drop the %Time column from y_raw as it's redundant now
  df_merged.drop(['%Time'], axis=1, inplace=True)

  # x_time = x_raw['%time']
  # x_raw = x_raw.drop(columns=['%time'])

  df_merged = df_merged.drop(columns=['%time'])

  x_train = []
  time_values = []  # To store the time values for each group

  for iter in range(0, len(df_merged), 12):
      selected_rows = df_merged[iter:iter+12]
      selected_rows = selected_rows.tail(12)
      average_row = selected_rows.mean()  # Calculate the mean for the
selected rows
      x_train.append(average_row)
      # Calculate the corresponding hour and add it to time_values
      current_time = (iter // 12) % 24
      time_values.append(current_time)

  # Create a DataFrame from the averaged rows
  df_x_train = pd.DataFrame(x_train)

  # Add the 'time' column to the DataFrame
  df_x_train['hour'] = time_values

  #print("len x_train", len(x_raw))
  return df_x_train
```

B3 Code for State Regression Prediction Model

Importing Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
```

```
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import cross_val_score
from sklearn.metrics import make_scorer
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
from sklearn.preprocessing import PolynomialFeatures
import joblib
```

Reading the dataset and lens.csv from B1

```
file_path = 'cyprus/'
X = pd.read_csv('GreenhouseClimate_X_adjusted.csv')
#terminal_state_indicator = pd.read_csv('/terminal_state_indicator.csv')
lens = pd.read_csv('lens.csv') # length of each team observations
#pd.set_option('display.max_rows', None)

merged_df = pd.concat([X], axis=1)
```

Plot correlation map of realized and set points

```
# Lists of columns for sp and vip
sp_columns = [col for col in merged_df.columns if '_sp' in col]
vip_columns = [col for col in merged_df.columns if '_vip' in col]

# Initialize an empty DataFrame to store the correlation values
correlation_matrix = pd.DataFrame(index=sp_columns, columns=vip_columns)

# Compute the correlation between each 'sp' column and each 'vip' column
for sp_col in sp_columns:
    for vip_col in vip_columns:
        correlation = merged_df[sp_col].corr(merged_df[vip_col])
        correlation_matrix.loc[sp_col, vip_col] = correlation

# Convert the correlation matrix to a float type for plotting
correlation_matrix = correlation_matrix.astype(float)

# Plot the heatmap
plt.figure(figsize=(len(vip_columns), len(sp_columns)))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm',
cbar_kws={'label': 'Correlation Coefficient'})
plt.title("Correlation between SP and VIP Variables")
plt.ylabel('SP Variables')
plt.xlabel('VIP Variables')
plt.tight_layout()  # Adjusts plot to ensure everything fits without
overlapping
plt.show()
```

Declaration of action and state columns

```
action_columns = ['VentLee', 'Ventwind', 'AssimLight', 'EnScr',
'BlackScr',
                  'co2_dos', 'Cum_irr', 'co2_sp', 'dx_sp',
                  't_grow_min_sp', 'assim_sp',
'scr_enrg_sp','scr_blck_sp', 't_heat_sp',
```

```
                  't_vent_sp', 'window_pos_lee_sp', 'int_blue_sp',
'int_red_sp',
                  'int_farred_sp', 'int_white_sp',
'water_sup_intervals_sp_min', 'water_sup']

state_columns = ['Tair', 'Rhair', 'HumDef', 'CO2air', 'PipeLow',
'PipeGrow',
                 'EC_drain_PC', 'Tot_PAR','Tot_PAR_Lamps', 'pH_drain_PC',
'hour',
                 't_grow_min_vip']


columns_to_keep = state_columns + action_columns

data_df = merged_df[columns_to_keep]
```

Creating expected outputs for each record and normalizing the dataset

```
lens_df = lens["length"]
end_indices = (lens_df - 1).astype(int).tolist()
print(end_indices)

# Initialize empty lists to store features and targets for all sequences
features_list = []
targets_list = []

# Iterate through each sequence, identified by start and end indices
start_idx = 0
for i, end_idx in enumerate(end_indices):
    # Select the current sequence excluding the last row
    current_sequence = data_df.iloc[start_idx:end_idx]
    current_targets = current_sequence[state_columns].shift(-1)

    # Drop the last row to avoid using it for predicting the start of the
next sequence
    current_sequence = current_sequence.iloc[:-1, :]
    current_targets = current_targets.iloc[:-1, :]

    #if i != i_test_sequence:
        # Store the processed sequence and targets
    features_list.append(current_sequence)
    targets_list.append(current_targets)
    #else:
        #X_test = current_sequence
        #Y_test = current_targets

    # Update start index for the next sequence
    start_idx = end_idx + 1

# Concatenate all processed sequences and targets
all_features = pd.concat(features_list, ignore_index=True)
all_targets = pd.concat(targets_list, ignore_index=True)

# Initialize scalers for features and targets
scaler_features = MinMaxScaler(feature_range=(0, 1))
scaler_targets = MinMaxScaler(feature_range=(0, 1))
```

```
# Scale the training features and targets
all_features_scaled = scaler_features.fit_transform(all_features)
all_targets_scaled = scaler_targets.fit_transform(all_targets)   #
Assuming Y_train is a Series
```

