Master's thesis

2021

Jakob Løver

Master's thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Jakob Løver

Explaining a Deep Reinforcement Learning Agent Using Regression Trees

July 2021

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Explaining a Deep Reinforcement Learning Agent Using Regression Trees

Jakob Løver

Cybernetics and Robotics
Submission date: July 2021
Supervisor: Anastasios Lekkas
Co-supervisor: Vilde Gjærum

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

The adoption of black-box machine learning systems in control systems poses a problem for applications where safety is of critical importance. Deep reinforcement learning systems are often built using black-box models such as neural networks. The accuracy of these systems usually come at the cost of interpretability, meaning how understandable their decisions are. Explaining decisions made by a black-box model without knowing more about the internal workings of the model poses several issues. Instead, an training interpretable model to approximate the black-box model can alleviate these issues by giving domain experts a more holistic understanding of why certain decisions were made.

Docking ships to harbor is a difficult control problem. In the past, different strategies have been employed for automatic docking, such as more traditional controllers and supervised machine learning, with various success. Several issues arise when one uses for example supervised learning to solve the docking problem. For example, to be able to generalize well from harbor to harbor, great care must be taken to collect good data, which on its own is a difficult problem. This can be remedied by using Deep Reinforcement Learning (DRL). Instead of explicitly teaching the agent how to dock the vessel, the agent may learn itself through simulations how to dock. The deep reinforcement learning agent however is powered by a black-box neural network trained through Proximal Policy Optimization (PPO). It is therefore of interest to investigate methods that may aid in explaining what the neural network has taught itself. In this master thesis, we will investigate alternative models that approximate the neural network powering the DRL agent through imitation learning. This approximation can lead to new insights into what the neural network learned through self-learning, and provide engineers with another tool in their toolkit to ensure the agent will behave as expected.

This master thesis demonstrates that recent developments in decision tree methods are able to sufficiently approximate the behavior of a deep reinforcement learning agent trained to dock a vessel to harbor. The state-of-the-art methods Optimal Regression Trees with linear predictions, and Near-optimal Nonlinear Regression Trees are trained through imitation learning by growing trees using data from the docking agent to create more interpretable models. The optimal regression tree method was chosen because of the similar structure to decision tree methods such as Classification and Regression Trees, and Linear Model Trees, without the disadvantage of being built through greedy algorithms leading to sub-optimal solutions. Near-optimal Nonlinear Regression Trees was chosen because of the supposed further improvement over the optimal regression trees and its non-linear prediction functions, possibly enabling shallower trees than all the aforementioned methods.

The resulting trees are evaluated on new unseen data to compare their performance to that of the original black-box agent trained through PPO, and the previously implemented method Linear Model Trees. It is shown that Optimal Regression Trees were able to function as a replacement for the PPO agent with a lower failure rate than the linear model trees. Through common metrics for regression evaluation, it is shown that Optimal Regression Trees approximated the PPO agent better than the Linear Model Tree, and are able to approximate all states of the docking agent by trading some accuracy for vastly improved interpretability. We provide evidence that Optimal Regression Trees are shallower and more accurate than the Linear Model Trees at the cost of needing one tree per action.

Sammendrag

Bruken av "svart-boks"-modeller innen maskinlæring skaper problemer for systemer med fokus på sikkerhet. Systemer som nyttegjør seg av dyp forsterkende læring (engelsk: deep reinforcement learning / DRL) designes ofte med svart-boks-modeller som for eksempel nevrale nettverk. Nøyaktigheten til disse systemene kommer ofte på bekostning av hvor forståelig valgene til det nevrale nettverket er. Flere problemer oppstår hvis man prøver å forklare en svart-boks-modell uten å vite mer om den innvendige strukturen til modellen. Istedet kan en mer forståelig modell brukes til å approksimere svart-boks-modellen for å gi domene-eksperter et mer helhetlig innsyn i hvorfor svart-boks-modellene oppfører seg som de gjør.

Å legge båter til kai er et vanskelig reguleringsproblem. Tidligere har flere forskjellige metoder blitt forsøkt brukt for å automatisere denne prosessen med varierende suksess, slik som mer tradisjonelle regulatorer og overvåket maskinlæring. Problemer oppstår ved bruk av for eksempel overvåket masinlæring til å løse reguleringsproblemet. For at modellen man trener opp skal kunne generalisere bra fra kai til kai må man være nøye med hva slags data man oppsamler, som er et vanskelig problem i seg selv. Dette kan løses ved å trene opp en agent ved hjelp av DRL. Istedet for å eksplisitt trene opp en agent til å legge båten til kai kan agenten lære seg selv hvordan dette kan gjøres. Agenten nyttegjør seg av et svart-boks nevralt nettverk trent opp ved hjelp av PPO (engelsk: Proximal Policy Optimization). Derfor er det interessant å undersøke metoder som kan hjelpe å forklare hva det nevrale nettverket har lært seg. I denne masteroppgaven undersøker vi alternativer modeller som kan approksimere det nevrale nettverket til agenten gjennom å herme dens oppførsel (engelsk: imitation learning). Denne approksimasjonen kan føre til ny lærdom om hva det nevrale nettverket lærte seg selv gjennom DRL, og kan gi ingeniører enda et verktøy i verktøykassa for å sikre at agenten oppfører seg som forventet.

I denne masteroppgaven beviser vi at nye utviklinger innen beslutningstrær kan approksimere en DRL-agent trent til å legge båt til kai med tilfredstillende høyere nøyaktighet enn tidligere metoder. De nye metodene "Optimale Regresjonstrær" (engelsk: Optimal Regression Trees / ORT) med lineære regresjoner i løvnodene, og "Nær Optimale Regresjonstrær" (engelsk: Near-optimal Nonlinear Regression

Trees / NNRT) er trent opp gjennom å herme DRL agenten med et mål om å øke innsikten i agenten. Metoden ORT ble valgt fordi strukturen er svært lik andre regresjonstrær som "Klassifisering- og regresjonstrær" (engelsk: Classification and Regression Trees / CART) og "Lineær-modell-trær" (engelsk: Linear Model Trees / LMT), men til forskjell fra disse metodene bygges ikke ORT med grådige algoritmer som ofte fører til sub-optimale trær. NNRT ble valgt på grunn av den påstått økte ytelsen over ORT, i tillegg til de ulineære funksjonene i løvnodene som kan føre til trær med lavere dybde enn de tidligere nevnte metodene.

Trærne ble testet på ny usett data for å sammenlikne ytelsen med DRL-agenten, og den tidligere implementerte metoden LMT. Det bevises i oppgaven at ORT har jevnt over høyere ytelse enn LMT. Det bevises også at ORT kan fungere som en erstatning for DRL-agenten som er trent opp ved hjelp av PPO med lavere feilrate enn LMT. Gjennom kjente ytelsesmål for regresjoner viser resultatene at ORT approksimerte PPO-agenten bedre enn LMT, og at ORT kan approksimere alle fem pådragene til agenten ved å ofre noe nøyaktighet for en enorm økning i innsikt i agenten. Det bevises at ORT har lavere dybde og er mer nøyaktige enn LMT på bekostning av at man behøver ett tre for hvert pådrag i agenten.

Preface

This thesis concludes my Master's degree in Cybernetics and Robotics at the Norwegian University of Science and Technology, and is part of the Explainable Artificial Intelligence Systems for Gradual Industry Adoption (EXAIGON) project at NTNU. This thesis is a continuation of the project thesis written during the fall semester. As such, some of the material in Chapter 2 and 3 has been repurposed.

All code was written in Python on a workstation with an AMD Ryzen 9 3950X CPU with 64 GB RAM using the libaries PyTorch [1], Scikit-Learn [2], Numpy [3], Pandas [4, 5], and Matplotlib [6]. The work presented in this thesis builds on a framework developed by Ella-Lovise Hammervold Rørvik [7] as part of her master thesis. The deep reinforcement learning agent she developed was implemented using TensorFlow [8] in an OpenAI Gym [9] environment, based on an open source implementation by SpinningUp [10]. The Optimal Regression Trees implementation used in this thesis is part of a software package provided by Interpretable AI [11]. I would like to thank Jack Dunn at Interpretable AI who provided license keys to use the software.

I would like to extend my gratitude to my supervisor Anastasios Lekkas. He was integral to the success of this thesis, and has provided outstanding guidance through both my project and master thesis. I would also like to thank my co-supervisor Vilde Gjærum, who I have been fortunate enough to have technical discussions with in order to improve the results and work out technical problems, and who provided the Linear Model Tree implementation used in this thesis [12].

Thanks to my supervisors, I was able to submit my first journal article which got accepted to the International Federation of Automatic Control Conference on Control Applications in Marine Systems, Robotics, and Vehicles. The thesis and the journal paper which they have co-authored would not have been possible without their support and guidance.

Lastly, I want to thank my family and the friends I have made throughout my years at NTNU for their continuous support.

Jakob Løver
Trondheim, July 18, 2021

Contents

Figures

Tables

Acronyms

Adam Adaptive Moment Estimation. 28, 29, 64

AI Artificial Intelligence. 1, 2, 7, 17, 21

ANN Artificial Neural Network. 2, 7, 8, 33, 34

CART Classification and Regression Trees. 3, 22–24, 26

DL Deep Learning. 7

DRL Deep Reinforcement Learning. iii, 2–5, 21, 34

DT Decision Trees. 3, 7, 21, 22, 24

GDPR General Data Protection Regulation. 1, 2

IG Integrated Gradients. 2, 20

LIME Locally Interpretable Model-Agnostic Explanations. 2, 20

LMT Linear Model Trees. 3, 4, 24, 25, 37–40, 43, 46, 53, 58, 60, 62, 65, 68

MAE Mean Absolute Error. 42, 46, 58, 64, 65

MARS Multivariate Adaptive Regression Splines. 27

MDP Markov Decision Process. 16, 17

MIO Mixed Integer Optimization. 25

ML Machine Learning. 1, 7, 14

MSE Mean Squared Error. 10, 13, 42, 46, 57, 58, 65

NNRT Near-optimal Nonlinear Regression Trees. 4, 5, 27–30, 37, 39–41, 43, 64, 67, 68

ORT Optimal Regression Trees. 4, 25, 27, 37, 41, 43, 50, 51, 56–58, 62, 64, 65, 67, 68

PPO Proximal Policy Optimization. iii, iv, 17, 21, 34, 42–44, 46, 50, 53, 54, 57, 60, 65, 68

ReLU Rectified Linear Unit. 10, 34

RL Reinforcement Learning. 2, 15, 17

RSS Residual Sum of Squares. 15, 42

SGD Stochastic Gradient Descent. 11

SHAP Shapley Additive Explanations. 2, 3, 20

XAI Explainable Artificial Intelligence. 2, 7, 18, 19

Chapter 1

Introduction

Parts of the upcoming Section 1.1 has been adapted from the author's project thesis [13].

1.1 Background and Motivation

Autonomous systems are becoming increasingly ubiquitous in many industries, including the maritime industry [14]. Moreover, Machine Learning (ML) is gradually becoming a larger part of autonomous systems [15, 16], as tasks that may be simple for humans may be too complex or abstract for traditional non-ML algorithms. The accuracy of recent machine learning methods introduces systems that even perform better than humans in some tasks [17]. Machine learning is often applied to tasks that cannot easily be programmed by a set of rules, and may discover powerful underlying patterns in data.

Machine learning is also becoming a larger part of our everyday lives. Big data is being described as "the new oil", referencing the upcoming revolution having access to vast amounts of data brings. Big corporations like Facebook and Google are harvesting data from billions of users to use in their ML algorithms. The European Union therefore recently passed the General Data Protection Regulation (GDPR) to give its citizens control over how their data is used. If some algorithm determines an outcome, such as the right to a loan, this new regulation requires businesses to be able to trace down why the user was given a certain outcome [18]. This kind of policy dubbed *Right-to-Explanation* notes that an algorithmic decision has a large impact on an individual, and that individuals need to know how their data is being used. Additionally, the lack of ability to explain decisions also impedes the general acceptance of Artificial Intelligence (AI) and robot systems [19]. Advocates against this type of policy argue it will impose unnecessary restraints on AI and stifle many social and economic benefits [20], and that the need to create explanations hinders the performance of machine learning systems.

Reinforcement Learning (RL) refers to the process of training some algorithm called an agent to perform a task by maximizing some pre-defined reward function. Deep Reinforcement Learning (DRL) simply refers to the fact that the underlying model being trained is a neural network. The agent faces a sequential decision-making problem where, at every time step, it observes its state, performs an action, receives a reward and moves to a new state [21]. Instead of explicitly training the agent, the agent trains itself by trial-and-error. Reinforcement learning garnered public attention when AlphaGo from Deepmind, an RL agent trained to play the Chinese board game Go, managed to beat a human world champion. A recent study by Deepmind [22] argues that *reward is enough*, and RL agents guided by reward maximization is enough to achieve the holy grail of *artificial generalized intelligence*.

Despite the rapid adoption of machine learning systems, understanding the underlying decision making is becoming increasingly difficult. Black-box predictors such as Artificial Neural Network (ANN), which are often used due to their great generalization potential, are become increasingly complex in dimensionality and design. This poses a serious problem when humans need to be involved in the decision-making loop. Being unable to explain the reasoning behind black-box decisions is unacceptable for safety-critical systems. Explainable Artificial Intelligence (XAI) is a relatively new field within the AI community, popularized by the DARPA agency of the U.S. Department of Defense which aims to develop new or modified machine-learning techniques that will produce more explainable models [23].

It is clear that XAI tools are needed to assess the safety and reliability of machine learning systems. XAI can become a part of an engineer's toolkit when designing a new system, and provide the system designer with knowledge about what the system has learned. This could help shift the focus during the training process, and detect possible errors before the systems are put into use in the real world. XAI should also provide human-interpretable explanations, in light of the recently enacted GDPR.

The explainability of deep reinforcement learning agents built on neural networks has not been studied extensively [24], serving as motivation for writing this thesis. However, several efforts have been made to make machine learning more interpretable, such as the popular post-hoc explainers Shapley Additive Explanations (SHAP), Locally Interpretable Model-Agnostic Explanations (LIME), and Integrated Gradients (IG). The two former methods have been applied to a DRL agent trained to dock a vessel to harbor in order to provide explanations post-hoc

[25]. However, model-agnostic methods rely on post-hoc modeling of an arbitrary function, and can be slow and suffer from sampling variability [26].

There are several issues with using black-box models and applying post-hoc methods to obtain explanations instead of using more interpretable methods, as outlined by Rudin [27]. The author argues that explanations must be wrong, because a completely faithful explanation would mean the explanation is equal to the model being explained, and thus the original model is not needed. In fact, post-hoc methods may even mislead [28]. Previous studies show that SHAP may even be ill-suited to approximate neural networks [29]. Researchers have also been able to hide the biases of a classifier using SHAP and LIME, proving they can provide deliberately wrong explanations [30].

Additionally, Rudin argues that there is not necessarily a trade-off between accuracy and interpretability [27]. This can be seen in other literature as well, where interpretable models such as decision trees are in some cases proven to perform better than many black-box models on for example tabular data [31].

Instead, if the entire model behavior is visible, domain experts can validate whether the predictor is behaving as it should. There has been a trend of moving away from black-box models towards white-box models, particularly for critical industries such as healthcare, finances, and military, because of the need for understandable models [32]. According to [28], one should however exhibit caution if one decides to give up predictive power, and ensure that the desire for transparency is justified. For example the short-term goal of building trust with doctors by developing transparent models might clash with the longer-term goal of improving health care [28].

Apart from linear and logistic regression, Decision Trees (DT) are currently the most popular form of machine learning methods [33]. Most DTs are inherently interpretable, and as such are called "white-box" models. However, there are several issues with the decision tree methods that are currently in use. Even though Linear Model Trees (LMT) have proven to approximate a DRL agent trained for autonomous docking of an autonomous surface vehicle [12], the LMT used is quite deep. LMT and most other decision tree methods that build on Classification and Regression Trees (CART) [34] lack optimality, meaning they are built using greedy algorithms, which may cause the trees to be deeper than needed. Deeper trees introduce more parameters, making them harder to grow and harder to interpret. DTs with hard decision boundaries also suffer from weak expressivity [35]. Other variants of decision trees have already been attempted applied to reinforcement learning agents [36], but it is noted that the inherent instability of those decision trees limit the explainability [35].

1.2 Objective

The goal of this thesis is to investigate whether novel decision tree methods are able to satisfactorily approximate the decisions made by a deep reinforcement learning agent trained to dock a vessel to harbor. As outlined above, current methods such as LMT and LMTs are greedily optimized, and thus create deep trees that may not generalize well. It is desirable to investigate whether new state-of-the-art methods are able to better approximate the actions of the DRL agent using shallower trees. This may enable humans to partially or even fully understand the model behavior. The methods Near-optimal Nonlinear Regression Trees (NNRT) and Optimal Regression Trees (ORT) will therefore be investigated. The method which ORT is based on has already proven successful to provide human-readable explanations [37]. These methods will then be applied to tabular data generated by the agent to train trees that approximate its behavior. The performance of these methods over an entire docking episode will then be investigated.

1.3 Main contributions

- We have implemented the novel decision tree method Near-optimal Nonlinear Regression Trees (NNRT) from scratch. The NNRT was trained with data obtained from a deep reinforcement learning agent to approximate the agent's neural network. To the author's knowledge, no such implementation is public and has never been tested on a cyber-physical system nor a deep reinforcement learning agent.
- We have applied the method Optimal Regression Trees to the deep reinforcement learning agent. This implementation was provided by Interpretable AI [11]. We prove that this method is able to partially replace the neural network with high accuracy. The method was able to fully replace the neural network with higher accuracy than the linear model trees, though not with the same accuracy as the neural network.
- We provide quantitative analysis of the performance of each of these methods in terms of computational resources and accuracy. The methods will be juxtaposed with the ground truth DRL agent and a Linear Model Tree to observe the difference in performance. The implementation of the LMT was provided by Gjærum [12].
- We explore the interpretability of these methods by comparing feature importances, and discuss the discrepancies between these methods.

1.4 Thesis Outline

Chapter 2 will present the theory that serves as the backdrop for the methods used in this thesis. The chapter will cover fundamental theory behind neural networks, and how they are trained, as well as an introduction to deep reinforcement learning. Chapter 3 presents how the methods previously discussed were implemented to approximate the DRL agent, as well as an introduction to the docking problem. Chapter 4 presents the experimental results from applying the methods to the DRL agent. It contains extensive quantitative analysis of the performance of the methods used, and a discussion around the results obtained. Finally, Chapter 5 summarizes the findings and presents further work that may be done. The appendix covers some of the methods that were researched and attempted to be implemented, but not used either due to time constraints, difficulty of implementation, or lack of meaningful results or relevancy. The appendix also contains the code for the NNRT.

Chapter 2

Theory

The goal of this chapter is to provide the reader with a fundamental understanding of Artificial Neural Network (ANN), Decision Trees (DT), and Explainable Artificial Intelligence (XAI). This chapter is a continuation of the author's project thesis [13], and covers many of the same topics. Most of Section 2.1 and Section 2.3, and some of Section 2.4 was written during the project thesis, but has been revised for the master thesis. Section 2.5.2 was also written as part of the author's paper submission [25].

2.1 Deep Learning

Machine Learning (ML) is a branch of Artificial Intelligence (AI) focused on building applications that learn from data and improve their accuracy over time without being explicitly programmed to do so [38], meaning they adapt themselves when they see new data. Examples of popular ML algorithms are support vector machines, linear regression, and K-nearest neighbors. These methods are sometimes insufficient as they may not generalize well enough. Deep Learning (DL), a subset of ML, takes on a more holistic approach through building Artificial Neural Network (ANN). The main difference between ML and DL is that the latter does not require any feature extraction. In image processing, feature extraction can be done by performing some edge detection for example, which is usually required before applying machine learning algorithms. Feature extraction is a powerful inherent property of a neural network, but neural networks usually require larger amounts of data than other machine learning algorithms to perform adequately.

2.1.1 Neurons

The most basic unit of a neural network is called a neuron or perceptron. Every input to the neuron is multiplied by a weight that represent the how much of a role each input to the neuron should take in the output. Bias is added to this weighted sum, and an activation function is applied to the sum to obtain the output called activation.

The activation function describes how the weighted sum should affect the prediction. Non-linear activation functions are required to produce non-linear output [39]. Leaving out the activation function or using a linear activation function would in large part eliminate the need for using ANNs. Linear functions are additive, so any composition of linear functions can be reduced to a single linear function.