Defining the model types

```
# Define your models
models = {
    'Random Forest Regressor': RandomForestRegressor(n_estimators=100,
random_state=42),
    'Gradient Boosting':
MultiOutputRegressor(GradientBoostingRegressor(n_estimators=100,
learning_rate=0.05, random_state=42)),
    'Support Vector Regressor': MultiOutputRegressor(SVR(kernel='rbf')),
    'K-Nearest Neighbors': KNeighborsRegressor(n_neighbors=5),
    'Linear Regression': LinearRegression(),
    'Elastic Net': ElasticNet(random_state=42)
}
```

Training, saving and evaluating models with the train_test_split function

```
# Split the normalized data into training and testing sets
# X_train, X_test, Y_train, Y_test =
train_test_split(all_features_scaled, all_targets_scaled, test_size=0.2,
random_state=42) # false because the data is sequential
X_train, X_test, Y_train, Y_test = train_test_split(all_features_scaled,
all_targets_scaled, test_size=0.2, random_state=42) # false because the
data is sequential

predictions = []

for model_name, model in models.items():
  model.fit(X_train, Y_train)
  print(model_name + " trained\n" )
   # Save the model
  filename = file_path + f'models/{model_name.replace(" ", "_")}.joblib'
  joblib.dump(model, filename)
  predictions.append(model.predict(X_test))

# Function to calculate metrics
def calculate_metrics(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)
    return mae, mse, rmse, r2

# Calculate metrics for each model
metrics_rfr = calculate_metrics(Y_test, predictions[0])
metrics_gbr = calculate_metrics(Y_test, predictions[1])
metrics_svr = calculate_metrics(Y_test, predictions[2])
metrics_knn = calculate_metrics(Y_test, predictions[3])
metrics_lr = calculate_metrics(Y_test, predictions[4])
metrics_en = calculate_metrics(Y_test, predictions[5])

# Create a DataFrame to hold the metrics
metrics_df = pd.DataFrame({
```

```
    'Model': ['Random Forest Regressor', 'Gradient Boosting', 'Support Vector Regressor', 'K-Nearest Neighbors', 'Linear Regression', 'Elastic Net'],
    'MAE': [metrics_rfr[0], metrics_gbr[0], metrics_svr[0], metrics_knn[0], metrics_lr[0], metrics_en[0]],
    'MSE': [metrics_rfr[1], metrics_gbr[1], metrics_svr[1], metrics_knn[1], metrics_lr[1], metrics_en[1]],
    'RMSE': [metrics_rfr[2], metrics_gbr[2], metrics_svr[2], metrics_knn[2], metrics_lr[2], metrics_en[2]],
    'R2': [metrics_rfr[3], metrics_gbr[3], metrics_svr[3], metrics_knn[3], metrics_lr[3], metrics_en[3]]
})

metrics_df

# Set up the matplotlib figure and axes, one for each metric
fig, axes = plt.subplots(1, 4, figsize=(20, 5), sharey=True)
metrics = metrics_df.columns[1:]  # Skip the 'Model' column

for ax, metric in zip(axes, metrics):
    ax.bar(metrics_df['Model'], metrics_df[metric], color=plt.cm.Set3(np.arange(len(metrics_df))))
    ax.set_title(metric)
    ax.set_xticklabels(metrics_df['Model'], rotation=45, ha='right')

plt.tight_layout()
plt.show()
```

Training and evaluating models with CV 10 folds and saving the GBR ones

```
from sklearn.model_selection import cross_validate
# Define scorers
scorers = {
    'MAE': make_scorer(mean_absolute_error),
    'MSE': make_scorer(mean_squared_error),
    'R2': make_scorer(r2_score)
}

# Results container
results = []

# Iterate over each model
for model_name, model in models.items():

    model_results = {'Model': model_name}

    # Skip the two models that performed worst for faster training
#   if model_name == 'Elastic Net' or model_name == 'K-Nearest Neighbors' or model_name == 'Random Forest Regressor' or model_name == 'Support Vector Regressor' or model_name == 'Linear Regression':
    if model_name == '':
      # Skip cross-validation and set scores to 0 for all scorers
      for scorer_name in scorers.keys():
          model_results[scorer_name] = 0
    else:
       # Perform 10-fold cross-validation and record mean scores
```

```
      cv_results = cross_validate(model, all_features_scaled,
all_targets_scaled, cv=10, scoring=scorers, return_estimator=True)

      # Save each fold's estimator if the model is Gradient Boosting
      if model_name == 'Gradient Boosting':
          for i, estimator in enumerate(cv_results['estimator']):
              model_path = file_path + f'/gb_cv/{model_name.replace(" ",
"_")}_fold_{i+1}.joblib'
              joblib.dump(estimator, model_path)

      # Record the mean scores
      for scorer_name in scorers.keys():
          model_results[scorer_name] = cv_results['test_' +
scorer_name].mean()

      results.append(model_results)

# Convert results to DataFrame for easy viewing
results_df = pd.DataFrame(results)

# Print the results
print(results_df)
```

B4 Code for Crop Parameters Regression Prediction Model

Import Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import cross_val_score
from sklearn.metrics import make_scorer
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
from sklearn.preprocessing import PolynomialFeatures
import joblib
```

Reading the dataset and lens.csv from B1

```
file_path = 'cyprus/'
X = pd.read_csv(file_path + 'RewardDataset_X.csv')
lens = pd.read_csv(file_path + 'lens.csv') # length of each team
observations
merged_df = pd.concat([X], axis=1)
```

Keep columns needed for Crop Parameters' prediction