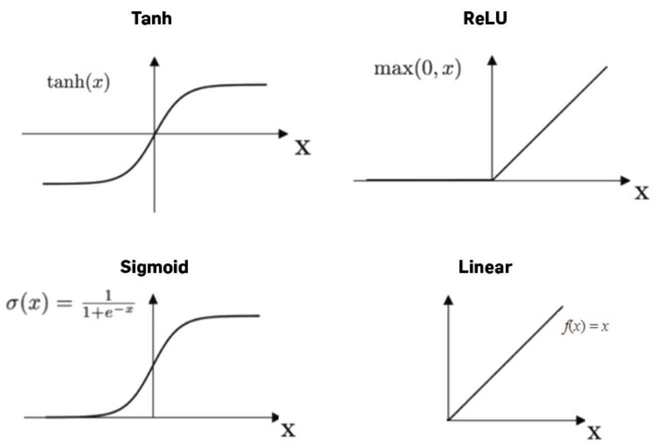


Figure 2.1: Four common activation functions [39].

2.1.2 Layers

Neural networks are organized in layers of neurons. In general, there is one input layer, one output layer, and an arbitrary number of hidden layers. The input layer receives the raw data input, whether it be the red-green-blue values of pixels in an image, or raw tabular data. The output layer squeezes the last hidden layer down to desired number of outputs from the neural network. The layers between the input and output layers are called hidden layers. The way the layers are connected together depends on the structure of the network. In a fully connected neural network—such as Figure 2.2—every neuron in every layer is connected to every neuron in the previous layer. A neural network that consists of no more than one hidden layer is usually called a "shallow neural network", while those with two or more hidden layers are often referred to as "deep neural networks."

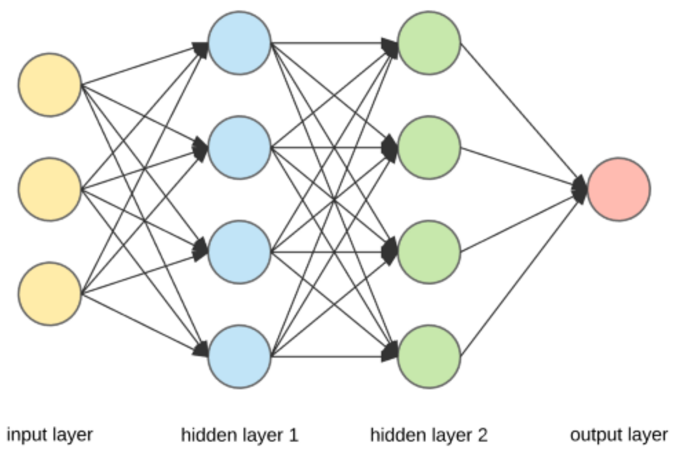


Figure 2.2: Example illustration of a fully connected deep neural network with two hidden layers [40].

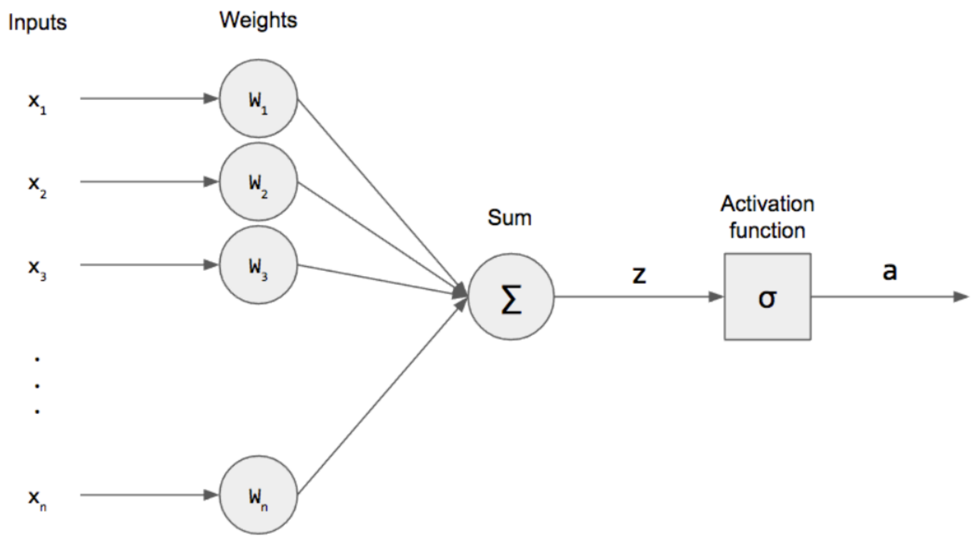


Figure 2.3: Visualization of perceptron [41].

$$
\begin{aligned}
&z = \mathbf{w}^T \mathbf{x} + b = \sum_{j=1}^{n} x_i w_i + b \\
&a = \sigma(z).
\end{aligned}
\tag{2.1}
$$

A neuron is commonly described using (2.1) where the vector of weights for inputs from the previous layer are denoted with **w**, and the vector of inputs (or activations from the previous layer) with **x**. This is illustrated visually in Figure 2.3. There is a tendency to add a node with a constant value b to the layers, which will be called a bias node. The bias node allows for shifting the activation function up or down. Applying the activation function σ to the weighted sum z gives the output a, also called activation. The most common activation function used at the time of writing this thesis is the Rectified Linear Unit (ReLU) activation function shown in Figure 2.1.

2.1.3 Training

Training is the process of updating all the weights to minimize the overall loss function. Loss is a measure that quantifies how good the predictions of the network was. First, a forward pass is performed through the network, which simply involves applying an input to the network and observing the output. The output is then used to compute the loss. The loss function depends on what problem is being solved. Classification problems— meaning those where the output from the network is discrete—may for example use the cross-entropy loss function. For example, image recognition usually deals with classifying what types of objects are present in pictures. Regression problems— where the output of the network is a continuous variable—may use the better known Mean Squared Error (MSE) loss function

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{2.2}$$

MSE is computed by summing the squared difference between the predicted value $\hat{y}_i$ and the true value y_i for every ith sample, and then dividing by the number of samples N. To minimize the MSE, the weights **w** must to be adjusted. This is done by using optimizers, which are in essence different strategies for adjusting the weights to minimize the loss. Most optimizers for neural networks are based on gradient descent, and the weights are adjusted according to the update step equation

$$w_n^{t+1} := w_n^t - \alpha\frac{\partial L}{w_n^t}. \tag{2.3}$$

For every time step t the weights are updated such that the loss function L decreases in the direction of its gradient. By applying a hyperparameter α called learning rate, the rate at which the weights are updated in the direction of the gradient is constrained to prevent overshooting the minimum of the loss function. This hyperparameter is usually chosen through trial and error.

Computing the gradient for all weights over the entire dataset can be extremely slow because of large amounts of weights. Stochastic Gradient Descent (SGD) is a drastically less computationally expensive solution, where the gradient is computed with respect to only a single data point at a time. Even though the gradient will be more aggressive because of potential outliers in the data, this strategy assumes that the combined effect of noise in the data on the gradient will be cancelled out over time. This behavior may even help getting out of local minima or saddle points, which is a real concern since the loss function neural networks are minimizing is usually not convex.

Another strategy, one which keeps the efficient computation time from SGD—but provides a more stable gradient—is called mini-batch gradient descent. Instead of computing the gradient of the loss function for every sample, it is computed in batches of samples. These matrix operations can often be computed in parallel using parallel processing units such as graphical processing units [42].

2.1.4 Generalization

The end goal of training is to train a model that generalizes well when it is exposed to new data. During training, it is therefore important to not overfit the model to the training data. Overfitting occurs when the model performs very well on the training data, but not as well when it sees new data. To combat overfitting, the dataset is often split into a training set, a test set, and a validation set. The training set may consist of for example 80 % of the total data available, 5 % for the validation set, and the remaining 15 % is reserved for the test set. Another phenomenon called underfitting is also a real concern, where the model has not yet fitted properly to the data. This is usually remedied by adding more parameters to the model.

When training begins, the weights are updated according to the chosen optimizer, but the total loss of both the validation set and training set is recorded after each new update. After a number of iterations, the total loss of the training set and validation set will eventually start to diverge. The loss of the training set may keep decreasing while the loss for the validation set will start to increase. This is a sign that further training will lead to overfitting. If the training stops before this point, the model will usually underfit. When training stops, the trained model is tested on the test data to observe how well it generalizes. Figure 2.4 shows how a model might behave when it overfits or underfits the training data.

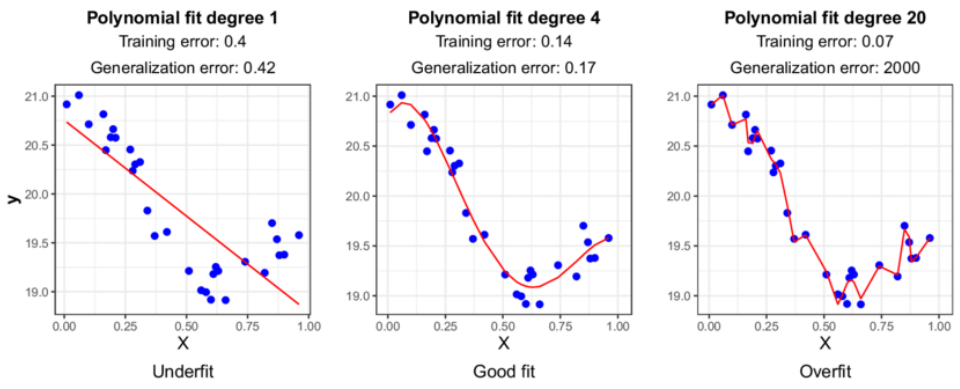


Figure 2.4: Example of overfitting and underfitting versus for a regression problem [43].

Bias refers to how the accurately the model is able to capture the underlying relationship between the variables. This is often connected with the number of parameters in the model. A model that is not able to capture the dynamics of the underlying data well will have high bias, and can often be remedied by adding more parameters in the model. This poses another problem however, as complex models often need more training data to generalize better. The fluctuations in the learned models will therefore be much larger for the more complex model than the simpler model [44]. These fluctuations are referred to as variance. In the machine learning community this balance between bias and variance is also known as the bias-variance trade-off.

2.1.5 Backpropagation

Seppo Linnainmaa laid down the groundwork in the 70s [45] for the method which today is called backpropagation. It is an essential building block in modern deep-learning frameworks, and is a procedure to compute the gradient of the cost function. By clever use of the chain rule, the gradient of the cost function can be efficiently computed and used to update the weights of each layer using the update step in (2.3).

The name "backpropagation" comes from the fact that the algorithm works its way from the last layer to the first layer after performing a forward pass through the network to calculate the gradient. The gradient of each layer of weights in the network is computed by making use of the activation in the previous layer a_{k-1}. As the neural network grows deeper and is able to capture more complex behavior, the time needed to perform this backpropagation increases.

The partial derivative of the loss function in (2.3) can be decomposed with the chain rule and rewritten as

$$\frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_k}. \tag{2.4}$$

For example, one may choose MSE as the loss function L

$$L = \frac{1}{2}(y - a_k)^2, \tag{2.5}$$

$$\tag{2.6}$$

which yields the following

$$\frac{\partial L}{\partial a_k} = y - a_k. \tag{2.7}$$

Choosing ReLU activation as the activation a gives the expression

$$a_k = ReLU(z_k) \tag{2.8}$$

$$\frac{\partial a_k}{\partial z_k} = ReLU'(z_k). \tag{2.9}$$

This derivative will be 0 for all $z \leq 0$ and 1 for all $z > 0$. Computing the derivative of the term in (2.1)

$$z_k = w_k^T x_k - b_k \tag{2.10}$$

$$\frac{\partial z_k}{\partial w_k} = a_{k-1}. \tag{2.11}$$

Then, $\frac{\partial L}{\partial w_k}$ can be computed as follows

$$\frac{\partial L}{\partial w_k} = a_{k-1} \cdot ReLU'(z_k) \cdot (y - a_k). \tag{2.12}$$

2.1.6 Momentum

The topology of the loss function often looks like rugged terrain. There are lots of local minima and saddle points. If the optimizer gets stuck in one of these points, methods are needed to steer the optimizer towards the global minimum. Momentum is a strategy that introduces a form of memory in the training loop to prevent getting stuck, by re-using a calculated term from the previous time step.

From [46], the update step in (2.3) can be rewritten as

$$z_n^{t+1} = \beta z^t + \frac{\partial L}{w_n^t} \tag{2.13}$$

$$w_n^{t+1} = w_n^t - \alpha z_n^{t+1}. \tag{2.14}$$

β is a constant between [0,1] that determines how much memory should be introduced. $\beta = 0$ results in the original update step in (2.3). This term is often set to a value close to 1. Momentum introduces smoother updates to the weights, improves convergence towards a better minimum, and prevents the optimizer from getting stuck.

2.2 Regression

In general, ML deals with two types of problems: classification problems and regression problems. Classification deal with grouping data into classes. For example, a model trained for image classification can be used to determine what type of object is present in an image. It can be trained to see the difference between species of animals or types of vehicles. Regression on the other hand is about relationships between data, and how a dependent variable can be estimated from one or more independent variables. Models trained for regression can output continuous functions as opposed to discrete classes.

2.2.1 Ridge regression

Linear regression describes the relationship between one dependent variable and several independent variables. The equations in this section are from [2].

Through linear regression, a linear function is fit to data, and results in a regression line of the form

$$\hat{y}(w,x) = w_0 + w_1 x_1 + \ldots + w_p x_p \tag{2.15}$$

where w is a vector of coefficients of the linear model, $\hat{y}$ is the predicted output, and x is the input data.

The process of fitting a regression line to data is commonly done through the least squares method. Least squares find the parameters w by finding a regression line that minimizes the Residual Sum of Squares (RSS). The residual is the distance between the observed variable y and the predicted variable $\hat{y}$. RSS is defined as

$$RSS = \sum_{i=1}^{N} (y_i - \hat{y})^2. \tag{2.16}$$

The function that least squares is trying to minimize can then be written as

$$\min_{w} ||Xw - y||_2^2 \tag{2.17}$$

where X is the a vector of data the regression line is fitted to.

Problems arise when using linear regression to fit a line to data where the assumption that all the variables are independent no longer holds. When the dataset is small, overfitting to the training data may also be of concern. Therefore, it is common to introduce L2 regularization to the loss function in (2.17). This is also called ridge regression, and the function to minimize can be written as

$$\min_{w} ||Xw - y||_2^2 + \alpha ||w||_2^2 \tag{2.18}$$

where $\alpha \geq 0$ is a parameter called *shrinkage*. Shrinkage determines how much the coefficients should be penalized. The idea behind penalizing large coefficients is that by sacrificing some of the accuracy in the training data, a new line that generalizes better on new data is created.

2.3 Reinforcement learning

Reinforcement Learning (RL) is the process of training an agent to take actions in an environment in which it may have little or no prior information. The goal is to guide the agent through the environment using rewards in order for it to take actions that maximize the cumulative expected reward. This is usually a trade-off between the reward the agent can expect to receive immediately, and the rewards it may receive in the future. As the agent performs actions according to some policy, it obtains more information about the environment. A policy can be seen as a function π that maps a state to an action.

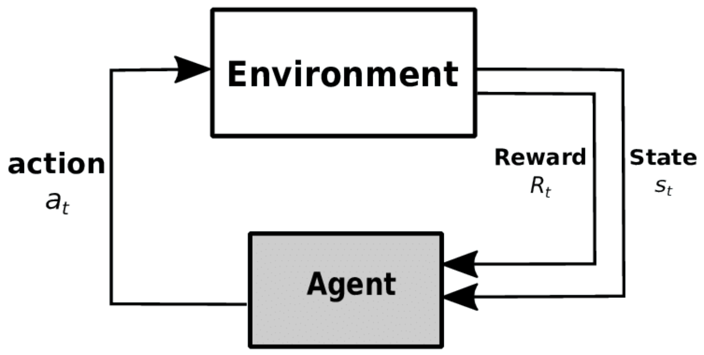


Figure 2.5: Illustration of the agent-environment interaction in reinforcement learning [47].

Contrary to supervised learning, the agent is not explicitly instructed regarding what to do. With deep reinforcement learning, this concept of "self-learning" is applied to neural networks by designing a neural network where the weights are updated through interactions with the environment.

2.3.1 Markov Decision Process

Reinforcement learning theory relies heavily on being able to express the problem at hand as a Markov Decision Process (MDP).

An MDP can be described as a tuple $\langle S, A, R, P, \gamma \rangle$ [48]

States S**:** A set of states.

Actions A**:** A set of actions an agent is allowed to take given a state.

Reward R**:** The reward function tells the agent whether or not a certain action was a wise decision, given a certain state.

Transition Probability P**:** A matrix of likelihoods of moving from one state to another when applying an action.

Discount factor $\gamma \in [0, 1]$**:** A multiplication factor that decides how heavy immediate rewards should be weighed against future rewards.

Central to the Markov decision process is the Markov property. The Markov property says that the current state encapsulates all the information that is needed to know what the next state will be. Therefore, the next state depends only on the current state and action, and it is not required to consider the past. If the underlying Markov decision process is known, the optimal policy can be computed directly. That is usually not the case, since the world is not entirely deterministic.

2.3.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) quickly became one of the most used reinforcement learning methods after its introduction by AI research company OpenAI. Contrary to model-based RL methods, where one needs to learn or know the underlying MDP, model-free methods such as PPO do not need that. Most RL methods being used today are model-free because of how much easier they are to implement. PPO is based on the idea that when updating the policy, the new policy should not be too far from the old policy—a concept called a trust region. The PPO-clip implementation will be discussed. The origin of the name becomes apparent when investigating the objective function. The goal of PPO-clip is to maximize the objective function in (2.21), and to use an optimizer (such as mini-batch gradient descent as discussed in Section 2.1.3) to update the policy π by updating the parameters θ of the policy through (2.19). The following equations and notations are from the PPO documentation page from OpenAI [49]. The parameters θ are updated through the optimization problem

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{E} \left[L(s,a,\theta_k,\theta)\right] \tag{2.19}$$

The advantage function can be defined as

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.20}$$

$Q^{\pi}(s,a)$ is called the action-value function, and describes how good an action a is to perform, given a state s. $V^{\pi}(s)$ is called the state-value function, and describes how good being in a state s is. The difference between these can then essentially be seen as a measure of how much "better" it is to take an action a, given a state s when following the policy π.

The objective function to maximize can be written as

$$L(s,a,\theta_k,\theta) = min\left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right) \tag{2.21}$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & A \geq 0 \\ (1-\epsilon)A, & A < 0 \end{cases} . \tag{2.22}$$

The term ϵ limits how much the policy is allowed to change. If the advantage is positive for a single state-action pair, meaning the action increases the objective, the objective function then reduces to

$$L(s, a, \theta_k, \theta) = min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a). \tag{2.23}$$

If the objective increases beyond a certain point, it will "clip" to prevent making updates far too large, hence the name PPO-clip. If, however, the advantage is negative, (2.21) reduces to

$$L(s, a, \theta_k, \theta) = max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s, a). \tag{2.24}$$

Because of the max term, the new policy will not improve beyond a certain point this time either if it takes steps too far away from the old policy.

2.4 Explainable Artificial Intelligence

Explainable Artificial Intelligence (XAI) methods are usually characterized as being either model-based or post-hoc [50]. Model-based methods such as linear regression and decision trees are machine learning methods that are interpretable by design [51]. Post-hoc methods are methods that are applied after the system has made a prediction, and is not concerned about how the model operates.

2.4.1 Interpretability

Interpretability is defined as the degree to which an observer can understand the cause of a decision [52], and can roughly be divided into the following levels [53]:

Local Interpretability Being able to explain reasoning behind single decisions or groups of decisions.

Global Interpretability Understanding the reasoning behind the entire model behavior on a holistic or modular level.

Moreover, the measures of accuracy *predictive accuracy* and *descriptive accuracy* may be defined [50]. Predictive accuracy refers to how well the model will adapt to the underlying relationships in the data. Descriptive accuracy is then defined as the degree to which an interpretation method objectively captures the relationships learned by machine learning models.

Figure 2.6 illustrates how using model-based or post-hoc methods affects predictive and descriptive accuracy.