```
parameters = ['Stem_elong', 'Stem_thick', 'Cum_trusses']
not_used = ['assim_vip', 'co2_vip', 'dx_vip', 'int_blue_vip',
'int_farred_vip', 'int_red_vip', 'int_white_vip', 'scr_blck_vip',
'scr_enrg_vip',
        't_heat_vip', 't_rail_min_sp', 't_rail_min_vip', 't_ventlee_vip',
't_ventwind_vip', 'water_sup_intervals_vip_min', 'window_pos_lee_vip']
other_features = [col for col in merged_df.columns if col not in
parameters + not_used]
```

Plot correlation map of features with the 3 crop parameters

```
# Create a new DataFrame with only the parameters and other features
data_for_correlation = merged_df[parameters + other_features]

# Calculate the correlation matrix
correlation_matrix = data_for_correlation.corr().loc[parameters,
other_features]

# Create a larger figure to improve readability
plt.figure(figsize=(20, 10))  # You can adjust these values as needed

# Create the heatmap using seaborn, with better text size for readability
ax = sns.heatmap(correlation_matrix, annot=True, fmt=".2f",
cmap='coolwarm', cbar_kws={'label': 'Correlation Coefficient'},
                 annot_kws={"size": 8})

# Set larger x-tick and y-tick labels for better readability
plt.xticks(rotation=90, fontsize=10)  # Rotate and set a smaller size if
necessary
plt.yticks(fontsize=12)  # Set a larger size for y-ticks for better
readability

# Set the title and labels with larger font sizes for better readability
plt.title('Correlation between Parameters and Other Features',
fontsize=14)
plt.xlabel('Other Features', fontsize=12)
plt.ylabel('Selected Parameters', fontsize=12)

# Ensure everything fits without overlapping
plt.tight_layout()

# Display the heatmap
plt.show()
```

Creating expected outputs for each record and normalizing the dataset

```
data_df = merged_df[other_features + parameters]
lens_df = lens["length"]
end_indices = (lens_df - 1).astype(int).tolist()
crop_parameters = ['Stem_elong', 'Stem_thick', 'Cum_trusses']
# Initialize empty lists to store features and targets for all sequences
features_list = []
targets_list = []

# Iterate through each sequence, identified by start and end indices
start_idx = 0
```

```
for i, end_idx in enumerate(end_indices):
    # Select the current sequence excluding the last row
    current_sequence = data_df.iloc[start_idx:end_idx]
    current_targets = current_sequence[crop_parameters].shift(-1)

    # Drop the last row to avoid using it for predicting the start of the
next sequence
    current_sequence = current_sequence.iloc[:-1, :]
    current_targets = current_targets.iloc[:-1, :]

    #if i != i_test_sequence:
        # Store the processed sequence and targets
    features_list.append(current_sequence)
    targets_list.append(current_targets)
    #else:
        #X_test = current_sequence
        #Y_test = current_targets

    # Update start index for the next sequence
    start_idx = end_idx + 1

# Concatenate all processed sequences and targets
all_features = pd.concat(features_list, ignore_index=True)
all_targets = pd.concat(targets_list, ignore_index=True)

# Initialize scalers for features and targets
scaler_features = MinMaxScaler(feature_range=(0, 1))
scaler_targets = MinMaxScaler(feature_range=(0, 1))


# Scale the training features and targets
all_features_scaled = scaler_features.fit_transform(all_features)
all_targets_scaled = scaler_targets.fit_transform(all_targets)  #
Assuming Y_train is a Series
```

Defining the model types

```
# Define your models
models = {
    'Random Forest Regressor': RandomForestRegressor(n_estimators=100,
random_state=42),
    'Gradient Boosting':
MultiOutputRegressor(GradientBoostingRegressor(n_estimators=100,
learning_rate=0.05, random_state=42)),
    'Support Vector Regressor': MultiOutputRegressor(SVR(kernel='rbf')),
    'K-Nearest Neighbors': KNeighborsRegressor(n_neighbors=5),
    'Linear Regression': LinearRegression(),
    'Elastic Net': ElasticNet(random_state=42)
}
```

Training, saving and evaluating models with the train_test_split function

```
# Split the normalized data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(all_features_scaled,
all_targets_scaled, test_size=0.2, random_state=42) # false because the
data is sequential

predictions = []
```

```
for model_name, model in models.items():
  model.fit(X_train, Y_train)
  print(model_name + " trained\n" )
  filename = file_path + f'/r_models/{model_name.replace(" ",
"_")}.joblib'
  joblib.dump(model, filename)
  predictions.append(model.predict(X_test))

# Function to calculate metrics
def calculate_metrics(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)
    return mae, mse, rmse, r2

# Calculate metrics for each model
metrics_rfr = calculate_metrics(Y_test, predictions[0])
metrics_gbr = calculate_metrics(Y_test, predictions[1])
metrics_svr = calculate_metrics(Y_test, predictions[2])
metrics_knn = calculate_metrics(Y_test, predictions[3])
metrics_lr = calculate_metrics(Y_test, predictions[4])
metrics_en = calculate_metrics(Y_test, predictions[5])