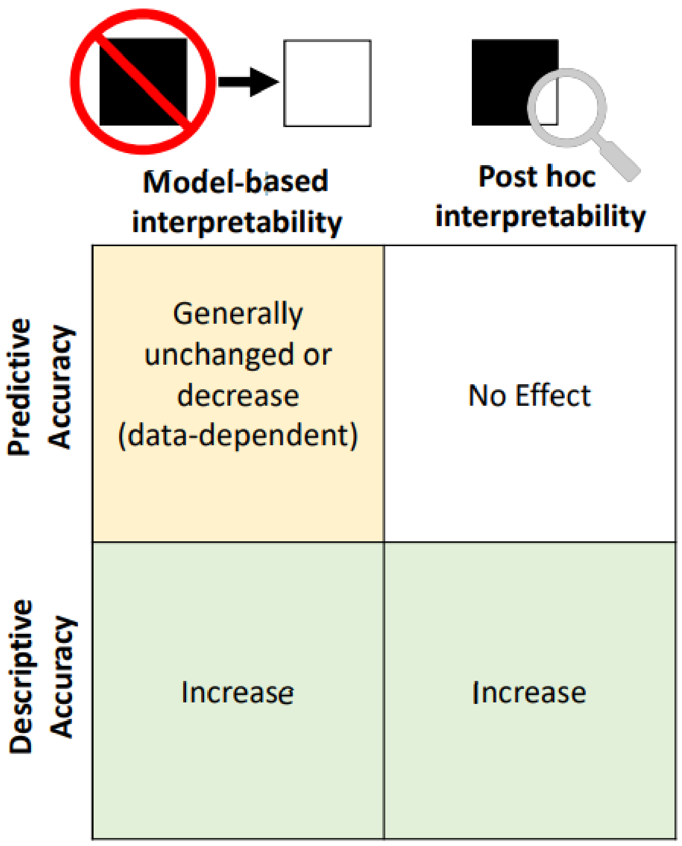

Model-based interpretability
Post hoc interpretability
Predictive Accuracy
Descriptive Accuracy
Generally unchanged or decrease (data-dependent)
No Effect
Increase
Increase

Figure 2.6: Impact of interpretability methods on descriptive and predictive accuracies [50].

2.4.2 Post-hoc Methods

XAI systems that only work for specific predictors are characterized as model-specific. If they are not concerned about the internal predictor structure, they are model-agnostic.

Locally Interpretable Model-Agnostic Explanations (LIME) is a method that creates explanations for a single data point. LIME creates a linear surrogate model that approximates the local behavior of the predictor for a single prediction. By creating a perturbed dataset made up of local perturbations around the decision in question, the importance of each feature in the black-box predictor can be inferred. The output from LIME is a vector of coefficients that suggests how increasing or decreasing variables affect the prediction [54]. It is a model-agnostic method that works for any black-box predictor.

Anchors is another method by the same authors as LIME. The method replaces the local surrogate model in LIME with IF-THEN statements called anchors. The linear surrogate model in LIME has unclear coverage, meaning it is unclear in what cases certain feature values contribute negatively or positively to the prediction. Anchor explanations "anchor" predictions to regions if certain feature values fulfill criterion. This method does suffer from many of the same issues as LIME. Overly specific anchors may be required to adequately explain a decision, which causes the explanation to lose some of its significance. Some anchors may also overlap, and overly complex output spaces may also be a factor. As with its predecessor LIME, finding a distribution of data that is representative can still be a challenge for some domains [55].

Shapley Additive Explanations (SHAP) is an XAI method based on the Shapley value, a coalitional game theory concept based on a set of fundamental axioms of fairness. It is proven that the Shapley value is the only solution that satisfies these axioms. The Shapley value therefore provides a unique solution to distribution of reward based on work contributed [56]. SHAP attributes feature importance to the inputs of a black box predictor for a single data sample by "removing" input features, and examining the resulting change in output by making use of the Shapley value [57]. SHAP has both model-specific (DeepSHAP) and model-agnostic (KernelSHAP) implementations.

Integrated Gradients (IG) is a form of model-specific, post-hoc XAI method. Its implementation is described more in detail in Appendix B, as it was not implemented due to incompatibilities between the method and docking agent which will be described in Section 3.2.

2.4.3 Imitation Learning

With the various problems state-of-the-art post-hoc methods possess, it is clear that better solutions are needed. The black-box nature of neural networks do not allow much insight into the internal workings of the network. However, if another more interpretable model was trained to approximate the behavior of the network, more insight could be gained into how the network actually behaves. Imitation learning may therefore be employed, where a typical approach is to train a regressor to predict an expert's behavior given training data of the encountered observations (input) and actions (output) performed by the expert [58]. Imitation learning is often associated with a machine learning by imitating a human expert. In this thesis however, the expert is a DRL agent trained through PPO as described in Section 2.3.2. The models that are trying to imitate the expert are various forms of decision trees, a topic which will be explained in the upcoming Section 2.5. The resulting trees might not perfectly imitate the DRL agent, but close approximations will give insight into what the DRL agent thinks, e.g. what states are important given certain states.

2.4.4 Symbolic Regression

Linear regression assumes that the underlying model can be expressed as a linear function. Linear regression then simply fits the parameters of a linear function to the data. Symbolic regression does not make any assumptions about the underlying model nor its parameters. Symbolic regression searches over possible models and parameters at the same time to fit the model to the data, instead of fitting the data to the model. Symbolic regression gathered public attention when the software Eureqa was made available by the AI lab at Cornell University. Eureqa used an evolutionary search, where some random equations would be applied and fitted to the data, and randomly recombined billions of times to find a better solution. A more recent prominent method which uses neural networks to auto-discover modularity is AI Feynman. However, due to time constraints, this method was not implemented. More information about the method can be found in Appendix A.

2.5 Decision Trees

Decision Trees (DT) are machine learning models that divide the input domain into subregions. The subregions are assigned a prediction by performing splits in the data. This can be visualized as a tree-like structure, where each of the splits are called internal nodes. When there are no more splits to be evaluated, a leaf node has been reached. The prediction depends on which leaf node, or subregion, the input data falls into. Regression trees are most often associated with decision trees where the predicted output is a constant real number. For DTs, the splits in the input data can be either multivariate or univariate.

Univariate splits Splits are created by evaluating one input variable

Multivariate splits Splits are created by evaluating multiple input variables at once

DTs with multivariate splits that tests linear combinations of variables are also called oblique DTs [59]. Most DTs also suffer from sensitivity to the training data. Removing a feature will result in drastically different trees. The axis-parallel decision trees may also yield complex tree structure and increase computational cost, when decision boundaries are not parallel to axes [60, 61]. This is illustrated in Figure 2.7. In these cases it is either desirable to rotate the training data or use multivariate splits.

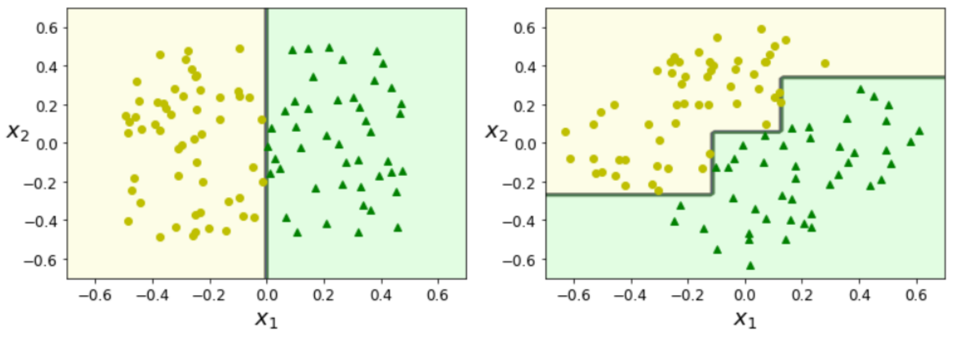


Figure 2.7: Illustration of how a rotation in training data affects the decision boundary of a decision tree [62]. More axis-aligned splits are needed to separate the training data in the figure to the right.

In terms of interpretability as discussed in Section 2.4.1, it is highly likely that decision trees are able to provide global interpretability if they are shallow enough.

2.5.1 Classification and Regression Trees

The decision tree method called Classification and Regression Trees (CART) by Breiman [34] is considered the seminal work on regression trees [63].

The algorithm works for both classification and regression problems. Depending on whether the problem is a classification or regression problem, a different splitting criterion is utilized. For classification problems, an impurity measure is used. The Gini impurity measure gives a probability of how likely it is that a sample falls into the wrong leaf. Each time a new split is considered, the split with the lowest Gini impurity score will be chosen. A regression tree built using the CART algorithm may for example attempt to minimize the mean squared error from (2.2) instead of the Gini impurity measure.

CART uses univariate splits, also called axis-parallel splits, to partition the input space and assign an output prediction to each subregion, also called a leaf node. The output prediction is the mean value of every data sample that falls into their respective subregion in the input space. This can be visualized as a tree structure. An example illustration of a regression tree and a visualization of how the input space is divided into subregions can be seen in Figure 2.8.

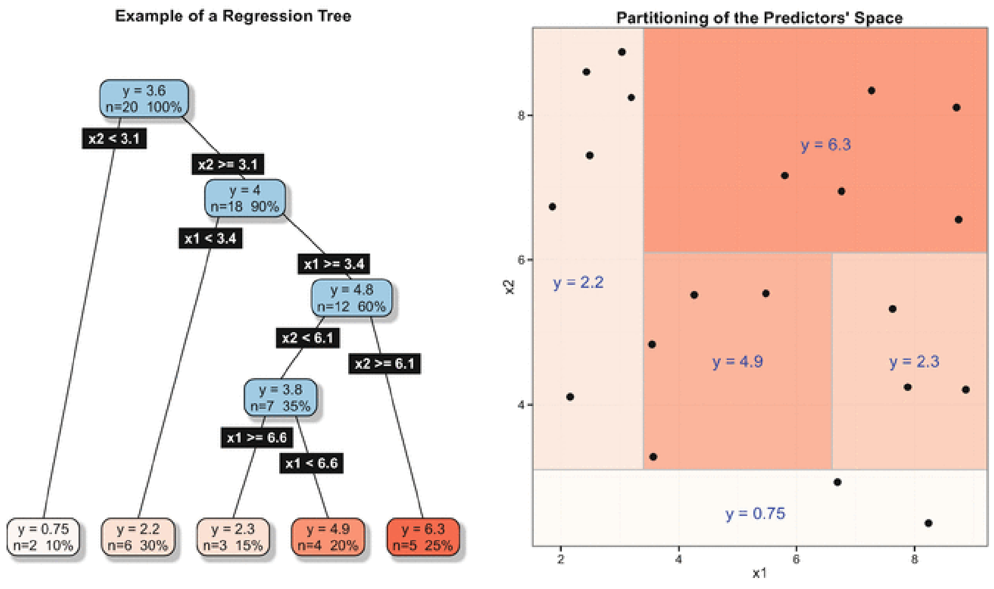


Figure 2.8: Illustration of a regression tree [63].

A CART is grown using a procedure called recursive binary splitting, meaning after every new split is created, a new split will be evaluated on the nodes resulting from the split. An overview of this process will be summarized from [64], where the equations and corresponding descriptions are from.

Given M regions $R_1, R_2, \ldots, R_M$ and the output of the tree is a constant c_m, the prediction function of the tree can be written as

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m). \tag{2.25}$$

The algorithm starts off by finding the best splits on the training data using a greedy algorithm, where the two resulting regions can be written as

$$R_1(j,s) = \{X | X_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{X | X_j > s\} \tag{2.26}$$

where j is the variable and s is the point in j which the algorithm will split on. If training a regression tree, the desired j and s are those that solve the minimization problem

$$\min_{j,s}\left[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2\right] \tag{2.27}$$

where c_1 and c_2 are constants equal to the averages of y_i in the respective regions. Once j and s have been found, the splitting may continue on the two new regions recursively until some stop condition. The stop condition may be some minimum number of leaf nodes, or a maximum depth in the tree.

The resulting tree T_0 is then pruned to create a subtree T such that $T \subset T_0$. Pruning means that internal nodes that do not contribute significantly to the minimization of the sum of squares are combined to lower the complexity of the tree. Pruning often allows the model to generalize better, as it prevents overfitting to the training data.

2.5.2 Linear Model Trees

It has been previously discussed that CART are DTs where the predicted output is a constant real number. Model trees on the other hand can output predictions based on any type of model. Linear Model Trees (LMT) is a regression tree where the only structural difference is that instead of constant predictions in the leaf nodes, an LMT uses a linear function to form its prediction. For the LMT implementation used, the splits on the input data are univariate. This is done to retain interpretability and reduce computation time. Growing an LMT is done by greedily splitting the data using a similar procedure as CART, which gives no guarantees for global optimality. A seemingly bad split may cause a good split to never be found [12].

By using an LMT to form a piece-wise linear approximation of a black box predictor, the simpler structure of the LMT can be used to understand the predictions made by the black box predictor. The resulting tree sacrifices some accuracy to give more interpretability. By weighting the linear regression coefficients in the leaf nodes, the predictions by the LMT can be interpreted. From Gjærum [12], the linear functions in the leaf nodes can be written on the form

$$y = \sum_{f \in \mathcal{F}} a_f x_f + C \tag{2.28}$$

where y is the model prediction. a_f is the linear regression coefficient for the feature x_f, and C is a constant. $\mathcal{F}$ is the set of all features. Total feature importance I_f for each feature f can then be calculated as

$$I_f = \frac{a_f x_f}{\sum_{j \in \mathcal{F}} |a_j x_j|}. \tag{2.29}$$

I_f then describes within the interval [-1,1] how much each feature f contributes to the overall prediction y. $I_f = -1$ would mean f is the only feature that matters to output a negative prediction, and vice versa.

2.5.3 Optimal Regression Trees

Optimal Regression Trees (ORT) is a novel method proposed by Bertsimas and Dunn. Instead of growing and pruning a tree greedily, the entire process is modeled as a Mixed Integer Optimization (MIO) problem. The method is described in detail in [65], but the algorithms from the book will be presented in this section. Though the original problem was formulated as an MIO problem, the authors found that to speed up computational time, a local search heuristic could be employed instead of standard MIO solvers. This section will therefore cover the local search method, which is the one used by the implementation in this thesis.

ORT-L is a variation of ORTs with the same structure as the LMTs in Section 2.5.2, where the splits are univariate and the leaves contain linear regressions. ORT-LH is another variation that introduces multivariate splits, but this thesis will focus on ORT-L. From now on, ORT-L will simply be referred to as ORT unless specified otherwise.

The overall objective function of the optimization problem can be formulated as

$$\min \frac{1}{\hat{L}} \sum_{i=1}^{n} (y_i - f_i)^2 + \lambda \sum_{t \in \mathcal{T}_L} ||\boldsymbol{\beta}_t||_1 \tag{2.30}$$

which is essentially the minimization of least squares between the prediction of the tree and the training data, plus a regularization term. $\mathcal{T}_L$ is the set of all leaf nodes, $\hat{L}$ is the baseline error in the training data, $\boldsymbol{\beta}_t$ is the vector of regression coefficients in the leaf node t, and λ is a regularization factor. This problem can be separated such that the objective function can be solved individually for each leaf using the using the `GLMNet` algorithm [66], which employs a coordinate-descent approach to minimize the objective function. The objective function can then be rewritten as

$$\min \frac{1}{2|\mathcal{I}_t|} \sum_{i \in \mathcal{I}_t} (y_i - f_i)^2 + \frac{\lambda \hat{L}}{2|\mathcal{I}_t|} ||\boldsymbol{\beta}_t||_1 \tag{2.31}$$

where $\lambda_{GLM} = \lambda \hat{L} / 2|\mathcal{I}_t|$, $\mathcal{I}_t$ is the set of data points that fall into leaf node t, and $|\mathcal{I}_t|$ is the number of data points in leaf node t.

To start off the local search, many different trees (for example 100) are created with different starting points with the hope that one of them will result in a tree close to the global minimum. Each of the trees are then used as inputs to Algorithm 1 together with the training data. LocalSearch loops through the nodes of the tree in random order. Using that node as a root node, a subtree is created. Each of these subtrees are then locally optimized using Algorithm 2. This procedure continues until the error does not improve. When the procedure stops, the resulting tree is locally optimal, and the procedure continues with a new tree.

Algorithm 1 LocalSearch [65]

```
    **Input**: Starting decision tree T; training data X, y
    **Output**: Locally optimal decision tree
 1: repeat
 2:     $\text{error}_{\text{prev}} \leftarrow \text{loss}(\mathbb{T}, X, y)$
 3:     for all $t \in \text{shuffle}(\text{nodes}(\mathbb{T}))$ do
 4:         $\mathcal{I} \leftarrow \{i : x_i \text{ is assigned by } \mathbb{T} \text{ to a leaf contained in } \mathbb{T}_t\}$
 5:         $\mathbb{T}_t \leftarrow \text{OptimizeNodeParallel}(\mathbb{T}_t, \mathbf{X}_{\mathcal{I}}, \mathbf{y}_{\mathcal{I}})$
 6:         Replace $t$th node in $\mathbb{T}$ with $\mathbb{T}_t$
 7:         $\text{error}_{\text{curr}} \leftarrow \text{loss}(\mathbb{T}, \mathbf{X}, \mathbf{y})$
 8:     end for
 9: until $\text{error}_{\text{prev}} = \text{error}_{\text{curr}}$
10: return $\mathbb{T}$
```

Algorithm 2 takes the subtrees discussed above as an input. Step 5 then calculates the loss of the subtree given the current data. Step 7 calculates a new univariate split on the current root node similar to CART by looping over all variables and possible splitting points. The algorithm may now perform one of three actions. If the loss of the subtree improved when the new split was created, the algorithm returns a new subtree with a branch node as its root node and two children. The two other options are to replace the current branch node with either the left or right child, also called the lower or upper child, making either one of them the new root node.

Algorithm 2 OptimizeNodeParallel [65]

Input: Subtree $\mathbb{T}$ to optimize; training data $\mathbf{X}, \mathbf{y}$.
Output: Subtree $\mathbb{T}$ with optimized parallel split at root.

```
 1: if $\mathbb{T}$ is a branch then
 2:     $\mathbb{T}_{\text{lower}}, \mathbb{T}_{\text{upper}} \leftarrow$ children($\mathbb{T}$).
 3: else
 4:     $\mathbb{T}_{\text{lower}}, \mathbb{T}_{\text{upper}} \leftarrow$ new leaf nodes.
 5: $\text{error}_{\text{best}} \leftarrow \text{loss}(\mathbb{T}, \mathbf{X}, \mathbf{y})$
 6:
 7: $\mathbb{T}_{\text{para}}, \text{error}_{\text{para}} \leftarrow$ BestParallelSplit($\mathbb{T}_{\text{lower}}, \mathbb{T}_{\text{upper}}, \mathbf{X}, \mathbf{y}$)
 8: if $\text{error}_{\text{para}} < \text{error}_{\text{best}}$ then
 9:     $\mathbb{T}, \text{error}_{\text{best}} \leftarrow \mathbb{T}_{\text{para}}, \text{error}_{\text{para}}$
10: $\text{error}_{\text{lower}} \leftarrow \text{loss}(\mathbb{T}_{\text{lower}}, \mathbf{X}, \mathbf{y})$
11: if $\text{error}_{\text{lower}} < \text{error}_{\text{best}}$ then
12:     $\mathbb{T}, \text{error}_{\text{best}} \leftarrow \mathbb{T}_{\text{lower}}, \text{error}_{\text{lower}}$
13: $\text{error}_{\text{upper}} \leftarrow \text{loss}(\mathbb{T}_{\text{upper}}, \mathbf{X}, \mathbf{y})$
14: if $\text{error}_{\text{upper}} < \text{error}_{\text{best}}$ then
15:     $\mathbb{T}, \text{error}_{\text{best}} \leftarrow \mathbb{T}_{\text{upper}}, \text{error}_{\text{upper}}$
16: return $\mathbb{T}$
```

Finally, the term $\alpha \cdot C$ is also added to the loss function. The complexity C is equal to the number of splits in the tree, and α is called the complexity parameter. α is a factor that balances adding new splits to a node and keeping the complexity of the tree down. This parameter can be automatically found through a grid search using a procedure the authors nicknamed *batch-complexity pruning*.

2.5.4 Near-Optimal Nonlinear Regression Trees

Near-optimal Nonlinear Regression Trees (NNRT) is another method devised by the some of the same authors as ORT. The method combines ideas from ORTs and Multivariate Adaptive Regression Splines (MARS) by including the splitting criterion in the prediction functions. Additionally, the method uses multivariate splits instead of univariate splits. The equations and descriptions in the following section are from [67].

An NNRT has polynomial prediction functions in the leaf nodes. The prediction function is constructed using a combination of the splitting criteria and a set of coefficients. The parameters that determine the splits in the tree are trained through gradient descent, more specifically through the Adaptive Moment Estimation (Adam) optimizer, which will be discussed later this section.

The model assumes the data is given on the form $(\mathbf{x}_i, y_i)$, where the following transformation is applied to the data

$$\hat{x}_{ij} := \frac{x_{ij} - x_{j,\min}}{x_{j,\max} - x_{j,\min}}, \; i \in [n], \; j \in [m]. \tag{2.32}$$

n represents the total number of samples in the dataset, and m represents the total number of features per sample. Given data on the form , this transformation ensures that $\boldsymbol{x}_i \in [0,1]^m$, meaning all features are normalized between 0 and 1 for all data points.