# Create a DataFrame to hold the metrics
metrics_df = pd.DataFrame({
    'Model': ['Random Forest Regressor', 'Gradient Boosting', 'Support
Vector Regressor', 'K-Nearest Neighbors', 'Linear Regression', 'Elastic
Net'],
    'MAE': [metrics_rfr[0], metrics_gbr[0], metrics_svr[0],
metrics_knn[0], metrics_lr[0], metrics_en[0]],
    'MSE': [metrics_rfr[1], metrics_gbr[1], metrics_svr[1],
metrics_knn[1], metrics_lr[1], metrics_en[1]],
    'RMSE': [metrics_rfr[2], metrics_gbr[2], metrics_svr[2],
metrics_knn[2], metrics_lr[2], metrics_en[2]],
    'R2': [metrics_rfr[3], metrics_gbr[3], metrics_svr[3],
metrics_knn[3], metrics_lr[3], metrics_en[3]]
})

metrics_df

# Set up the matplotlib figure and axes, one for each metric
fig, axes = plt.subplots(1, 4, figsize=(20, 5), sharey=True)
metrics = metrics_df.columns[1:]  # Skip the 'Model' column

for ax, metric in zip (axes, metrics):
    ax.bar(metrics_df['Model'], metrics_df[metric],
color=plt.cm.Set3(np.arange(len(metrics_df))))
    ax.set_title(metric)
    ax.set_xticklabels(metrics_df['Model'], rotation=45, ha='right')

plt.tight_layout()
plt.show()
```

Training and evaluating models with CV 10 folds and saving the GBR ones

```
from sklearn.model_selection import cross_validate
# Define scorers
scorers = {
    'MAE': make_scorer(mean_absolute_error),
    'MSE': make_scorer(mean_squared_error),
    'R2': make_scorer(r2_score)
}

# Results container
results = []

# Iterate over each model
for model_name, model in models.items():

    model_results = {'Model': model_name}

    # Skip the two models that performed worst for faster training
#   if model_name == 'Elastic Net' or model_name == 'K-Nearest Neighbors' 
or model_name == 'Random Forest Regressor' or model_name == 'Support 
Vector Regressor' or model_name == 'Linear Regression':
    if model_name == '':
      # Skip cross-validation and set scores to 0 for all scorers
      for scorer_name in scorers.keys():
          model_results[scorer_name] = 0
    else:
       # Perform 10-fold cross-validation and record mean scores
      cv_results = cross_validate(model, all_features_scaled, 
all_targets_scaled, cv=10, scoring=scorers, return_estimator=True)

      # Save each fold's estimator if the model is Gradient Boosting
      if model_name == 'Gradient Boosting':
          for i, estimator in enumerate(cv_results['estimator']):
              model_path = file_path + f'/r_gb_cv/{model_name.replace(" 
", "_")}_fold_{i+1}.joblib'
              joblib.dump(estimator, model_path)

      # Record the mean scores
      for scorer_name in scorers.keys():
          model_results[scorer_name] = cv_results['test_' + 
scorer_name].mean()

      results.append(model_results)

# Convert results to DataFrame for easy viewing
results_df = pd.DataFrame(results)

# Print the results
print(results_df)
```

B5 Code for Environment Simulator

Import Libraries

```
!pip install tensorflow==2.15.0
!pip install dm-reverb
!pip install tf-agents[reverb]
!pip install --upgrade tf-agents
!pip install tf-keras
!pip show tensorflow
import os
# Keep using keras-2 (tf-keras) rather than keras-3 (keras).
os.environ['TF_USE_LEGACY_KERAS'] = '1'
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import tensorflow as tf
import numpy as np
import pandas as pd
import gin
import tf_agents
from tf_agents.environments import py_environment
from tf_agents.environments import tf_environment
from tf_agents.environments import tf_py_environment
from tf_agents.trajectories import time_step as ts
from tf_agents.specs import BoundedArraySpec
import joblib
```

Environment Simulator class

```
class GreenhouseEnvironment(py_environment.PyEnvironment):

  def action_spec(self):
        return {
          'EnScr': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='EnScr'),
          'AssimLight': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='AssimLight'),
          'BlackScr': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='BlackScr'),
          'dx_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='dx_sp'),
          'co2_dos': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='co2_dos'),
          'Cum_irr': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='Cum_irr'),
          'VentLee': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='VentLee'),
          'Ventwind': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='Ventwind'),
          'co2_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='co2_sp'),
          't_grow_min_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='t_grow_min_sp'),
          'assim_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='assim_sp'),
          'scr_enrg_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='scr_enrg_sp'),
          'scr_blck_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='scr_blck_sp'),
```

```
          't_heat_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='t_heat_sp'),
          't_vent_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='t_vent_sp'),
          'window_pos_lee_sp': BoundedArraySpec(shape=(),
dtype=np.float32, minimum=0.0, maximum=1.0, name='window_pos_lee_sp'),
          'int_blue_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='int_blue_sp'),
          'int_red_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='int_red_sp'),
          'int_farred_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='int_farred_sp'),
          'int_white_sp': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='int_white_sp'),
          'water_sup_intervals_sp_min': BoundedArraySpec(shape=(),
dtype=np.float32, minimum=0.0, maximum=1.0,
name='water_sup_intervals_sp_min'),
          'water_sup': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='water_sup')
    }