An NNRT has a fixed number of leaf nodes, meaning every branch node has two children each. The tree also has a fixed depth D. An NNRT consists of a set of nodes N, and a set of arcs A between the nodes. For every node ℓ in the tree, there exists an associated vector $\mathbf{a}_\ell$ and scalar b_ℓ. The arc the data sample $\mathbf{x}$ traverses down the tree is determined by evaluating whether $\mathbf{a}_\ell^\top \mathbf{x} < b_\ell$. If this condition holds true, the data will traverse down the left arc. Otherwise, it will traverse down the right arc. This evaluation will continue until a leaf node is reached. For any leaf node p, its prediction function $g_p(\mathbf{x})$ can then be expressed as

$$g_p(\mathbf{x}) = f_{p,D+1} + \sum_{i=1}^{D} f_{p,i} \prod_{\ell=\ell_1}^{\ell_i} |\mathbf{a}_\ell^\top \mathbf{x} - b_\ell|. \tag{2.33}$$

The overall prediction function $g(\mathbf{x})$ can then be written as a function of $g_p(\mathbf{x})$ such that

$$g(\mathbf{x}) = \sum_{p \in L} g_p(\mathbf{x}) \prod_{\ell \in \mathcal{L}_p^<} \mathbb{1}\{\mathbf{a}_\ell^\top \mathbf{x} < b_\ell\} \prod_{\ell \in \mathcal{L}_p^\geq} \mathbb{1}\{\mathbf{a}_\ell^\top \mathbf{x} \geq b_\ell\} \tag{2.34}$$

$$= \sum_{p \in L} \left(\sum_{i=i}^{D} f_{p,i} \prod_{\ell \in \mathcal{L}_{\ell_i}^<} \max(b_\ell - \mathbf{a}_\ell^T \mathbf{x}, 0) \prod_{\ell \in \mathcal{L}_{\ell_i}^\geq} \max(\mathbf{a}_\ell^T \mathbf{x} - b_\ell, 0) + f_{p,D+1} \right) \tag{2.35}$$

where L is the set of all leaf nodes and $\mathcal{L}_p$ is the set of all nodes the sample has traversed through on its way to the leaf node. $\mathcal{L}_p^\geq$ and $\mathcal{L}_p^<$ are the nodes the data traveled through when it traversed a right and left arc respectively.

Since $\mathbf{f}$ can be defined as the concatenation of all the coefficients f, $\mathbf{a}$ is the concatenation of all vectors $\mathbf{a}_\ell$ and likewise $\mathbf{b}$ is the concatenation of all vectors $\mathbf{b}_\ell$, the construction of the NNRT can then be written as the optimization problem

$$\min_{\mathbf{a},\mathbf{b},\mathbf{f}} L(\mathbf{a},\mathbf{b},\mathbf{f}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - g(\mathbf{x}_i))^2 + \lambda(||\mathbf{a}||_2^2 + ||\mathbf{b}||_2^2 + ||\mathbf{f}||_2^2). \tag{2.36}$$

This looks very similar to the loss functions seen earlier, using least squares and regularization parameters. The optimization of $\mathbf{a}$ and $\mathbf{b}$ can be separated from the optimization of $\mathbf{f}$. Since $\mathbf{a}$ and $\mathbf{b}$ are found through gradient descent, $\mathbf{f}$ can be calculated through ridge regression as discussed in Section 2.2.1. The ridge regression problem can then be written as

$$\min_{\mathbf{f}} L(\mathbf{f}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_{p_i,D+1} - \sum_{j=1}^{D} f_{p_i,j} \prod_{\ell \in \mathcal{L}_{\ell_j}} |\mathbf{a}_\ell^\top \mathbf{x} - b_\ell|)^2 + \lambda||\mathbf{f}||_2^2. \tag{2.37}$$

Algorithm 3 describes how the NNRT is built. It requires some initialization of the parameters $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{f}$. $\mathbf{a}$ and $\mathbf{b}$ are randomly initialized, and the subsequent $\mathbf{f}$ is calculated through ridge regression. This initialization is done M times, and the set of parameters with the lowest loss value $L(\mathbf{a},\mathbf{b},\mathbf{f})$ are selected as the starting point for the training loop. This technique is common when training neural networks, where clever initialization of the weights reduce the time it takes to reach the end of the training loop. Steps 5 through 10 are the steps taken by the Adam optimizer to update the parameters $\mathbf{a}$ and $\mathbf{b}$. This algorithm continues until the difference between the current loss and the previous loss is below a threshold ϵ.

Adam is an optimizer that is commonly used because of its stability when exposed to problems with noisy and sparse gradients [68]. Adam makes use of momentum as discussed in Section 2.1.6 by storing an exponentially decaying average of past squared gradients v_t in addition to an exponentially decaying average of past gradients m_t. The rates at which these averages decay are determined by the chosen coefficients β_1 and β_2.

The chosen learning rate α determines how fast Adam should update the parameters. A larger α will cause Adam to take larger steps in the direction of the gradient. ϵ is used for two purposes in this implementation. It is used as a numerical constant in the Adam optimizer to provide numerical stability by preventing division by 0 in step 9 and step 10 in Algorithm 3. It is also used in the training loop to stop the parameter updates when the difference between the current loss and the previous loss is lower than ϵ.

Algorithm 3 NNRT

```
1: Initialize $\mathbf{a}^{[1]}$,$\mathbf{b}^{[1]}$ and $\mathbf{f}^{[1]}$.
2: while $|L(\mathbf{a}^{[t-1]},\mathbf{b}^{[t-1]},\mathbf{f}^{[t-1]}) - L(\mathbf{a}^{[t]},\mathbf{b}^{[t]},\mathbf{f}^{[t]})| \geq \epsilon$ do
3:     $t \leftarrow t+1$
4:     $g_{t,1} = \nabla_{\mathbf{a}^{[t]}} L(\mathbf{a}^{[t]},\mathbf{b}^{[t]},\mathbf{f}^{[t]}), g_{t,2} = \nabla_{\mathbf{b}^{[t]}} L(\mathbf{a}^{[t]},\mathbf{b}^{[t]},\mathbf{f}^{[t]})$    ▷ Get gradients
5:     $m_{t,1} = \beta_1 m_{t-1,1} + (1-\beta_1) g_{t,1}, m_{t,2} = \beta_1 m_{t-1,2} + (1-\beta_1) g_{t,2}$    ▷ Update biased first moment
6:     $\hat{m}_{t,1} = m_{t,1}/(1-\beta_1^t), \hat{m}_{t,2} = m_{t,2}/(1-\beta_1^t)$    ▷ Get bias-corrected first moment
7:     $v_{t,1} = \beta_2 v_{t-1,1} + (1-\beta_2) g_{t,1}^2, v_{t,2} = \beta_2 v_{t-1,2} + (1-\beta_2) g_{t,2}^2$    ▷ Update bias-corrected second moment
8:     $\hat{v}_{t,1} = v_{t,1}/(1-\beta_2^t), \hat{v}_{t,2} = v_{t,2}/(1-\beta_2^t)$    ▷ Get bias-corrected second moment
9:     $\mathbf{a}^{[t]} = \mathbf{a}^{[t-1]} - \alpha \hat{m}_{t,1}/(\sqrt{\hat{v}_{t,1}} + \epsilon)$    ▷ Update the parameter $\mathbf{a}$
10:    $\mathbf{b}^{[t]} = \mathbf{b}^{[t-1]} - \alpha \hat{m}_{t,2}/(\sqrt{\hat{v}_{t,2}} + \epsilon)$    ▷ Update the parameter $b$
11:    Use ridge regression to calculate the value of $\mathbf{f}^{[t]}$ based on the value of $\mathbf{a}^{[t]}$ and $\mathbf{b}^{[t]}$.
```

To find the best tree, Algorithm 4 is used. The algorithm is initialized by determining the maximum depth of the tree D_{max} and a list of possible complexity parameters. The parameter λ is used in the loss function as a regularization parameter as described in Section 2.2.1. The algorithm iterates through every possible depth, and generates an NNRT with the given depth and complexity parameter. The best NNRT is the tree with the largest R^2 on the validation dataset.

Algorithm 4 Tuning the parameters for NNRTs

```
1: Set the maximum depth of NNRT $D_{max}$ and possible complexity parameters $\{\lambda_1, \dots, \lambda_j\}$.
2: for $D = 1, \dots, D_{max}$ do
3:     for $\lambda = \lambda_1, \dots, \lambda_j$ do
4:         Find NNRT with depth $D$ using complexity parameter $\lambda$.
5:         Add this NNRT to the solution pool.
6:     end for
7: end for
8: Identify the solution with largest $R^2$ on the validation set.
```

Figure 2.9 puts the above equations into context by illustrating the NNRT as a tree structure.

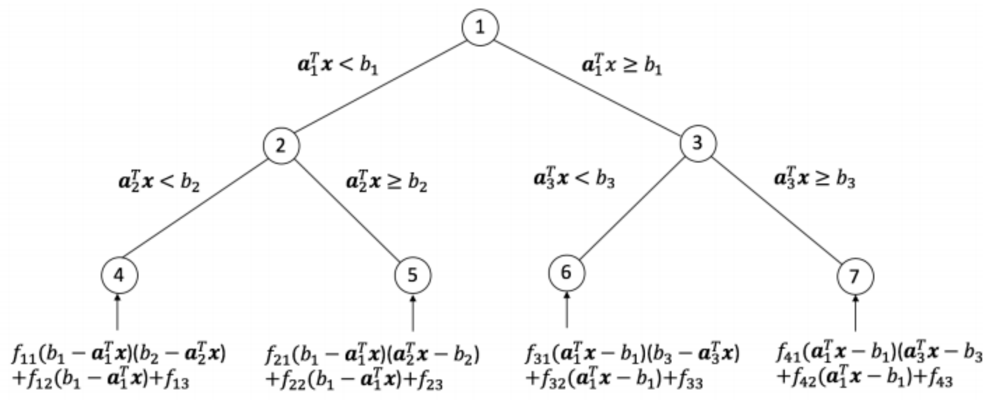

1
$a_1^T x < b_1$
$a_1^T x \geq b_1$
2
3
$a_2^T x < b_2$
$a_2^T x \geq b_2$
$a_3^T x < b_3$
$a_3^T x \geq b_3$
4
5
6
7
$f_{11}(b_1 - a_1^T x)(b_2 - a_2^T x) + f_{12}(b_1 - a_1^T x) + f_{13}$
$f_{21}(b_1 - a_1^T x)(a_2^T x - b_2) + f_{22}(b_1 - a_1^T x) + f_{23}$
$f_{31}(a_1^T x - b_1)(b_3 - a_3^T x) + f_{32}(a_1^T x - b_1) + f_{33}$
$f_{41}(a_1^T x - b_1)(a_3^T x - b_3 + f_{42}(a_1^T x - b_1) + f_{43}$

Figure 2.9: The prediction function $g_p(\mathbf{x})$ for each leaf node p is described along with the evaluations performed at each internal node in an NNRT of depth 2 [67].

Chapter 3

Design and Implementation

This section will cover the deep reinforcement learning agent the decision tree methods attempted to approximate, and how the methods were applied. Section 3.1 and 3.2 were written as part of the project thesis, and as such, has been adapted from the project thesis [13] and conference paper [25].

3.1 Docking

Docking involves various complex maneuvers to steer a vessel from the open sea towards a designated area in the harbor area called a berth. It has been characterized as one of the hardest problems to solve within ship control [69, 70]. Not only does the vessel need to take into account the speed limits of the harbor, distance to other ships and obstacles, but it has to simultaneously deal with extremely nonlinear motions, reduced maneuverability at low speeds, and environmental forces.

Historically, auxiliary devices such as tug boats have been used to dock large vessels, but with the increased freedom of maneuverability in the form of azimuth thrusters and tunnel thrusters, more sophisticated strategies can be employed. Performing automatic docking using auxiliary devices together with neural networks have already proven successful [69, 71, 72]. For example, [72] used a neural network architecture with one hidden layer to perform supervised learning. The ANN was trained from data collected by observing a skilled captain berth the vessel, but there are a multitude of reasons why this is a sub-optimal approach. The captain would have to berth the ship perfectly every time, which is not possible in practice. Errors are bound to happen, and the ANN will be entirely limited to the data provided. Even though the captain may have experience docking the vessel, the procedure the captain follows may not necessarily the most efficient for any given scenario. A major drawback with some of these aforementioned implementations is that they do not generalize well from one arbitrary port to another. These methods also had strict limits from what angle the vessel may approach the berth. Recent advances such as [73] allowed an ANN to berth both starboard and port side on multiple ports successfully without re-training, but did not take into account environmental forces such as wind and waves. Some publications were found using deep reinforcement learning to solve similar problems, but most of them were applied to underwater vehicles [74]. Other methods that have been proposed are backstepping controllers [75] and model predictive control [76].

3.2 Deep Reinforcement Learning Agent

Deep Reinforcement Learning (DRL) agents have already been shown to perform well in collision avoidance and trajectory following [77]. Using a deep reinforcement learning agent has been proposed as a solution to the docking problem [7]. While many of the aforementioned methods relied on real-world data collection and using supervised learning to train a docking agent, using DRL instead to solve the docking problem removes that dependency. The docking agent was proven to successfully solve the docking scenario from a variety of poses relative to the berth. It was trained using Proximal Policy Optimization (PPO) with two hidden layers of 400 hidden units each, ReLU activations for both hidden layers, and a hyperbolic tangent activation for the output layer.

The agent was trained on a three degrees-of-freedom vessel. It is 76.2 meters long, weighing 6000 tonnes in dead weight [76]. The vessel actuators are three thrusters: one tunnel thruster, and two azimuth thrusters. Their numbering and location on the vessel is shown in Figure 3.2. The tunnel thruster is used to create a side force on the vessel, while the two azimuth thrusters are mounted in the aft of the ship, and are rotatable thrusters. The action vector is described in (3.1). f_i is the applied thrust measured in newton, and a_i is the azimuth angle in radians for thruster i. Azimuth angles are measured from their neutral positions pointing forward. The azimuths are allowed to rotate 90 degrees clockwise, and 90 degrees counter-clockwise.

Since the docking problem as a whole is fairly complex, the training of the agent was divided into five separate learning phases. Every phase built upon previous phases, and solved sub-problems of the docking problem. An illustration of the relation between the learning phases can be seen in 3.1.

LP1 An agent commanded the vessel to hold a specified pose in the middle of the

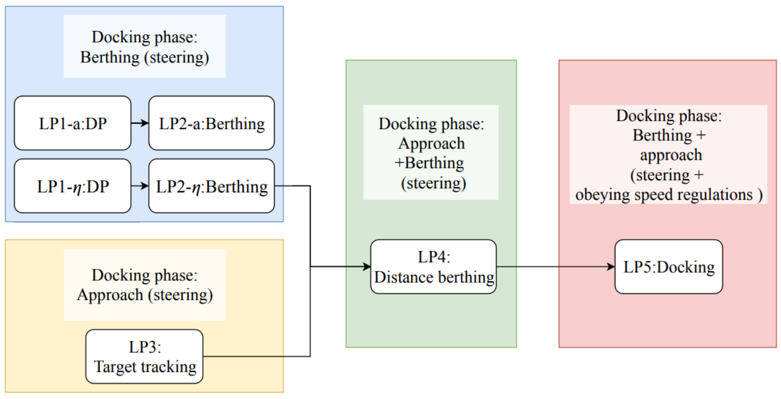


Figure 3.1: Illustration of learning phases [7].

harbor, performing dynamic positioning.

LP2 An agent commanded the vessel from the proximity of the berthing point to a desired berthing pose. Together with LP1, this learning phase solved the berthing phase.

LP3 An agent commanded the vessel from outside the harbor to the proximity of the berthing point, performing dynamic positioning.

LP4 Similar to LP2, an agent commanded the vessel to the berthing point from larger distances. This learning phase solved both the approach and berthing phases.

LP5 An agent commanded the vessel from outside the harbor area while keeping the harbor speed limits. This phase solved the entire docking problem.

This thesis uses an agent that is trained up to LP4 distance berthing. The valid ranges for the outputs of the agent in this learning phase are shown in Table 3.1.

$$y = \begin{bmatrix} f_1 & f_2 & f_3 & \alpha_1 & \alpha_2 \end{bmatrix} \tag{3.1}$$

Variable	Valid range
f_1 [kN]	$(-70, 100)$
f_2 [kN]	$(-70, 100)$
f_3 [kN]	$(-50, 50)$
α_1 [rad]	$(-\frac{\pi}{2}, \frac{\pi}{2})$
α_1 [rad]	$(-\frac{\pi}{2}, \frac{\pi}{2})$

Table 3.1: Valid output ranges for action vector.

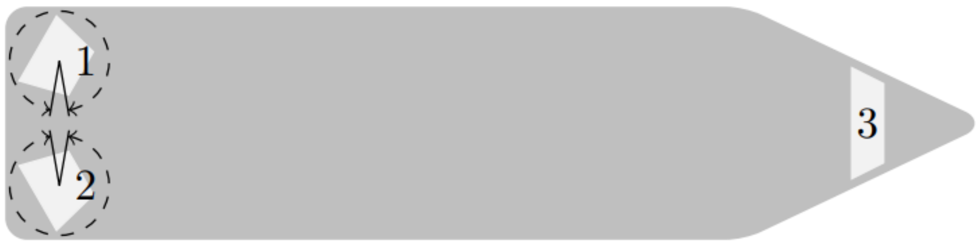


Figure 3.2: Thruster numbering on vessel [76].

The state vector for this learning phase is described in (3.2), and the respective state descriptions in Table 3.2.

$$x = \begin{bmatrix} \tilde{x} & \tilde{y} & l & u & v & r & d_{obs} & \tilde{\psi}_{obs} & \tilde{\psi} \end{bmatrix} \tag{3.2}$$

State	Description
$\tilde{x}$,$\tilde{y}$	The Cartesian distances from origin of the vessel to the target position, in body frame. x-direction is north-south, y-direction is east-west. Measured in meters.
$\tilde{\psi}$	The relative difference between heading of vessel and heading of the desired target. Measured in radians.
u,v,r	Linear and rotational velocities of the vessel, in body frame. Measured in meters per second and radians per second.
l	A binary variable describing whether the vessel is in contact with land. Only used when training the agent.
d_{obs},$\tilde{\psi}_{obs}$	The distance from the vessel's edge to the closest obstacle and the relative heading between the vessel and the closest obstacle. Measured in meters and radians respectively.

Table 3.2: Description of states [7].

3.3 Dataset

The data was collected by letting the reinforcement agent dock to harbor 500 times. The agent was placed at random initial starting points. Each docking episode had an episode length of 500 seconds with a sampling rate of 1 Hz. Altogether, the final dataset consisted of 242,384 data samples. The dataset was split according to Table 3.3.

Set	Number of samples	Fraction of total samples
Training	193662	79.90 %
Validation	11947	4.93 %
Test	36775	15.17 %

Table 3.3: Splitting of the dataset

The training set was used to train the ORT and the NNRT. The LMT was trained using a similar dataset, but with fewer samples and longer episode lengths. Furthermore, the test dataset was used to quantify the performance of each method on unseen data. Sampling from all of these locations allows the dataset to hopefully capture most of the edge cases. This is useful when the dataset is normalized during training. The more extreme values the dataset is able to capture, the better it hopefully will generalize on new data.

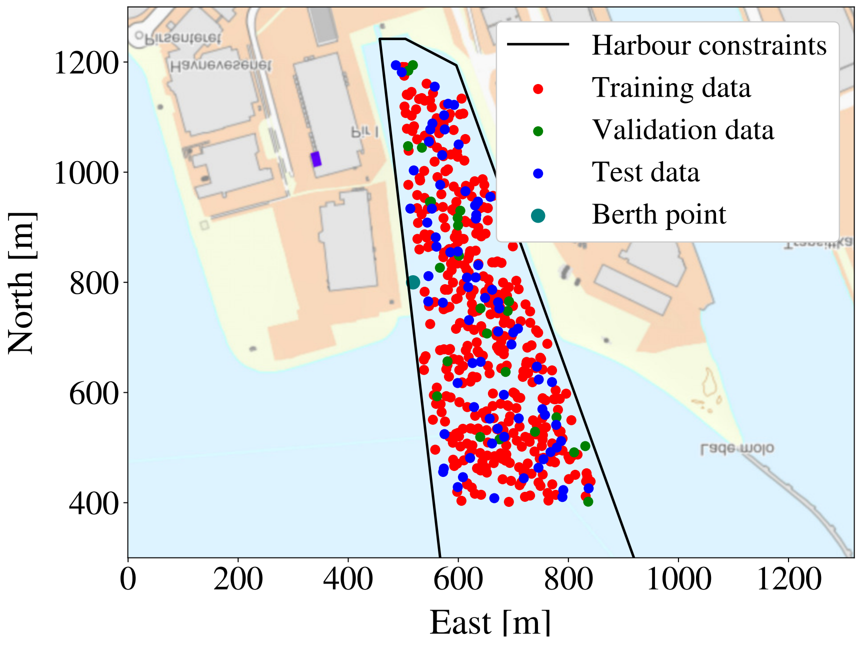


Figure 3.3: Starting points in the dataset.