  def observation_spec(self):
      return {
          'EC_drain_PC': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='EC_drain_PC'),
          'CO2air': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='CO2air'),
          'HumDef': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='HumDef'),
          'PipeGrow': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='PipeGrow'),
          'PipeLow': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='PipeLow'),
          'Rhair': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='Rhair'),
          'Tair': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='Tair'),
          'Tot_PAR': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='Tot_PAR'),
          'Tot_PAR_Lamps': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='Tot_PAR_Lamps'),
          'pH_drain_PC': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='pH_drain_PC'),
          't_grow_min_vip': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='t_grow_min_vip'),
          'hour': BoundedArraySpec(shape=(), dtype=np.float32,
minimum=0.0, maximum=1.0, name='hour'),
      }


  def __init__(self):

    self.column_state_order = ['Tair', 'Rhair', 'HumDef', 'CO2air',
'PipeLow', 'PipeGrow', 'EC_drain_PC', 'Tot_PAR', 'Tot_PAR_Lamps',
'pH_drain_PC', 'hour',
```

```
            't_grow_min_vip', 'VentLee', 'Ventwind', 'AssimLight',
'EnScr', 'BlackScr', 'co2_dos', 'Cum_irr', 'co2_sp', 'dx_sp',
't_grow_min_sp',
            'assim_sp', 'scr_enrg_sp', 'scr_blck_sp', 't_heat_sp',
't_vent_sp', 'window_pos_lee_sp', 'int_blue_sp', 'int_red_sp',
'int_farred_sp',
            'int_white_sp', 'water_sup_intervals_sp_min', 'water_sup']

    self.column_reward_order = ['AssimLight', 'BlackScr', 'CO2air',
'Cum_irr', 'EC_drain_PC', 'EnScr', 'HumDef', 'PipeGrow', 'PipeLow',
'Rhair', 'Tair', 'Tot_PAR',
       'Tot_PAR_Lamps', 'VentLee', 'Ventwind', 'assim_sp', 'co2_dos',
'co2_sp', 'dx_sp', 'int_blue_sp', 'int_farred_sp', 'int_red_sp',
'int_white_sp',
       'pH_drain_PC', 'scr_blck_sp', 'scr_enrg_sp', 't_grow_min_sp',
't_grow_min_vip', 't_heat_sp', 't_vent_sp', 'water_sup',
       'water_sup_intervals_sp_min', 'window_pos_lee_sp', 'hour',
'Stem_elong', 'Stem_thick', 'Cum_trusses']

    self._reset()

    self.episode_ended = False
    self.weights = {'diff_Stem_elong': 0.2, 'diff_Stem_thick': 0.2,
'diff_Cum_trusses': 0.6}
    self.timestep = 0
    self.file_path = 'cyprus/'
    self._state_model = joblib.load(self.file_path +
'models/Gradient_Boosting.joblib')
    self._state_cv_models = []
    self.cumulative_reward = 0
     # Load and predict with each model
    for fold in range(1, 11):
        model_path =
f'{self.file_path}models/Gradient_Boosting_fold_{fold}.joblib'
        self._state_cv_models.append(joblib.load(self.file_path +
f'gb_cv/Gradient_Boosting_fold_{fold}.joblib'))
    self._reward_model = joblib.load(self.file_path +
'r_models/Gradient_Boosting.joblib')
    self._reward_cv_models = []
    self.final_rewards = []
    for fold in range(1, 11):
        model_path =
f'{self.file_path}models/Gradient_Boosting_fold_{fold}.joblib'
        self._reward_cv_models.append(joblib.load(self.file_path +
f'r_gb_cv/Gradient_Boosting_fold_{fold}.joblib'))


  def _reset(self):
      # Define the initial state of the environment
      self._state = {
          'EC_drain_PC': np.float32(np.random.uniform(0.0, 0.1)),
          'CO2air': np.float32(np.random.uniform(0.0, 0.1)),
          'HumDef': np.float32(np.random.uniform(0.0, 0.1)),
          'PipeGrow': np.float32(np.random.uniform(0.0, 0.1)),
          'PipeLow': np.float32(np.random.uniform(0.0, 0.1)),
          'Rhair': np.float32(np.random.uniform(0.0, 0.1)),
```

```
            'Tair': np.float32(np.random.uniform(0.0, 0.1)),
            'Tot_PAR': np.float32(np.random.uniform(0.0, 0.1)),
            'Tot_PAR_Lamps': np.float32(np.random.uniform(0.0, 0.1)),
            'pH_drain_PC': np.float32(np.random.uniform(0.0, 0.1)),
            't_grow_min_vip': np.float32(np.random.uniform(0.0, 0.1)),
            'hour': np.float32(np.random.randint(0, 24)/24)  # 24 is
exclusive,
        }
        self._parameters = {
            'Stem_elong': np.float32(np.random.uniform(0.0, 0.1)),
            'Stem_thick': np.float32(np.random.uniform(0.0, 0.1)),
            'Cum_trusses': np.float32(np.random.uniform(0.0, 0.1)),
            'diff_Stem_elong': np.float32(0),   # Will be between -1 and 1
            'diff_Stem_thick': np.float32(0),
            'diff_Cum_trusses': np.float32(0)
            # Add other necessary parameters similarly
        }

        self.episode_ended = False
        self.timestep = 0
        self.cumulative_reward = 0
        # Directly use the state dictionary as the observation for the
TimeStep
        return ts.restart(self._state)