For the trained ORTs and NNRTs, the dataset was constructed with the raw states, and normalized actions. This is different from the LMTs and the neural network, which were trained using normalized states. The largest reason for leaving the states un-normalized was to improve interpretability in the tree splits. Without having prior knowledge about the environment, it might be difficult to put the normalized states into context. It is magnitudes more intuitive to analyze the splits in a tree when the values of the split are represented using well-known units like meters and meters per second. When data is fed to neural networks, data is almost always without exception normalized to improve converge and reduce computational time.

The action vector on the other hand was normalized between [-1,1]. As seen from Table 3.1, the magnitude of the backwards force on the azimuths are not the same as the force going forward. This introduces another element (of confusion, if you will) one must know about the environment when examining the resulting trees. Information is already known about what the maximum and minimum thruster forces and azimuth angles are, and it is therefore unnecessary to assume this is already known by the one examining the tree.

One downfall of not using normalized states is that the coefficients of the resulting linear models in the leaves in and of themselves may be less interpretable. In theory, if the states were normalized well, the scale of the coefficients would indicate the importance of the respective states in the leaves. There might be cases when this is preferable, however, feature importance can still be calculated in each leaf according to (2.29).

3.4 Approximating the agent using decision trees

As mentioned, decision trees can be used to train approximate models of other less interpretable models. Deep trees are often needed in order to capture more complex dynamics in the data. However, trees that are too deep may lead to overfitting and may prohibit the tree from generalizing well. Deep trees may also severely restrict their interpretability, which is the main goal when approximating the agent. The goal must therefore to create as shallow trees as possible for maximum interpretability, without sacrificing too much accuracy.

3.4.1 Linear Model Trees

The LMT implementation used is based on an adaptation of classification and regression trees [78]. The modifications done allowed the tree to grow to a maximum number of leaf nodes instead of a maximum depth, and added randomization in the process of searching for thresholds and choosing the next node splits [12]. The LMT used contains 681 leaf nodes, with the shallowest leaf node situated at depth 5, and the deepest at depth 15.

The LMT used was already trained with data. It was trained using normalized states, where the states were normalized by subtracting the mean and dividing by the standard deviation for every state, and normalized actions between [-1,1]. The output from the LMT was then multiplied by a scalar to recover the original scale of the actions.

3.4.2 Optimal Regression Trees

The implementation used is from the `Interpretable AI` Python package, and was created by a company with the same name [11] who provided the author with academic licenses. The implementation is originally written in Julia, but Python bindings are provided. For this thesis, optimal regression trees with linear predictions in the leaves were used. The object *OptimalTreeRegressor* was therefore used to grow the tree, which was embedded inside of a *GridSearch* object. The grid search was employed to perform automatic discovery of the optimal complexity parameter α. A training loop was set up to iterate over all possible depths up to a maximum of 10, record the evaluation metrics in Section 3.7 on the test data for each depth, and save the tree to a JSON file. The JSON files of the chosen trees was used to recover the trained trees when replacing the actions of the docking agent.

3.4.3 Near-optimal Nonlinear Regression Trees

The implementation of the NNRTs was made from scratch in Python. It was implemented as a module in PyTorch [1], a deep learning framework developed by Facebook. This made the NNRT implementation compatible with the optimizers and other features the framework provides. PyTorch operates using *tensors*, equivalent to a multi-dimensional matrix. As PyTorch is mainly focused on neural networks, few examples of decision trees are available. The tree structure used in the NNRT implementation was therefore inspired by an implementation of Soft Decision Trees [79].

The maximum depth of the tree D_{max} was first decided. This was set to 4, since no signifcant improvements were seen in NNRTs with depths deeper than 4 in the original paper [67].

As described in Section 2.5.4, the matrices **a**, **b**, and **f** were initialized randomly to give the method a warm start. **a** and **f** were implemented as two-dimensional tensors. **b** was implemented as a one-dimensional tensor. The dimensions of each tensor can be illustrated with the following equations

$$\mathbf{a} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_{\ell_i} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{\ell_i,1} & a_{\ell_i,2} & \cdots & a_{\ell_i,m} \end{bmatrix} \tag{3.3}$$

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{\ell_i} \end{bmatrix} \tag{3.4}$$

$$\mathbf{f} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \end{bmatrix} = \begin{bmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,D} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ f_{p,1} & f_{p,2} & \cdots & f_{p,D} \end{bmatrix} \tag{3.5}$$

where D is the depth of the tree, p is the number of leaf nodes, and m is the number of features. ℓ_i is the ith branch node, also called internal nodes. The number of nodes in an NNRT can be calculated as

$$\text{\# Branch Nodes} = 2^D - 1 \tag{3.6}$$

$$\text{\# Leaf Nodes} = 2^D \tag{3.7}$$

which were used to define the shape of the tensors. In contrast to the LMTs described in Section 2.5.2, NNRTs are built with a fixed depth, and a fixed number of leaf nodes. This causes the number of parameters to increase exponentially. The splits are also multivariate, as opposed to LMT's univariate splits. Even though the tree may be less interpretable than LMTs, the multivariate splits allow shallower trees, and the nonlinear prediction functions is able to better capture the dynamics of the system.

NNRT requires the input features to be normalized between 0 and 1 as described in (2.32). It is beneficial to keep the same scale on the features, as this brings more stable convergence to the training. The prediction functions in the leaf nodes contain successive multiplications, so scaling the features to the range [0,1] avoids large predictions during training. This normalization is therefore also done to help avoid the exploding gradient problem. The data was scaled using the *MinMaxScaler* from the `scikit-learn` Python package [2].

Normalizing is trivial when evaluating how well a model performs on a dataset, as long as the normalization is done consistently on both the training and test data. This proves to be more of a challenge when the NNRT is to be used as a replacement for the neural network. The normalization needs to be done using normalization constants calculated from the environment. Setting minimum and maximum limits on the features may unnecessarily restrict the agent's ability to generalize from harbor to harbor. This transformation was still done to investigate the feasability of the method without spending too much time.

The concatenated vector of coefficients **f** was calculated using the ridge regressor from `scikit-learn` [2]. The original algorithm was also improved upon by implementing mini-batches. When the dataset gets sufficiently large, the gradient and its backward pass through the computational graph takes too long to compute. Therefore, it is desirable to optimize on fewer samples at a time. This greatly improved both the convergence of the algorithm, and cut down on several orders of magnitude of computational time.

The implementation of the NNRTs can be found in Appendix C.

3.5 Hyperparameters

The hyperparameters used for each method are summarized in Table 3.4 and Table 3.5. In Table 3.4, the list of regularization parameters allowed the ORT implementation to choose the regularization parameter that resulted in the best performing tree.

Hyperparameter	Value
Regularization parameters λ	$0.00001, 0.0001, 0.005, 0.001$
Max depth $D_{\max}$	10

Table 3.4: Hyperparameters for ORT

Hyperparameter	Value
Regularization parameter λ	0.001
Learning rate α	0.001
Constant ϵ	0.0000001
Batch size N	128
Random initializations M	50
Max depth $D_{\max}$	4

Table 3.5: Hyperparameters for NNRT

3.6 Computational Hardware

The simulations and computations were performed on a workstation with Ubuntu 20.04. The workstation is powered by an AMD Ryzen 9 3950X CPU, and is equipped with 64 GB of RAM.

3.7 Evaluation Metrics

Three evaluation metrics were used when examining the performance of the methods: Mean Absolute Error (MAE), Mean Squared Error (MSE) seen in (2.2), and the coefficient of determination R^2.

MAE is a measure of how large the output error of an approximated is on average. This is simply calculated as the absolute value of the error between the ground truth (the PPO agent) and predictions [2] and can be described by the equation

$$MAE = \frac{1}{N}\sum_{i=i}^{N}|y_i - \hat{y}_i| \tag{3.8}$$

The coefficient of determination R^2 gives a measure of the proportion of variance of the dependent variable y that has been explained by the independent variables [2]. Therefore, it gives an indication of how well the models approximate the PPO agent, and how well they will generalize to new unseen data. From [2], R^2 is a function of RSS from (2.16), and can be expressed as

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{3.9}$$

where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. The best possible R^2 is therefore equal to 1 when the approximated model perfectly approximates the ground truth.

Chapter 4

Results

Section 4.1 shows the PPO agent maneuvering the environment throughout one docking episode. This will be used as a benchmark for how the other methods performed, using the same starting point.

The performance of the linear model tree will be presented in Section 4.2. Each of the actions of the PPO agent will be replaced in parts or fully by the LMT to investigate how well it performs in practice.

The experiment in Section 4.3.2 investigates whether an ORT can function as a partial replacement for the force f_3 of the neural network trained through PPO. It is desirable to investigate how well this will work in practice, because it can allow insight into a black-box predictor with a large action vector without necessarily approximating every action of the predictor. If the ORT is able to approximate the this force close enough, we should observe a similar failure rate to that of the PPO agent. Contrary to the azimuth thrusters, the tunnel thruster is in a fixed position on the vessel and can only generate a sideways force on the vessel. This makes it ideal for isolated analysis. This section goes in-depth to investigate the ORT approximation of the tunnel thruster at depths 1 and 8. Finally, the results for the tunnel thruster f_3 will be summarized.

In Section 4.3.3, the results will be summarized for the four remaining actions, and an attempt to fully replace the neural network with five ORTs will be made.

Section 4.4 summarizes the results when attempting to train an NNRT to approximate the tunnel thruster f_3.

Finally, a comparison between all the methods will be done in Section 4.5.

4.1 PPO

The PPO agent was allotted 1000 seconds to dock, and successfully docked the vessel to harbor in about 400 seconds. The states of the vessel are shown in Figure 4.3 and the actions are shown in Figure 4.2. This docking scenario had the initial poses shown in Table 4.1, and will be used as a baseline for the upcoming methods. When letting the PPO agent dock the vessel 100 times with random initial poses, only two crashes into the quay were observed.

State	Initial value
x	700 m
y	650 m
ψ	-0.384 rad
u	0.3 m/s
v	0 m/s
r	0 rad/s

Table 4.1: Initial pose when docking to harbor.

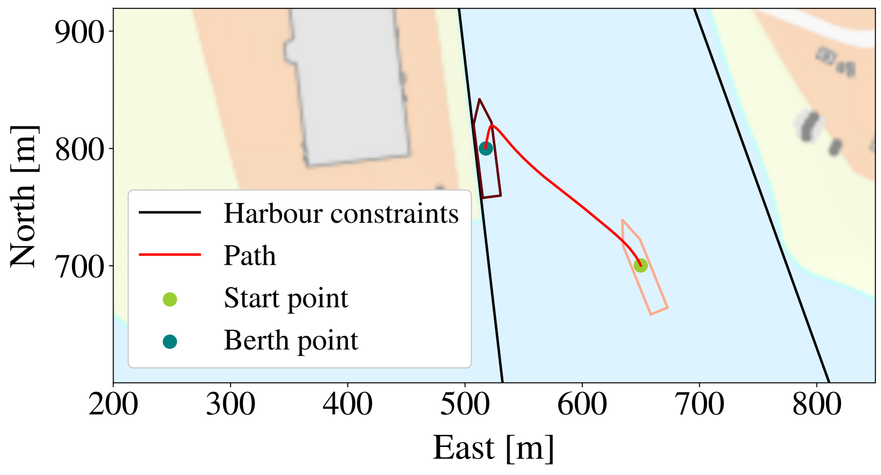


Figure 4.1: Vessel trajectory of the PPO-agent.

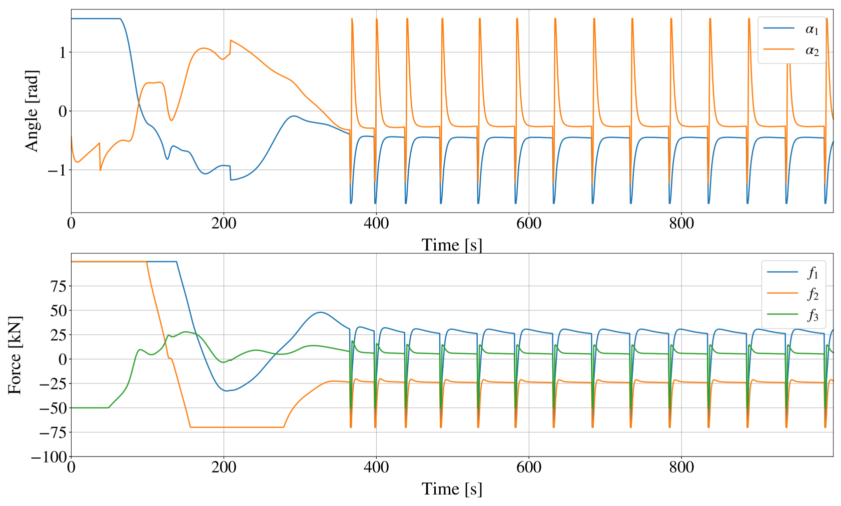


Figure 4.2: Vessel actions of the PPO-agent.

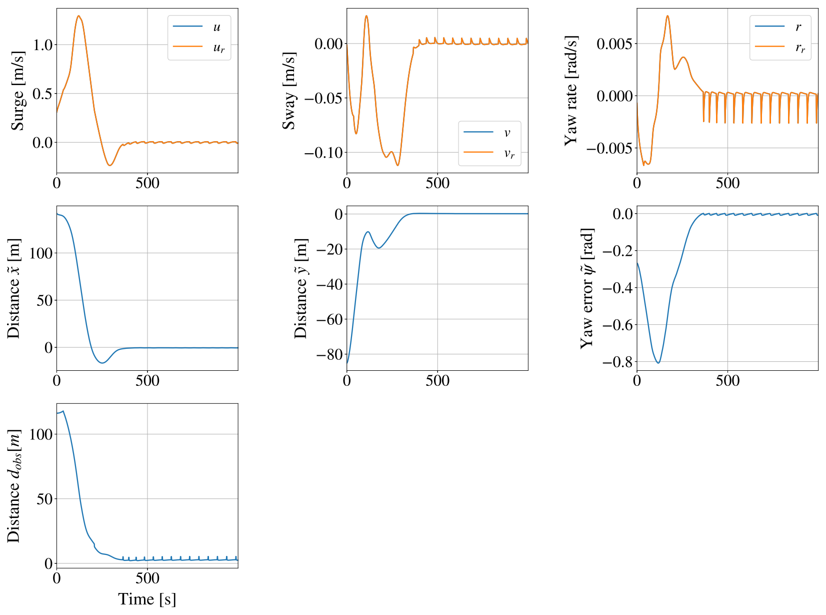


Figure 4.3: Vessel states of the PPO-agent.

4.2 LMT

Table 4.2 shows the MAE, MSE, and out-of-sample R^2 calculated on the test dataset described in Section 3.3. The failure rate is computed by letting the LMT control the action indicated, and letting the PPO control the rest of the actions for 100 episodes. The percentage indicated shows how many times the vessel collided with the quay. This quantifies how well the LMT approximates each action by letting it run in a real-world scenario.

Replaced action	MAE	MSE	Out-of-sample R^2	Failure rate
f_1	0.1130	0.0369	0.9255	13 %
f_2	0.1209	0.0457	0.8967	8 %
f_3	0.0956	0.0468	0.8304	8 %
a_1	0.1322	0.0605	0.7025	9 %
a_2	0.1242	0.0573	0.7228	7 %
All actions	0.1172	0.0494	0.8156	12 %

Table 4.2: Evaluation metrics for the trained LMT on each output on the test set.

Overall, a 12 % failure rate was seen when the LMTs were in control of the entire vessel. This is a little higher than the failure rate of the other actions except for f_1. It is also noted that the two actions with the highest isolated failure rates were f_1 and a_1, both actions that belong to the left azimuth. This might be a bit expected, because the quay is located on the port side of the ship when it is docking. It is suspected that this thruster has more complex dynamics than the other thrusters. It is therefore harder to approximate, and is quite important to hinder the vessel from colliding.

The vessel was initiated with the same starting pose as in Section 4.1, and the LMT was in control of all actions. Figure 4.4 shows the trajectory of the vessel, Figure 4.5 shows the actions, and Figure 4.6 shows the states of the vesssel for the entire docking episode. The vessel overshoots the berthing area, as can be seen from the dip in $\tilde{x}$ in Figure 4.6 right after 200 seconds into the episode. Intuitively, the vessel should therefore work to align itself with the berthing area in this direction. When observing the feature importance for the LMTs in Figure 4.7, the LMT seems to place importance on the surge speed u for the action f_2 during this period. Meanwhile, the tunnel thruster is working to prevent the vessel from crashing into the quay, as observed with its increased importance on d_{obs}. For the rest of the episode, quite high importance is placed on all states for d_{obs}, $\tilde{\psi}_{obs}$ and $\tilde{\psi}$. This is during the critical phase of the docking scenario where the vessel is in the vicinity of the berth, but needs to align itself correctly.

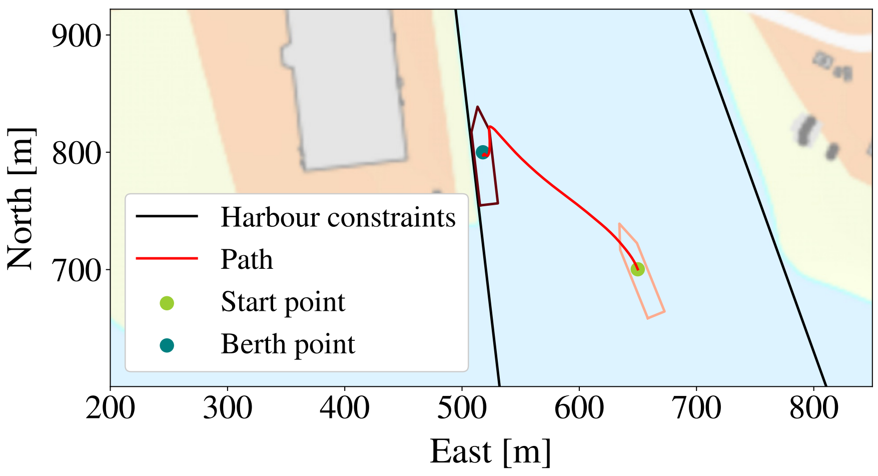


Figure 4.4: Trajectory map when LMT replaced the PPO agent.

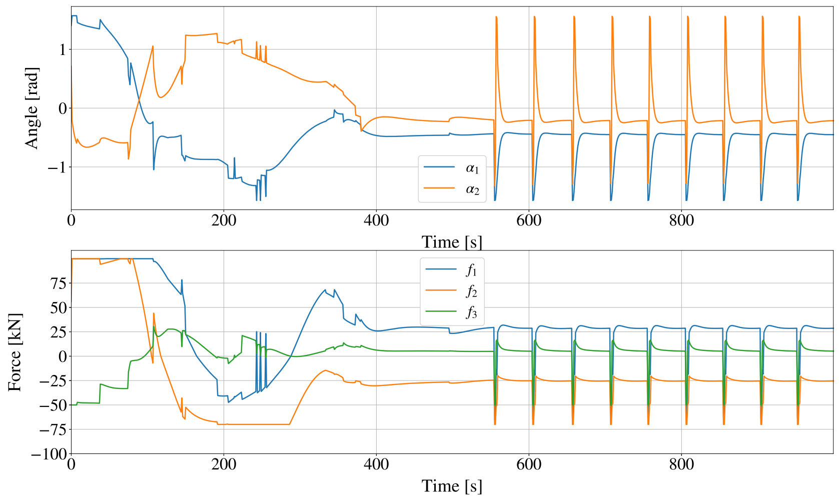


Figure 4.5: Actions when LMT replaced the PPO agent.

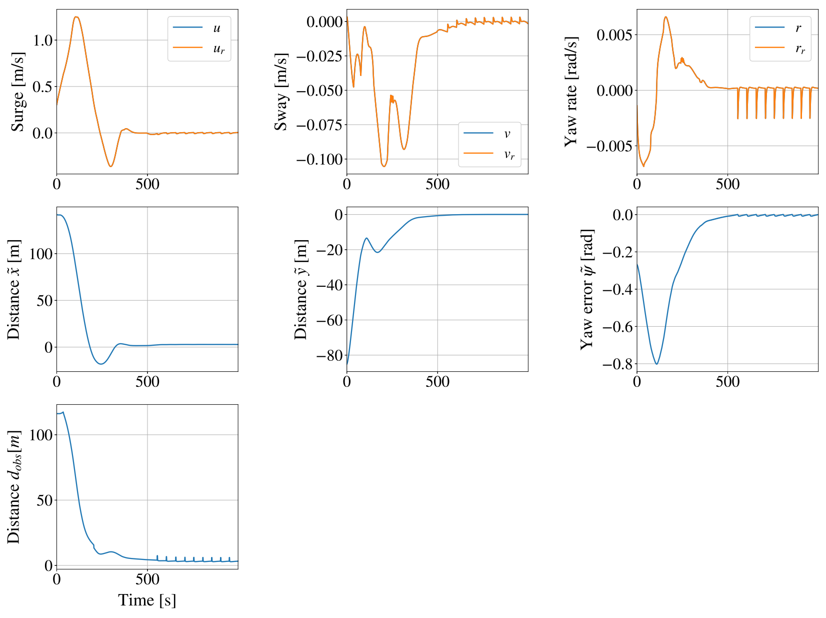


Figure 4.6: States when LMT replaced the PPO agent.

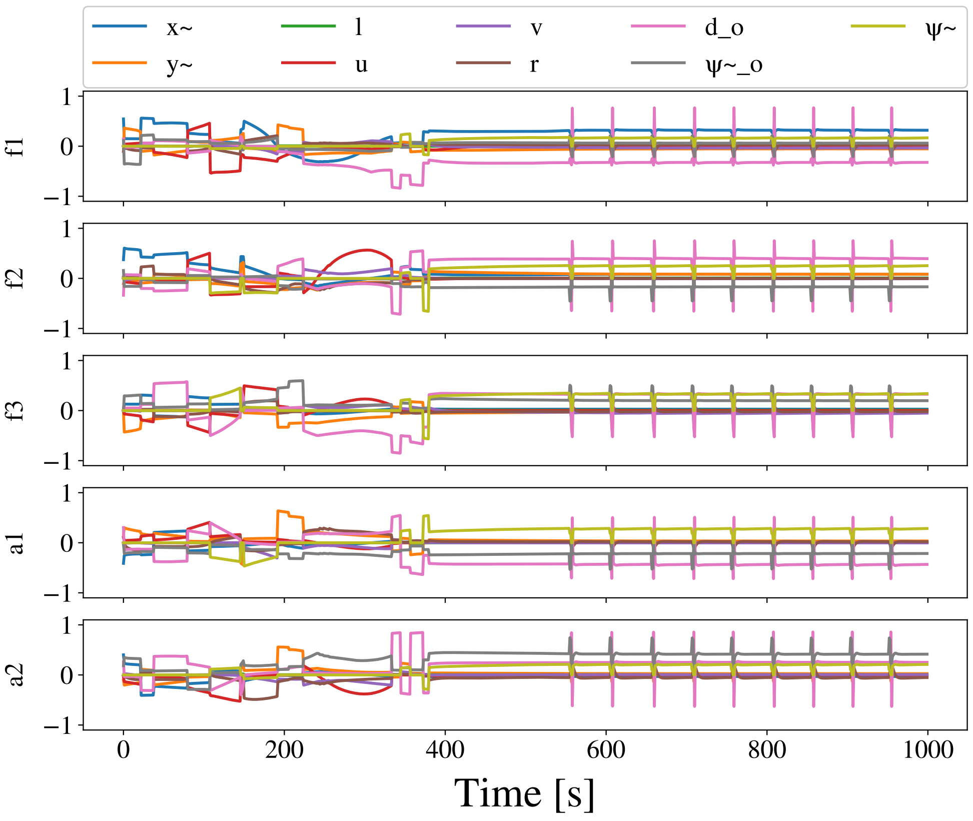


Figure 4.7: Feature importances when LMT replaced the PPO agent.

4.3 ORT

4.3.1 Computational time

Figure 4.8 shows how long it takes to train trees of various depths with training data provided by the PPO agent. Though this varies depending on the number of samples in the training data, it gives an idea of the rapid increase in computational time depending on the depth of the tree.

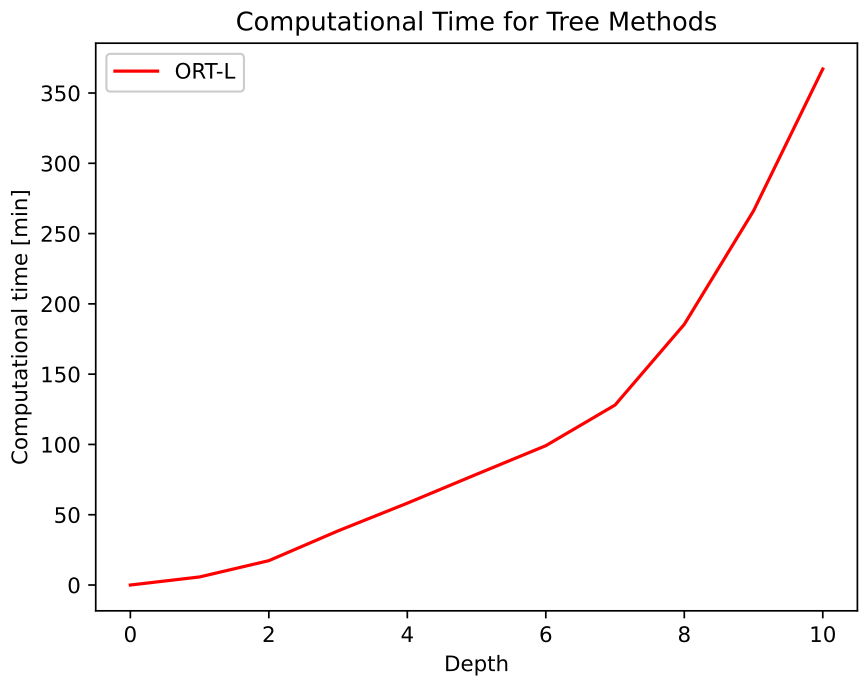


Figure 4.8: Computational time for various depths of ORTs

4.3.2 Approximating the tunnel thruster

ORT with depth 1

The ORT with depth $D = 1$ was trained to control the tunnel thruster, and was initiated with the same pose as in Section 4.1. The vessel trajectory is seen in Figure 4.9, its actions in Figure 4.10, and its states in Figure 4.11. It did not perform adequately enough to solve the docking problem. The PPO agent in Figure 4.1 used the tunnel thruster quite extensively initially during the docking episode at around 100 seconds and then correctly applied a positive tunnel thruster force to rotate the vessel into the right position. The ORT on the other hand, as can be seen from Figure 4.10, does not apply a large enough positive force on f_3 to prevent it from colliding with the quay.

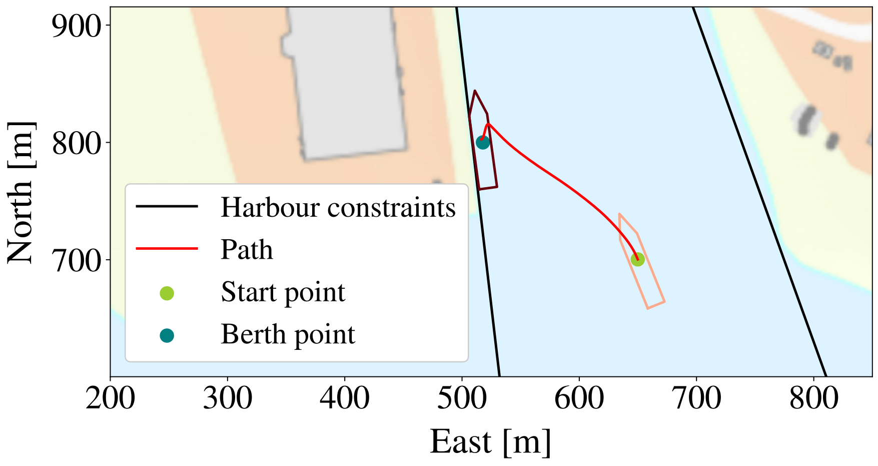


Figure 4.9: Vessel trajectory of a failed docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 1.

Figure 4.12 illustrates the ORT. The vessel will utilize one of two different linear regression depending on the state $\tilde{\psi}$. It is seen that the ORT learned to split on the state $\tilde{\psi}$ essentially at 0. If $\tilde{\psi}$ is below 0.0003 radians, the linear regression in node 2 will be used. Otherwise, the linear regression of node 3 will be used. The coefficients of the linear regression in the two leaf nodes can be seen in Table 4.3. By investigating the vessel states for this scenario in Figure 4.11, it is seen that $\tilde{\psi}$ is always negative. This means f_3 for this scenario is controlled solely by the linear regression in node 2.

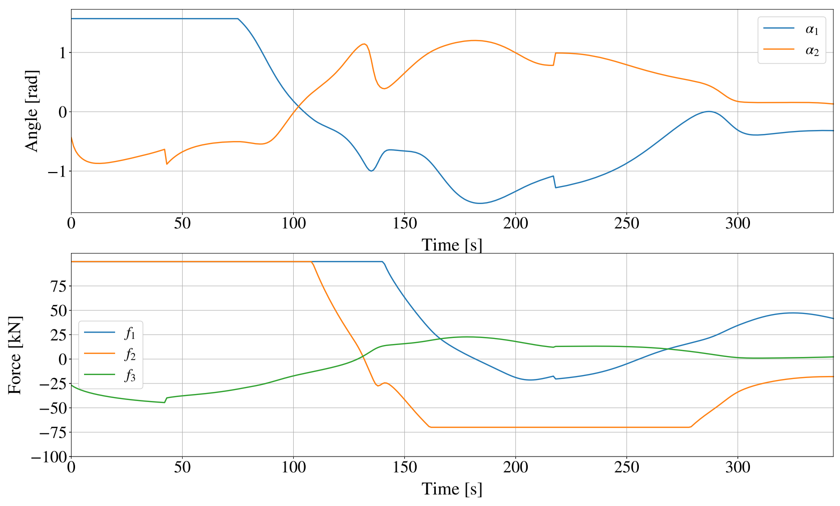


Figure 4.10: Vessel actions of a failed docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 1.

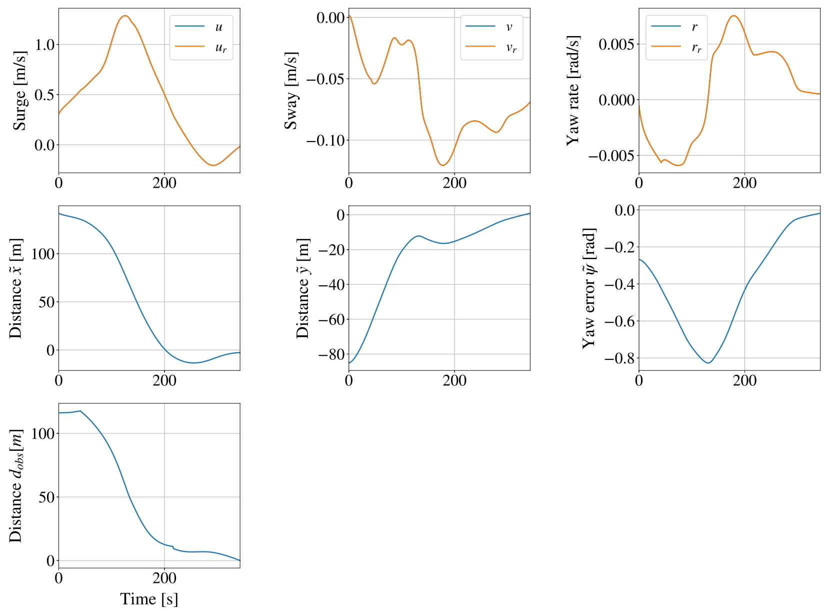


Figure 4.11: Vessel states of a failed docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 1.

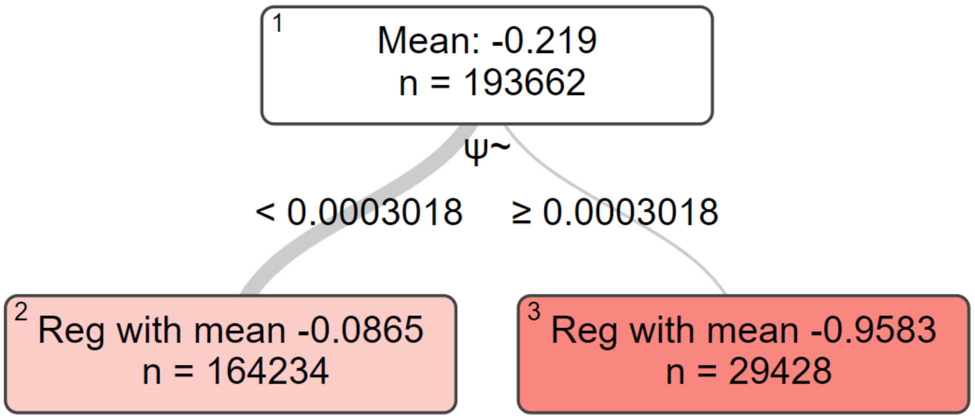


Figure 4.12: Optimal Regression Tree of depth 1.

State	Node 2	Node 3
$\tilde{x}$	0.0006596	-0.00009879
$\tilde{y}$	0.001748	-0.00006366
l	0	0
u	0.009009	-0.02378
v	1.673	-0.1146
r	61.72	12.71
d_{obs}	-0.005271	-0.0001507
$\tilde{\psi}_{obs}$	-0.03306	-0.009885
$\tilde{\psi}$	-0.3044	-0.1387
Constant	0.1124	-0.9055

Table 4.3: Coefficients of the linear regressions in node 1 and 2 in Figure 4.12.

To give context to the coefficients of the linear regressions, the feature importances for each feature is calculated at every time step depending on which leaf the data sample falls into through (2.29). Figure 4.13 illustrates the feature importances over the episode until it collided with the quay. As it should, the tunnel thruster should be quite concerned about the rotational force r, as it is the state it has the most influence over. When comparing the states of the PPO agent and this tree however, it is clear that the state with the largest deviations was the sway velocity v. Comparing the feature importances of this tree in Figure 4.13 to the feature importances of the LMT in Figure 4.7, this tree is placing way more importance on the state v than the LMT during the same time period. The LMT placed more importance on the the distance d_{obs} for f_3. This ultimately caused the vessel to collide with the harbor.

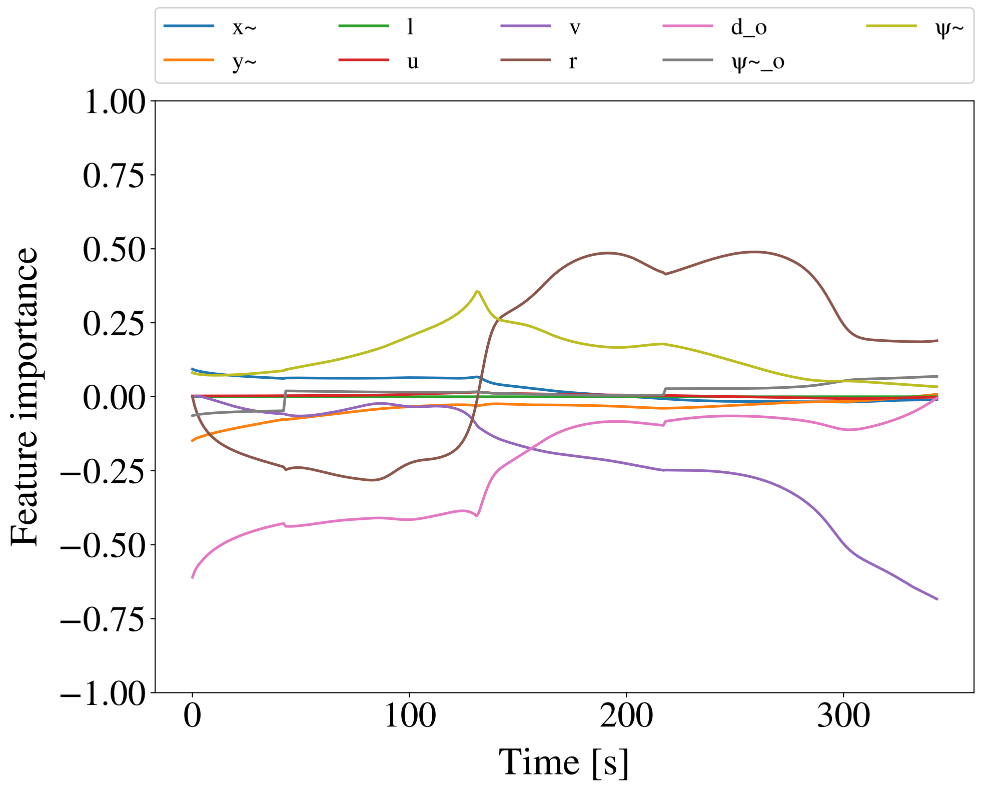


Figure 4.13: Feature importances of a failed docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 1.

ORT with depth 8

This tree managed to successfully solve the docking scenario in around 400 seconds, similarly to when using only the PPO-agent. The initial pose is the same as in Section 4.1. The trajectory of the vessel in this episode can be seen in Figure 4.14, the states in Figure 4.16, and the actions in Figure 4.15. The tunnel thruster is observed to almost perfectly replicate the tunnel thruster of the PPO agent. In addition, as will be seen in Table 4.4, the failure rate of this tree is almost exactly the same as the PPO agent.

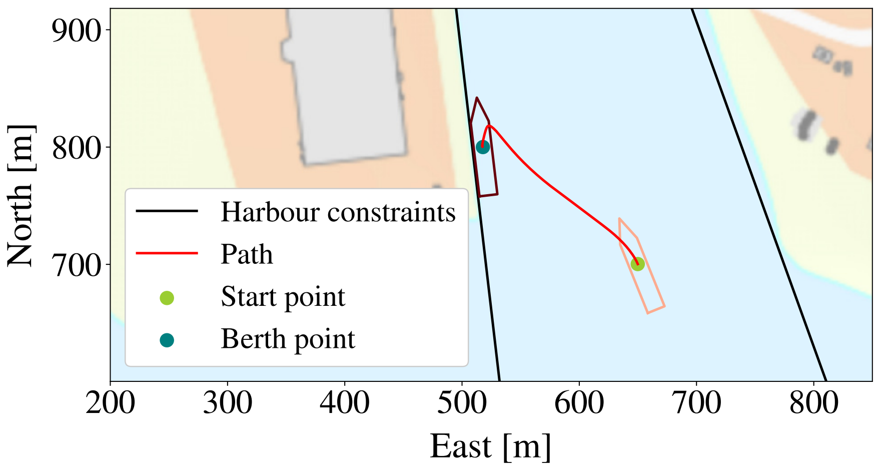


Figure 4.14: Vessel trajectory of a successful docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 8.

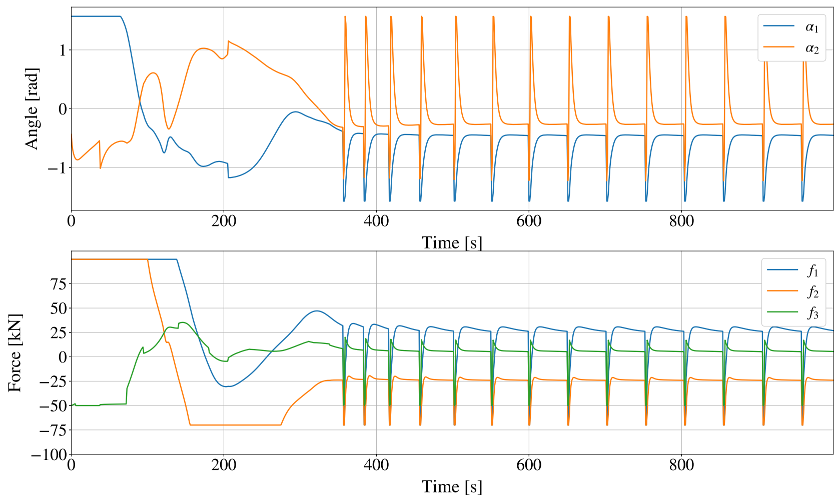


Figure 4.15: Vessel actions of a successful docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 8.