  def _calculate_reward(self):
    current_parameters = self._parameters
    weighted_df = pd.DataFrame()
    input_df = pd.DataFrame([{
    'diff_Stem_elong': current_parameters['diff_Stem_elong'],
    'diff_Stem_thick': current_parameters['diff_Stem_thick'],
    'diff_Cum_trusses': current_parameters['diff_Cum_trusses']
    }])
    for column, weight in self.weights.items():
      weighted_df[column] = input_df[column] * weight
    weighted_sum = weighted_df.sum(axis=1)
    return self.ensure_scalar(weighted_sum[0])

  def _action_spec(self):
    return self._action_spec

  def _observation_spec(self):
    return self._observation_spec

  def _parameters_spec(self):
    return self._action_spec

  def ensure_scalar(self, value):
        """Ensure numpy arrays with one element are converted to scalars
and cast to np.float32."""
        if isinstance(value, np.ndarray) and value.size == 1:
            # Convert single-element arrays to scalar and ensure type
np.float32
            return np.float32(value.item())
        elif isinstance(value, float):
            # Convert Python float to np.float32
```

```
            return np.float32(value)
        return value



    def _predict_state(self, input_data):
        # Assuming input_data is a DataFrame formatted correctly for your
models
        # The DataFrame should be ordered and formatted correctly for model
input
        predictions = []

        # Loop through each of the models, make predictions, and store them
        for model in self._state_cv_models:
            pred = model.predict(input_data.values)
            predictions.append(pred)

        # Convert predictions to a numpy array for easier averaging
        predictions = np.array(predictions)

        # Average predictions across all models for each parameter
        # Assuming the output is 1D per model
        average_predictions = predictions.mean(axis=0)

        # Return the average predictions
        return average_predictions

    def _predict_parameters(self, input_data):
        # Assuming input_data is a DataFrame formatted correctly for your
models
        # The DataFrame should be ordered and formatted correctly for model
input
        predictions = []

        # Loop through each of the models, make predictions, and store them
        for model in self._reward_cv_models:
            pred = model.predict(input_data.values)
            predictions.append(pred)

        # Convert predictions to a numpy array for easier averaging
        predictions = np.array(predictions)

        # Average predictions across all models for each parameter
        # Assuming the output is 1D per model
        average_predictions = predictions.mean(axis=0)

        # Return the average predictions
        return average_predictions

    def _update_parameters(self, action):
         # Create DataFrame from state and action dictionaries for model
input
        combined_data = {**self._state, **action, **self._parameters}
        input_df = pd.DataFrame([combined_data])

        # Order columns as expected by the model
```

```
    input_df = input_df[self.column_reward_order]

    # Predict the next state using the model USING NON CV MODEL
    # predicted_reward_array =
self._reward_model.predict(input_df.values)
    # Predict the next state using the model USINGCV MODEL
    predicted_reward_array = self._predict_parameters(input_df)

     # Convert the predicted array to a DataFrame if necessary, assuming
the output is 1D
    if isinstance(predicted_reward_array, np.ndarray) and
predicted_reward_array.ndim == 1:
        predicted_reward_df =
pd.DataFrame(predicted_reward_array.reshape(1, -1),
columns=['Stem_elong', 'Stem_thick', 'Cum_trusses'])
    else:
        predicted_reward_df = pd.DataFrame(predicted_reward_array,
columns=['Stem_elong', 'Stem_thick', 'Cum_trusses'])

    # Convert the predicted DataFrame back to a dictionary
    predicted_reward_dict = predicted_reward_df.iloc[0].to_dict()

    temp = predicted_reward_dict.copy()

    # Clip values to ensure they are within the specified range
    for key, value in predicted_reward_dict.items():
        clipped_value = np.clip(value, 0.0, 2.0)  # Ensuring the value
stays between 0 and 2 (2 means double the best in data)
        predicted_reward_dict[key] = np.float32(clipped_value)  # Convert
and store as float32

    # Calculate the difference for each parameter between new and old
values
    for key in temp:
        if key in self._parameters:
            old_value = self._parameters.get(key, 0)
            new_value = predicted_reward_dict[key]
            diff_key = f'diff_{key}'
            predicted_reward_dict[diff_key] = new_value - old_value



    #print("Old reward parameters:",self._parameters)
    #print("Predicted reward parameters:",predicted_reward_dict)

    # Now, update the parameters in the state with this new dictionary
    self._parameters.update(predicted_reward_dict)

    for key in self._parameters:
        self._parameters[key] = self.ensure_scalar(self._parameters[key])

    #print("Updated reward parameters:",self._parameters)

  def _update_state(self, action):
    # Create DataFrame from state and action dictionaries for model input
    combined_data = {**self._state, **action}
```

```
    input_df = pd.DataFrame([combined_data])

    # Order columns as expected by the model
    input_df = input_df[self.column_state_order]

    # Predict the next state
    predicted_state_array = self._predict_state(input_df)

    # Convert the predicted array to a DataFrame if necessary, assuming
the output is 1D
    if isinstance(predicted_state_array, np.ndarray) and
predicted_state_array.ndim == 1:
        predicted_state_df =
pd.DataFrame(predicted_state_array.reshape(1, -1), columns=['Tair',
'Rhair', 'HumDef', 'CO2air', 'PipeLow', 'PipeGrow',
                 'EC_drain_PC', 'Tot_PAR','Tot_PAR_Lamps', 'pH_drain_PC',
'hour', 't_grow_min_vip'])
    else:
        predicted_state_df = pd.DataFrame(predicted_state_array,
columns=['Tair', 'Rhair', 'HumDef', 'CO2air', 'PipeLow', 'PipeGrow',
                 'EC_drain_PC', 'Tot_PAR','Tot_PAR_Lamps', 'pH_drain_PC',
'hour', 't_grow_min_vip'])