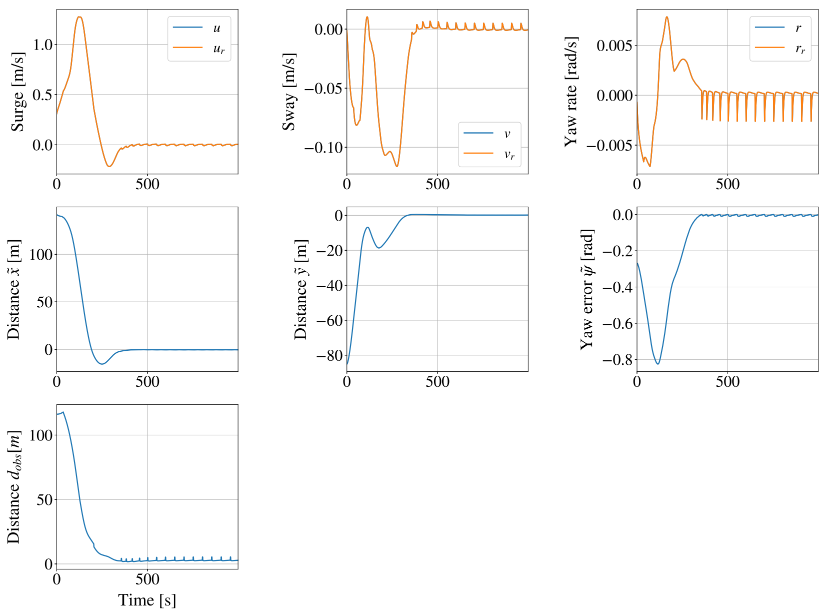


Figure 4.16: Vessel states of a successful docking scenario when replacing the tunnel thruster of the PPO-agent with an Optimal Regression Tree of depth 8.

Figure 4.17 shows the calculated feature importances from an ORT of depth 8 controlling the tunnel thruster. At the beginning of the approach phase, the vessel immediately turns counter-clockwise before it turns clockwise again to align itself with the harbor, as can be seen from the state $\tilde{\psi}$ in Figure 4.16. As expected, during this first counter-clockwise turn, most importance is attributed to the states r, d_{obs} and $\tilde{\psi}$. These are the states the tunnel thruster has the most control over. The tunnel thruster is not as effective at high speeds on open sea as it is to make minor adjustments to the vessel pose. When the vessel reaches its berthing pose, it is oscillating in yaw rate $\tilde{\psi}$ as if behaving like an underdampened PID-controller. During these oscillations, the related states r, $\tilde{\psi}_{obs}$ and d_{obs} contribute most to the tunnel thruster output.

It is noted that right around the time when the ORT with D=1 crashed, the feature importances for v and r suddenly switch and become positive. This is a sign that the tree learned another split to prevent the vessel from crashing in scenarios as was seen in the previous section.

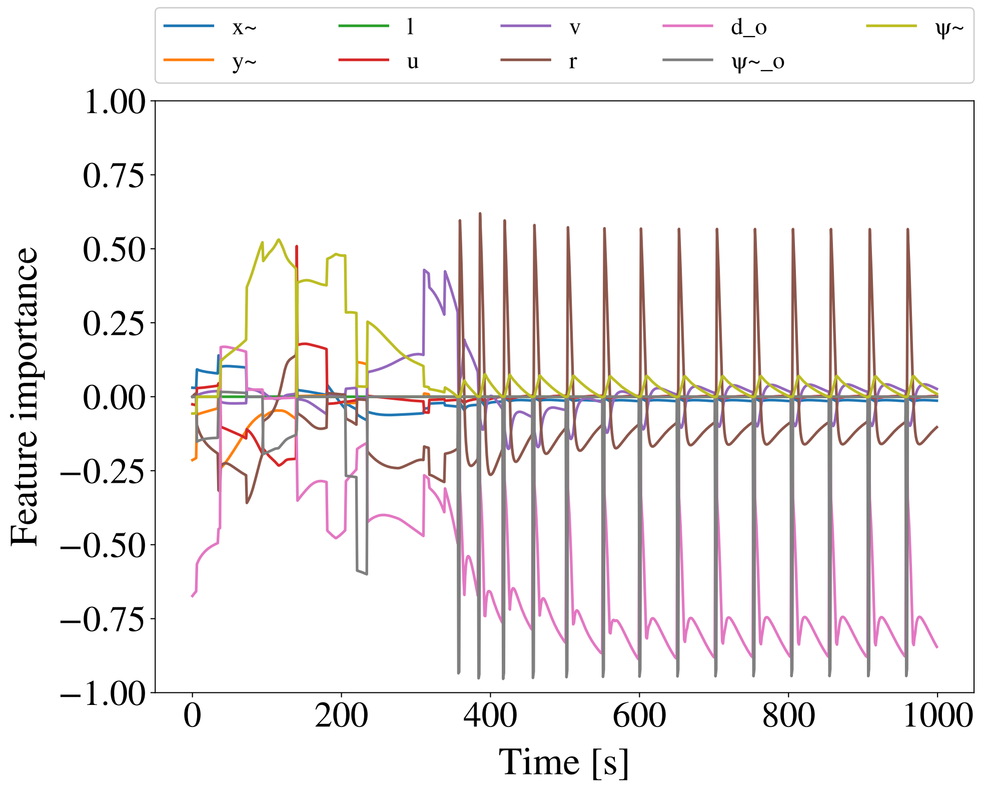


Figure 4.17: Feature importance values for an ORT of depth 8 trained to approximate the tunnel thruster

Summary

Table 4.4 illustrates the evaluation metrics on the test set, and the failure rate of the tree over 100 episodes. Given the same initial poses as the scenario in Section 4.1, it was observed that an ORT trained to control the tunnel thruster together with the PPO agent was never able to solve the docking scenario for depths of 1. For every depth added, the failure rate decreased until depth 8. When the depth of the tree increased to 9, a significant increase in failure rate of the agent is observed. Even though MAE decreased, the MSE increased, while R^2 decreased. MSE is often an indicator of outliers in the data, and this increase may demonstrate that the tree overfitted to outliers.

When comparing the ORT with depth 8 trained to control f_3 to the LMT of the same action, the ORT outperformed the LMT in terms of MAE at depth 3. The MSE was consistently lower for the ORTs than the LMT for all depths, and the R^2 was consistently higher. This illustrates that isolated analysis of these metrics alone do not necessarily give a definite answer to which tree is the best, and should be used in conjunction with for example the failure rate of the tree. In the end, the ORT with depth 8 had the highest R^2, the lowest MSE, and the lowest failure rate. It did however have a slightly higher MAE than the trees of depth 9 and 10.

Depth	MAE	MSE	Out-of-sample R^2	Failure rate
1	0.1216	0.0347	0.8742	100 %
2	0.1044	0.0235	0.9151	48 %
3	0.0816	0.0151	0.9454	37 %
4	0.0641	0.0092	0.9667	16 %
5	0.0555	0.0089	0.9677	10 %
6	0.0428	0.0056	0.9798	10 %
7	0.0378	0.0057	0.9794	9 %
8	0.0325	0.0042	0.9849	3 %
9	0.0312	0.0061	0.9778	9 %
10	0.0314	0.0088	0.9680	3 %

Table 4.4: Evaluation metrics for the best ORT-L for a given depth when approximating the force f_3.

4.3.3 Approximating the other actions

This section will explore approximating the rest of the actions, and will attempt to dock the vessel by replacing all the actions with ORTs.

Evaluation metrics

It is noted that f_3 had much higher rates of failure at lower depths than f_1 and f_2. This illustrates how important f_3 is, especially during the final phase of docking.

Interestingly, even though the R^2 for f_2 and especially a_2 were quite low at depth 1, it is observed that the failure rate is very low when compared to the other actions at similar depth. This may indicate that the second azimuth thruster is not as extensively used as the two other thrusters. This same phenomenon can be seen in Section 4.2 where the failure rate was lower when replacing the neural network with an LMT for f_2 and a_2 compared to the other actions.

Depth	MAE	MSE	Out-of-sample R^2	Failure rate
1	0.1861	0.0739	0.8509	30 %
2	0.1361	0.0392	0.9209	9 %
3	0.0889	0.0183	0.9631	6 %
4	0.0667	0.0143	0.9711	4 %
5	0.0494	0.0078	0.9843	9 %
6	0.0395	0.0069	0.9861	3 %
7	0.0320	0.0074	0.9850	4 %
8	0.0292	0.0056	0.9887	2 %
9	0.0265	0.0065	0.9869	9 %
10	0.0244	0.0068	0.9863	9 %

Table 4.5: Evaluation metrics for the best ORT for a given depth when approximating the force f_1.

Depth	MAE	MSE	Out-of-sample R^2	Failure rate
1	0.2308	0.0919	0.7923	7 %
2	0.1440	0.0460	0.8960	10 %
3	0.1135	0.0300	0.9322	10 %
4	0.0900	0.0210	0.9526	8 %
5	0.0680	0.0138	0.9690	7 %
6	0.0574	0.0115	0.9740	10 %
7	0.0426	0.0082	0.9816	6 %
8	0.0414	0.0634	0.8567	2 %
9	0.0355	0.0113	0.9744	6 %
10	0.0338	0.0081	0.9817	8 %

Table 4.6: Evaluation metrics for the best ORT for a given depth when approximating the force f_2.

Depth	MAE	MSE	Out-of-sample R^2	Failure rate
1	0.1863	0.0676	0.6673	56 %
2	0.1636	0.0528	0.7404	57 %
3	0.1278	0.0377	0.8144	24 %
4	0.1082	0.0280	0.8626	11 %
5	0.0849	0.0204	0.9000	7 %
6	0.0711	0.0160	0.9222	10 %
7	0.0578	0.0113	0.9442	8 %
8	0.0546	0.0139	0.9318	8 %
9	0.0423	0.0074	0.9637	6 %
10	0.0450	0.0114	0.9438	12 %

Table 4.7: Evaluation metrics for the best ORT for a given depth when approximating the azimuth angle a_1.

Depth	MAE	MSE	Out-of-sample R^2	Failure rate
1	0.2513	0.1080	0.4771	20 %
2	0.2007	0.0782	0.6214	15 %
3	0.1606	0.0547	0.7350	18 %
4	0.1163	0.0355	0.8282	9 %
5	0.0897	0.0194	0.9060	6 %
6	0.0767	0.0154	0.9255	10 %
7	0.0694	0.0195	0.9054	7 %
8	0.0605	0.0113	0.9449	10 %
9	0.0501	0.0097	0.9532	9 %
10	0.0479	0.0115	0.9444	4 %

Table 4.8: Evaluation metrics for the best ORT for a given depth when approximating the azimuth angle a_2.

Replacing the PPO agent with five ORTs

Finally, the entire PPO agent will be replaced by five trees. The trees with the chosen depths were based on how well each tree performed individually in the previous sections. The depth of the trees which were chosen to replace the PPO agent are shown in Table 4.9.

Action	Depth of tree
f_1	8
f_2	10
f_3	8
a_1	9
a_2	10

Table 4.9: ORTs chosen to replace the PPO agent

The five trees that replaced the PPO agent was tested on an entire episode with the same initial pose as in Section 4.1. The resulting trajectory of this episode is plotted in Figure 4.18. The actions can be seen in Figure 4.19, and the states in Figure 4.20. There are quite a lot of oscillations for when observing the actions a_2 and f_1. This same behavior was observed for the LMT in Figure 4.5. This indicates that the tree is perhaps switching between two different leaf nodes. This behavior may be the result of the trees controlling these two actions overfitting slightly to the training data, or the splitting point for a certain state may be slightly off.

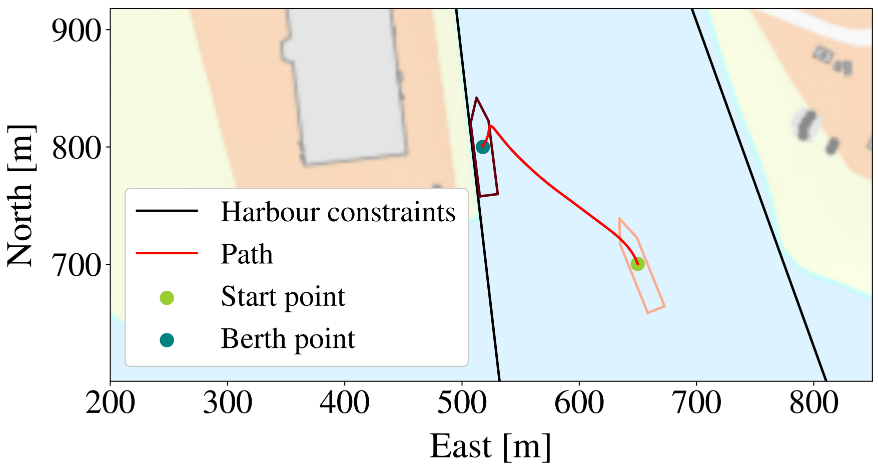


Figure 4.18: Vessel trajectory of a successful docking scenario when replacing the PPO-agent with five ORTs

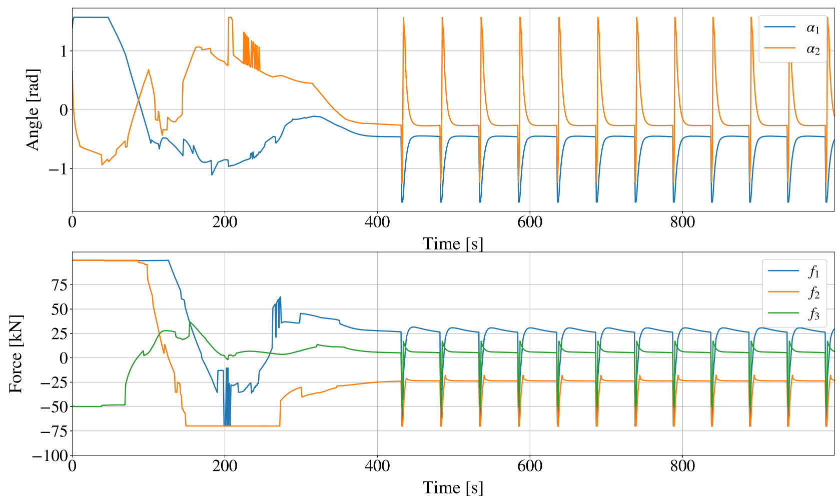


Figure 4.19: Vessel actions of a successful docking scenario when replacing the PPO-agent with five ORTs

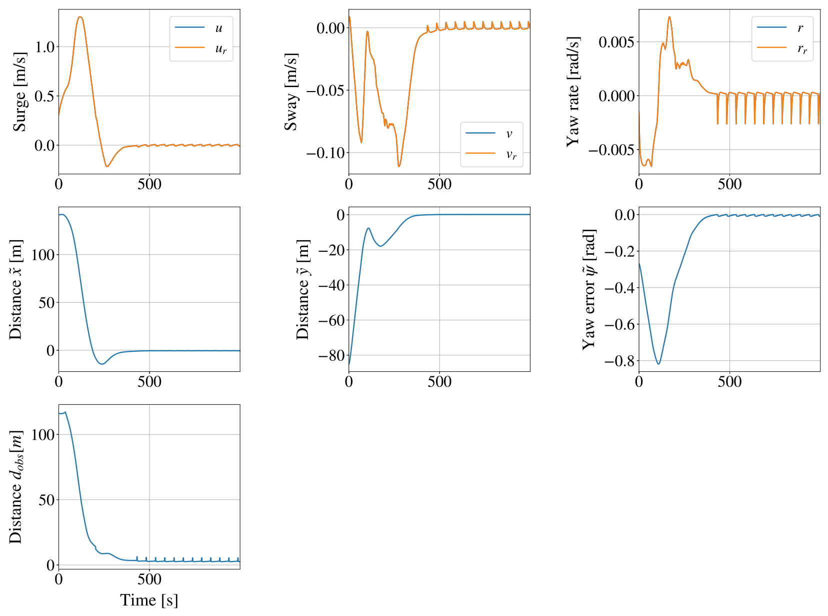


Figure 4.20: Vessel states of a successful docking scenario when replacing the PPO-agent with five ORTs

The feature importances for all actions are plotted in Figure 4.21. Notice the gaps in importance, which are especially prominent in f_2. This occurs because the tree has learned to output constant predictions at these points. Therefore, (2.29) does not apply anymore, and no definite answer to feature importance is found in these points.

Comparing the feature importances to those of the LMT in Figure 4.7, many of the same tendencies can be observed. For example, between 200 and 400 seconds into the episode, both the LMT and the ORTs attribute a lot of importance to the state d_{obs} for the action f_3. However, there are several states that are more prominent in the ORTs. For example, when observing feature importances for f_2, the LMT is only concerned about the state u until it reaches the berth. The ORT uses this state much more extensively during the oscillatory period after 400 seconds. The ORTs also seem to make more use of the state r than the LMT.

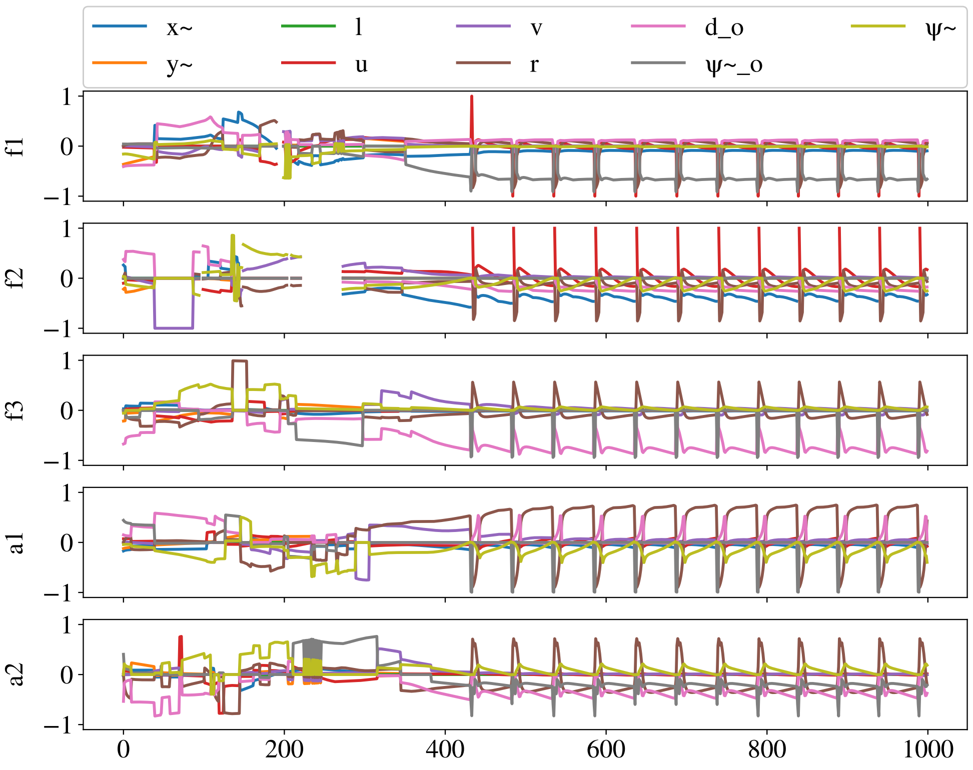


Figure 4.21: Feature importance values for all ORTs trained to approximate the PPO agent

4.4 NNRT

The NNRT method did not perform adequately in our tests. The implementation proved to be quite slow. Efforts were made to speed up the implementation by incorporating early stopping and mini-batches. In addition, the model was also attempted to run on the CUDA cores of the workstation GPU. This ultimately ended up slowing down the implementation, which is not surprising in hindsight. The sequential nature of the NNRT implementation is not suitable for parallel processing. The method may also be quite sensitive to hyperparameter tuning. Along with the parameters that needed to be set for Algorithm 4, the Adam optimizer needs to be initialized with a learning rate α. In the beginning of the training loop, the implementation was consistently improving the loss. As the loop continued however, the validation loss started oscillating and did not converge. This can be seen in Figure 4.22. However, the best R^2 observed was 0.7547, and the best MAE observed was 0.1476. This is not statistically insignificant. In addition, the ORT observed in Section 4.3.2 had a similar MAE. It is therefore concluded that the method did work to some extent, and that further work should be done to improve the implementation of this method.