    # Increment 'hour' by 1/24 to simulate the passing of one hour
    new_hour = self._parameters.get('hour', 0) + 1/24

    # If 'hour' exceeds 1, wrap it around to 0
    if new_hour >= 1:
        new_hour = 0


    predicted_state_dict = predicted_state_df.iloc[0].to_dict()

    # Clip values to ensure they are within the specified range
    for key, value in predicted_state_dict.items():
        clipped_value = np.clip(value, 0.0, 1.0)  # Ensuring the value
stays between 0 and 1
        predicted_state_dict[key] = np.float32(clipped_value)  # Convert
and store as float32

    #print("Old state parameters:", self._state)
    #print("Predicted state parameters:", predicted_state_dict)

    # Convert the predicted DataFrame back to a dictionary and update the
state
    self._state.update(predicted_state_dict)

    # Update 'hour' in the state
    self._state['hour'] = new_hour

    for key in self._state:
        self._state[key] = self.ensure_scalar(self._state[key])

    #print("Updated state parameters:", self._state)

  def _check_termination_condition(self):
```

```
    # Assuming these are your termination conditions
    max_timesteps = 4000  # Or another appropriate number for your
environment
    return self.timestep >= max_timesteps


  def _step(self, action):
    #print(action)
    # Assume action is a dict with keys corresponding to action_spec
    # Apply the action to the environment
    self._update_parameters(action)

    # Update the environment's state based on the action
    self._update_state(action)

    # Increment timestep
    self.timestep += 1

    # Check if the episode has ended, which could be based on a condition
or timestep
    if self._check_termination_condition():
        self._episode_ended = True
    else:
        self._episode_ended = False

    # Format observations to ensure they are scalars
    observation = {key: self.ensure_scalar(self._state[key]) for key in
self._state}
    reward = self.ensure_scalar(self._calculate_reward())
    self.cumulative_reward += reward  # Accumulate the reward
    discount = 1

    # Construct the TimeStep based on the episode end status
    if self._episode_ended:
        self.timestep = 0
        self.final_rewards.append(self.cumulative_reward)  # Store the
total reward for this episode
        print(self.cumulative_reward)
        self.cumulative_reward = 0  # Reset cumulative reward for the
next episode
        return ts.termination(self._state, reward)
    else:
        return ts.transition(self._state, reward, discount=1)
```

B6 Code for PPO agent

Import Libraries

```
import tensorflow as tf
from tf_agents.environments import tf_py_environment
from tf_agents.networks import actor_distribution_network, value_network
from tf_agents.agents.ppo import ppo_agent
```

```
from tf_agents.utils import common
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.drivers import dynamic_episode_driver
from tf_agents.policies import policy_saver
from tensorflow.keras.layers import Concatenate
import matplotlib.pyplot as plt
```

Custom Combiner

```
# Define a simple combiner that concatenates all input tensors along the
last dimension
class CustomCombiner(tf.keras.layers.Layer):
    def call(self, inputs):
        # Flatten each input and concatenate along the last axis
        flattened_inputs = [tf.reshape(input_tensor,
(tf.shape(input_tensor)[0], -1)) for input_tensor in inputs.values()]
        return tf.concat(flattened_inputs, axis=-1)

# Use the custom combiner in the EncodingNetwork
combiner = CustomCombiner()
```

Initialize training and evaluation environment

```
# Initialize the environment
environment = GreenhouseEnvironment()  # Assuming this is your custom
environment
tf_env = tf_py_environment.TFPyEnvironment(environment)
eval_env = tf_py_environment.TFPyEnvironment(GreenhouseEnvironment())
# Verify environment compatibility with TF-Agents
utils.validate_py_environment(train_py_env, episodes=1)
```

Implementing agent

```
# Create a concatenation layer for combining inputs
concat_layer = Concatenate(axis=-1)

# Create actor and value networks for the PPO agent
actor_net = actor_distribution_network.ActorDistributionNetwork(
    train_env.observation_spec(),
    train_env.action_spec(),
    preprocessing_combiner=combiner,
    fc_layer_params=(100,))

value_net = value_network.ValueNetwork(
    train_env.observation_spec(),
    preprocessing_combiner=combiner,
    fc_layer_params=(100,))

# Create the optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)

# Initialize the PPO agent
ppo_tf_agent = ppo_agent.PPOAgent(
    train_env.time_step_spec(),
    train_env.action_spec(),
    actor_net=actor_net,
    value_net=value_net,
    optimizer=optimizer,
```

```
    normalize_observations=True,
    normalize_rewards=True,
    use_gae=True,
    num_epochs=10)

ppo_tf_agent.initialize()

# Replay buffer to store the collected data
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=ppo_tf_agent.collect_data_spec,
    batch_size=train_env.batch_size,
    max_length=1000)

# Collect driver for gathering episodes
collect_driver = dynamic_episode_driver.DynamicEpisodeDriver(
    train_env,
    ppo_tf_agent.collect_policy,
    observers=[replay_buffer.add_batch],
    num_episodes=1)
```

Training Loop

```
# Training Loop
num_episodes = 100
for _ in range(num_episodes):
    # Collect and train
    collect_driver.run()
    trajectories = replay_buffer.gather_all()
    train_loss = agent.train(experience=trajectories)
    replay_buffer.clear()
    print(f'Training Loss: {train_loss.loss.numpy()}')
```

Plotting cumulative rewards of training

```
# Assuming the environment instance is `env`
episode_rewards = environment.final_rewards  # Access the cumulative
rewards after the training loop
# Plot cumulative rewards
plt.figure(figsize=(10, 5))
plt.plot(episode_rewards, label='Cumulative Reward')
plt.xlabel('Iterations')
plt.ylabel('Total Reward')
plt.title('Cumulative Reward per Episode Over Training')
plt.legend()
plt.grid(True)
plt.show()
```

Evaluation of policy

```
# Evaluate the agent's policy once training has completed
num_eval_episodes = 10  # Number of episodes to run for evaluation
episode_rewards = []
for _ in range(num_eval_episodes):
    time_step = eval_env.reset()
    episode_reward = 0
    while not time_step.is_last():
        action_step = ppo_tf_agent.policy.action(time_step)
        time_step = eval_env.step(action_step.action)
        episode_reward += time_step.reward.numpy()
```

```
    print(f'Episode Reward: {episode_reward}')
    episode_rewards.append(episode_reward)
```

Plotting cumulative rewards from evaluation

```
# Plotting the results
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
plt.plot(episode_rewards, label='Cumulative Reward per Episode')
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
plt.title('Cumulative Reward Per Episode Over Time for REINFORCE')
plt.legend()
plt.grid(True)
plt.show()
```

B7 Code for REINFORCE agent

Import Libraries

```
import tensorflow as tf
from tf_agents.environments import tf_py_environment
from tf_agents.networks import actor_distribution_network
from tf_agents.agents.reinforce import reinforce_agent
from tf_agents.drivers import dynamic_episode_driver
from tf_agents.replay_buffers import TFUniformReplayBuffer
from tf_agents.trajectories import trajectory
```

Custom Combiner

```
# Custom Combiner for the actor network
class CustomCombiner(tf.keras.layers.Layer):
    def call(self, inputs):
        return tf.concat([tf.reshape(tensor, [tf.shape(tensor)[0], -1])
for tensor in inputs.values()], axis=-1)
```

Initialize training and evaluation environment

```
# Initialize the environment
environment = GreenhouseEnvironment()  # Assuming this is your custom
environment
tf_env = tf_py_environment.TFPyEnvironment(environment)
eval_env = tf_py_environment.TFPyEnvironment(GreenhouseEnvironment())
```

Implementing agent

```
# Actor Network
actor_net = actor_distribution_network.ActorDistributionNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_combiner=CustomCombiner(),
    fc_layer_params=(100, 50),
)

# REINFORCE Agent
```

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
train_step_counter = tf.Variable(0)

agent = reinforce_agent.ReinforceAgent(
    tf_env.time_step_spec(),
    tf_env.action_spec(),
    actor_network=actor_net,
    optimizer=optimizer,
    normalize_returns=True,
    train_step_counter=train_step_counter
)

agent.initialize()

# Replay Buffer
replay_buffer = TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000
)

# Collect Driver
collect_driver = dynamic_episode_driver.DynamicEpisodeDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer.add_batch],
    num_episodes=1
)
```

Training Loop

```
# Training Loop
num_episodes = 100
for _ in range(num_episodes):
    # Collect and train
    collect_driver.run()
    trajectories = replay_buffer.gather_all()
    train_loss = agent.train(experience=trajectories)
    replay_buffer.clear()
    print(f'Training Loss: {train_loss.loss.numpy()}')
```

Plotting cumulative rewards of training

```
# Assuming the environment instance is `env`
episode_rewards = environment.final_rewards  # Access the cumulative
rewards after the training loop
# Plot cumulative rewards
plt.figure(figsize=(10, 5))
plt.plot(episode_rewards, label='Cumulative Reward')
plt.xlabel('Iterations')
plt.ylabel('Total Reward')
plt.title('Cumulative Reward per Episode Over Training')
plt.legend()
plt.grid(True)
plt.show()
```

Evaluation of policy

```
eval_env = tf_py_environment.TFPyEnvironment(GreenhouseEnvironment())
```

```python
# Evaluate the agent's policy once training has completed
num_eval_episodes = 10  # Number of episodes to run for evaluation
episode_rewards = []

for _ in range(num_eval_episodes):
    time_step = eval_env.reset()
    episode_reward = 0
    while not time_step.is_last():
        action_step = agent.policy.action(time_step)
        time_step = eval_env.step(action_step.action)
        episode_reward += time_step.reward.numpy()
    print(f'Episode Reward: {episode_reward}')
    episode_rewards.append(episode_reward)
```

Plotting cumulative rewards from evaluation

```python
# Plotting the results
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
plt.plot(episode_rewards, label='Cumulative Reward per Episode')
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
plt.title('Cumulative Reward Per Episode Over Time for REINFORCE')
plt.legend()
plt.grid(True)
plt.show()
```