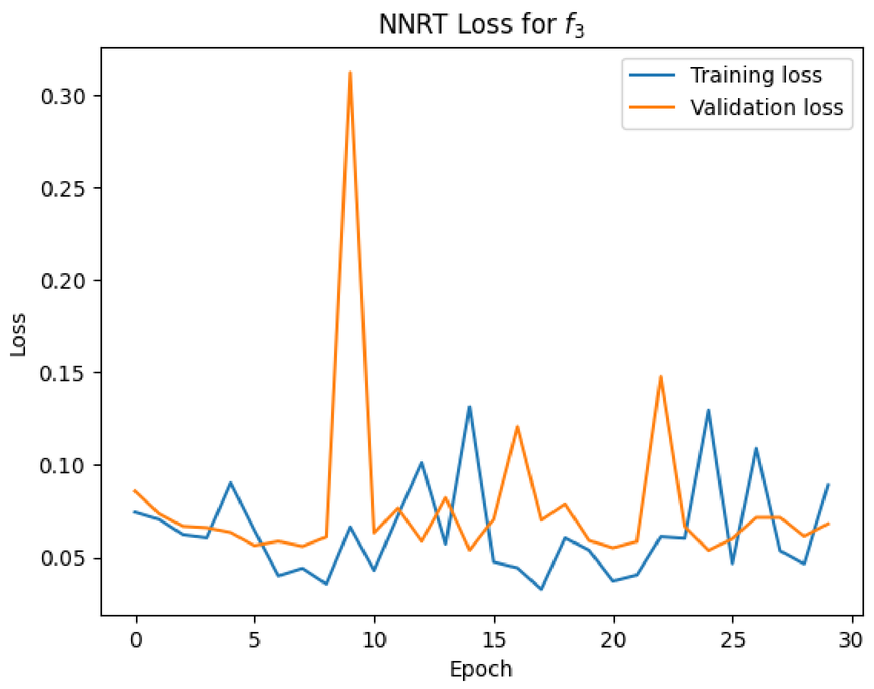


Figure 4.22: Training and validation loss when training NNRT with depth 4 for action f_3.

The dataset is very large, and the authors of the NNRT paper also noted that the NNRT was slower than for example `XGBoost` when the datasets were large [67]. From Figure 4.22, it is observed that the loss for both the training set and validation set initially decreases rapidly along with the MAE and MSE, and the R^2 increases rapidly. However, after a few epochs, the loss oscillates and does not converge. The issues with this method may lie in the implementation. The implementation was created from scratch with limited experience with decision trees. It may also be the result of some unknown parameters not presented in the original paper. There is little reason to believe there is something fundamentally wrong in the data presented to the method, as the method previously presented had great results. The fact that the loss and errors decrease and accuracy increases for the first few epochs indicates. It may however be plausible the method is not suitable for this kind of data. The method was tested on the *yacht hydrodynamics* dataset [80] and achieved similar performance to that presented in the original paper on the same dataset.

4.5 Summary of methods

The best performing ORTs along with the LMT were tested on 100 episodes. Table 4.10 summarizes the evaluation metrics for these 100 episodes when compared to the PPO agent.

Method	MAE	MSE	R^2	Failure rate
PPO	N/A	N/A	N/A	2%
LMT	0.1187	0.0473	0.7703	12%
ORT	0.0427	0.0139	0.9243	10%

Table 4.10: Quantitative test results obtained from 100 episode simulations in each scenario.

It is clear from this experiment that the five ORTs performed better than the single LMT across all metrics. It is notable that the results for the LMTs were substantially worse than outlined in [12]. This might be because the dataset created in this thesis may capture some dynamics that were not emphasized in the dataset the LMTs were trained on. The reduced MAE and MSE, and the increased R^2 for the ORTs signifies that this method may have the potential to generalize better. More care should be taken when generating the dataset for these methods. Creating a new dataset with less starting points and longer episodes may remedy this gap in performance between the methods. This might capture more dynamics at the berth, which may be more complex than those at open sea.

Chapter 5

Conclusion

The goal of this thesis was to explore whether inherently interpretable tree structures such as regression trees are able to approximate a deep reinforcement learning agent's neural network, and whether this could help to increase interpretability of black-box models.

We conclude that it is possible to build regression trees that sufficiently approximate a deep reinforcement learning agent. Through extensive quantitative analysis, we have shown that optimal regression trees with linear predictions in the leaf nodes are able to approximate the neural network of the DRL agent with higher accuracy than the linear model trees, and were able to successfully dock. Though one tree is generated per action, this may not be entirely undesirable. Shallower trees are easier to inspect, and it allows for isolated analysis of the approximation of each action. If one action has significantly more complex dynamics, it is undesirable and wasteful to generate one large tree to accommodate for that one action. Instead, several smaller trees can be created for the actions that are shallower. The accuracy of this method could very likely be increased further by tuning the regularization parameters better.

Though a lot of the effort went into making the NNRTs work for the docking problem, the method did not work as expected. There may be numerous reasons why this method did not succeed. The implementation may simply not be true to the implementation presented in the original paper. The optimizer requires more parameters to be set than was presented in the NNRT paper, which may be set to wrong values in our implementation. Alternatively, the method may just not work well for this kind of data. Despite the NNRTs either not being properly implemented or just not appropriate for this kind of problem, the optimal regression trees worked. ORT has both simpler splitting criterion because of univariate splits, and has simpler prediction functions in the leaf nodes. We also note that feature attributions may be harder to compute for a method with nonlinear prediction functions like NNRTs.

5.1 Future Work

Recent developments in symbolic regression such as AI Feynman 2.0 has the potential to give a definite answer to five functions that fit the output of the reinforcement learning agent. Methods such as AI Feynman should be explored in further detail. The method is described in more detail in Appendix A.

Ideally, better heuristics for producing the dataset should also be investigated. Constructing the dataset proved to be a challenge which may have affected the results when comparing LMT with ORT. Currently, the dataset is constructed by letting the PPO agent dock, and recording its actions over a set period of time. Instead, the actions during each episode should be recorded until the vessel is sufficiently close to the berthing pose. The episode should end the episode when some criterion for a successful docking is met. This is highly likely to result in a dataset that captures more dynamics than when setting a time limit.

One may also want to investigate other formulas for feature importance. The current formula does not generalize to nonlinear predictions, nor constant predictions.

Bibliography

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[3] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: `10.1038/s41586-020-2649-2`.

[4] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. [Online]. Available: `https://doi.org/10.5281/zenodo.3509134`.

[5] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 56–61. DOI: `10.25080/Majora-92bf1922-00a`.

[6] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: `10.1109/MCSE.2007.55`.

[7] E.-L. H. Rørvik, "Automatic Docking of an Autonomous Surface Vessel," eng, *Master thesis. Norwegian University of Science and Technology (NTNU)*, 2020.

[8] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[10] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

[11] L. Interpretable AI, *Interpretable ai documentation*, 2020. [Online]. Available: `https://www.interpretable.ai`.

[12] V. Gjærum, E.-L. H. Rørvik, and A. M. Lekkas, "Approximating a deep reinforcement learning docking agent using linear model trees," *Submitted to IEEE European Control Conference (ECC), 2021*,

[13] J. Løver, "Explaining a deep reinforcement learning agent with post-hoc explainers," *Project Thesis. Department of Engineering Cybernetics. Norwegian University of Science and Technology (NTNU)*, 2021.

[14] C. L. Benson, P. D. Sumanth, and A. P. Colling, "A quantitative analysis of possible futures of autonomous transport," 2018. arXiv: `1806.01696`.

[15] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, Apr. 2020, ISSN: 1556-4967. DOI: `10.1002/rob.21918`.

[16] E. Santana and G. Hotz, *Learning a driving simulator*, 2016. arXiv: `1608.01230 [cs.LG]`.

[17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017. arXiv: `1712.01815`.

[18] *Art. 22 GDPR – Automated individual decision-making, including profiling | General Data Protection Regulation (GDPR)*, Jul. 2018. [Online]. Available: `https://gdpr-info.eu/art-22-gdpr`.

[19] M. Edmonds, F. Gao, H. Liu, X. Xie, S. Qi, B. Rothrock, Y. Zhu, Y. N. Wu, H. Lu, and S.-C. Zhu, "A tale of two explanations: Enhancing human trust by explaining robot behavior," *Science Robotics*, vol. 4, no. 37, 2019.

[20] *EU's Right to Explanation: A Harmful Restriction on Artificial Intelligence*, Jul. 2021. [Online]. Available: `https://www.techzone360.com/topics/techzone/articles/2017/01/25/429101-eus-right-explanation-harmful-restriction-artificial-intelligence.htm`.

[21] O. Buffet, O. Pietquin, and P. Weng, *Reinforcement learning*, 2020. arXiv: `2005.14419`.

[22] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is enough," *Artificial Intelligence*, vol. 299, p. 103 535, 2021, ISSN: 0004-3702. DOI: `10.1016/j.artint.2021.103535`.

[23] M. Turek, *Explainable Artificial Intelligence*. [Online]. Available: `https://www.darpa.mil/program/explainable-artificial-intelligence`.

[24] A. Heuillet, F. Couthouis, and N. Díaz-Rodríguez, *Explainability in deep reinforcement learning*, 2020. arXiv: `2008.06693`.

[25] J. Løver, V. B. Gjærum, and A. M. Lekkas, "Explainable AI methods on a deep reinforcement learning agent for automatic docking," *IFAC-PapersOnLine*, 2021, In press.

[26] S. Lundberg, G. Erion, H. Chen, A. DeGrave, J. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee, "Explainable ai for trees: From local explanations to global understanding," May 2019.

[27] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," 2019. arXiv: `1811.10154`.

[28] Z. C. Lipton, "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.," *Queue*, vol. 16, no. 3, pp. 31–57, Jun. 2018, ISSN: 1542-7730. DOI: `10.1145/3236386.3241340`.

[29] I. E. Kumar, S. Venkatasubramanian, C. Scheidegger, and S. Friedler, "Problems with shapley-value-based explanations as feature importance measures," 2020.

[30] D. Slack, S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju, *Fooling lime and shap: Adversarial attacks on post hoc explanation methods*, 2020.

[31] H. Luo, F. Cheng, H. Yu, and Y. Yi, "Sdtr: Soft decision tree regressor for tabular data," *IEEE Access*, vol. PP, pp. 1–1, Apr. 2021. DOI: `10.1109/ACCESS.2021.3070575`.

[32] O. Loyola-González, "Black-box vs. white-box: Understanding their advantages and weaknesses from a practical point of view," *IEEE Access*, vol. 7, pp. 154 096–154 113, 2019. DOI: `10.1109/ACCESS.2019.2949286`.

[33] "State of data science and machine learning 2020," *Kaggle*, [Online]. Available: https://www.kaggle.com/kaggle-survey-2020.

[34] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[35] Z. Ding, P. Hernandez-Leal, G. W. Ding, C. Li, and R. Huang, "Cdt: Cascading decision trees for explainable reinforcement learning," 2021. arXiv: 2011.07553.

[36] N. Dahlin, K. C. Kalagarla, N. Naik, R. Jain, and P. Nuzzo, "Designing interpretable approximations to deep reinforcement learning," 2021. arXiv: 2010.14785.

[37] D. Bertsimas, J. Dunn, G. C. Velmahos, and H. M. A. Kaafarani, "Surgical Risk Is Not Linear: Derivation and Validation of a Novel, User-friendly, and Machine-learning-based Predictive OpTimal Trees in Emergency Surgery Risk (POTTER) Calculator," *Ann. Surg.*, vol. 268, no. 4, pp. 574–583, Oct. 2018, ISSN: 1528-1140.

[38] IBM Cloud Education, *What is Machine Learning?* en-us. [Online]. Available: https://www.ibm.com/cloud/learn/machine-learning (visited on 01/23/2021).

[39] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *CoRR*, vol. abs/1811.03378, 2018.

[40] G. Ognjanovski, *Everything you need to know about Neural Networks and Backpropagation — Machine Learning Made Easy…* en, Jun. 2020. [Online]. Available: https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

[41] *Perceptrons: The First Neural Networks*, Jun. 2021. [Online]. Available: https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks.

[42] Machine Learning for Artists, *How neural networks are trained*. [Online]. Available: https://ml4a.github.io/ml4a/how_neural_networks_are_trained/.

[43] S. Badillo, B. Banfai, F. Birzele, I. Davydov, L. Hutchinson, T. Kam-Thong, J. Siebourg-Polster, B. Steiert, and J. D. Zhang, "An introduction to machine learning," *Clinical Pharmacology & Therapeutics*, vol. 107, Mar. 2020. DOI: 10.1002/cpt.1796.

[44] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, pp. 1–124, May 2019, ISSN: 0370-1573. DOI: 10.1016/j.physrep.2019.03.001.

[45] S. Linnainmaa, “Taylor expansion of the accumulated rounding error,” *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, Jun. 1976, ISSN: 1572-9125.

[46] G. Goh, “Why Momentum Really Works,” *Distill*, vol. 2, no. 4, e6, Apr. 2017, ISSN: 2476-0757. DOI: 10.23915/distill.00006.

[47] R. Amiri, H. Mehrpouyan, L. Fridman, R. K. Mallik, A. Nallanathan, and D. Matolak, “A machine learning approach for power allocation in hetnets considering qos,” in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–7.

[48] T. Lozano-Pérez and L. Kaelbling, *Markov Decision Processes | MIT OCW*, en. [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-825-techniques-in-artificial-intelligence-sma-5504-fall-2002/lecture-notes/.

[49] OpenAI, *Proximal Policy Optimization — Spinning Up documentation*. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[50] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu, “Definitions, methods, and applications in interpretable machine learning,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 44, pp. 22 071–22 080, Oct. 2019, ISSN: 1091-6490. DOI: 10.1073/pnas.1900654116.

[51] A. Holzinger, C. Biemann, C. S. Pattichis, and D. B. Kell, “What do we need to build explainable ai systems for the medical domain?,” 2017. arXiv: 1712.09923.

[52] T. Miller, *Explanation in artificial intelligence: Insights from the social sciences*, 2018.

[53] C. Molnar, *Interpretable Machine Learning*. [Online]. Available: https://christophm.github.io/interpretable-ml-book/.

[54] M. T. Ribeiro, S. Singh, and C. Guestrin, “"why should I trust you?": Explaining the predictions of any classifier,” *CoRR*, 2016.

[55] M. T. Ribeiro, S. Singh, and C. Guestrin, *Anchors: High-precision model-agnostic explanations*, 2018.

[56] H. P. Young, “Monotonic solutions of cooperative games,” en, *International Journal of Game Theory*, vol. 14, no. 2, pp. 65–72, Jun. 1985, ISSN: 1432-1270.

[57] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 4765–4774.

[58] S. Ross, G. J. Gordon, and J. A. Bagnell, “A reduction of imitation learning and structured prediction to no-regret online learning,” 2011. arXiv: 1011.0686.

[59] D. S. Chaturvedi and S. Patil, "Oblique decision tree learning approaches - a critical review," *International Journal of Computer Applications*, vol. 82, pp. 6–10, Nov. 2013. DOI: 10.5120/14174-2023.

[60] B.-B. Yang, S.-Q. Shen, and W. Gao, "Weighted oblique decision trees," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 5621–5627, Jul. 2019. DOI: 10.1609/aaai.v33i01.33015621.

[61] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017, ISBN: 978-1491962299.

[62] N. Kishore, "Decision Trees, Random forests and PCA - Nitin Kishore - Medium," *Medium*, Sep. 2018, ISSN: 6764-1426. [Online]. Available: https://medium.com/@snk.nitin/decision-trees-random-forests-and-pca-e676e4c142c6.

[63] L. Torgo, "Regression trees," in *Encyclopedia of Machine Learning and Data Mining*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2017, pp. 1080–1083, ISBN: 978-1-4899-7687-1. DOI: 10.1007/978-1-4899-7687-1_717.

[64] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, ser. Springer series in statistics. Springer, 2009, pp. 307–308.

[65] D. Bertsimas and J. Dunn, *Machine learning under a modern optimization lens*. Dynamic Ideas LLC, 2019.

[66] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *Journal of Statistical Software*, vol. 33, no. 1, pp. 1–22, 2010.

[67] D. Bertsimas, J. Dunn, and Y. Wang, "Near-optimal nonlinear regression trees," *Operations Research Letters*, vol. 49, no. 2, pp. 201–206, 2021, ISSN: 0167-6377.

[68] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. arXiv: 1412.6980.

[69] V. L. Tran and N.-K. Im, "A study on ship automatic berthing with assistance of auxiliary devices," en, *International Journal of Naval Architecture and Ocean Engineering*, vol. 4, no. 3, pp. 199–210, Sep. 2012, ISSN: 2092-6782.

[70] C. Li, X. Yan, S. Li, J. Liu, F. Ma, *et al.*, "Survey on ship autonomous docking methods: Current status and future aspects," in *The 30th International Ocean and Polar Engineering Conference*, International Society of Offshore and Polar Engineers, 2020.

[71] Y. A. Ahmed and K. Hasegawa, "Automatic ship berthing using artificial neural network trained by consistent teaching data using nonlinear programming method," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 10, pp. 2287–2304, 2013, ISSN: 0952-1976.

[72] N.-K. Im and V.-S. Nguyen, "Artificial neural network controller for automatic ship berthing using head-up coordinate system," *International Journal of Naval Architecture and Ocean Engineering*, vol. 10, no. 3, pp. 235–249, 2018, ISSN: 2092-6782.

[73] V. Nguyen, "Investigation of a multitasking system for automatic ship berthing in marine practice based on an integrated neural controller," vol. 8, pp. 1–23, Jul. 2020.

[74] E. Anderlini, G. Parker, and G. Thomas, "Docking control of an autonomous underwater vehicle using reinforcement learning," *Applied Sciences*, vol. 9, p. 3456, Aug. 2019.

[75] Y. Zhang, M. Zhang, and Q. Zhang, "Auto-berthing control of marine surface vehicle based on concise backstepping," *IEEE Access*, vol. 8, pp. 197 059–197 067, 2020.

[76] A. B. Martinsen, A. M. Lekkas, and S. Gros, "Autonomous docking using direct optimal control," 2019.

[77] E. Meyer, H. Robinson, A. Rasheed, and O. San, "Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning," *IEEE Access*, vol. 8, pp. 41 466–41 481, 2020.

[78] A. Wong, *Ankonzoid/LearningX*, en. [Online]. Available: `https://github.com/ankonzoid/LearningX`.

[79] xuyxu, *Soft-Decision-Tree*, Jul. 2021. [Online]. Available: `https://github.com/xuyxu/Soft-Decision-Tree`.

[80] *UCI Machine Learning Repository: Yacht Hydrodynamics Data Set*, Jul. 2021. [Online]. Available: `https://archive.ics.uci.edu/ml/datasets/yacht+hydrodynamics`.

[81] S.-M. Udrescu and M. Tegmark, "Ai feynman: A physics-inspired method for symbolic regression," *Science Advances*, vol. 6, no. 16, 2020. DOI: `10.1126/sciadv.aay2631`. eprint: `https://advances.sciencemag.org/content/6/16/eaay2631.full.pdf`.

[82] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark, *Ai feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity*, Jun. 2020.

[83] *Planetary Motion: The History of an Idea That Launched the Scientific Revolution*, Jul. 2009. [Online]. Available: `https://earthobservatory.nasa.gov/features/OrbitsHistory/page2.php`.

[84] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," *CoRR*, vol. abs/1703.01365, 2017. arXiv: `1703.01365`.

[85] *Integrated gradients | TensorFlow Core*, Mar. 2021. [Online]. Available: `https://www.tensorflow.org/tutorials/interpretability/integrated_gradients`.

[86] D. Frankowski, “Should you explain your predictions with SHAP or IG?” *Medium*, Aug. 2019. [Online]. Available: https://towardsdatascience.com/should-you-explain-your-predictions-with-shap-or-ig-9cabe218b5cc.

Appendix A

AI Feynman 2.0

This section covers AI Feynman 2.0, an improvement to the original AI Feynman algorithm [81]. The equations and descriptions present in this section are from [82], and described in more detailed there. This section will summarize the method.

Johannes Kepler, assistant to astronomer Tycho Brahe, got access to Brahe's dataset of astronomical observations in order to calculate the orbit of Mars in the 1600s. Kepler noticed that the planets must move more quickly when it is near the Sun, but more slowly when it is farthest from the Sun [83]. This led to Kepler's first law: "The planets move in elliptical orbits with the Sun at one focus", an early example of symbolic regression, where a mathematical equation is found through data.

AI Feynman is a Pareto-optimal symbolic regression method. A Pareto-optimal solution is a solution that is the most accurate with a given complexity. In AI Feynman, this is described through what the authors called a Pareto-frontier. The Pareto-frontier describes the solutions in terms of inaccuracy versus *description-length complexity*, as shown in Figure A.1. Pruning candidates not on the frontier increases AI Feynman's robustness to noise and bad data [82]. Description-length complexity is described in Table 2 of [82], and is a measure of how complex the candidate is depending on whether the equation contains natural numbers, integers, etc.

The central idea behind AI Feynman revolves around the fact that any mystery function f can be expressed as a tree graph made up of elementary functions. In Figure A.2, the left tree represents a function f with three inputs x, y, z. The middle tree represents the possible decomposition of the mystery function. The right tree represents the decomposition AI Feynman seeks to find, where f is decomposed into two separate functions h and g with fewer input variables such

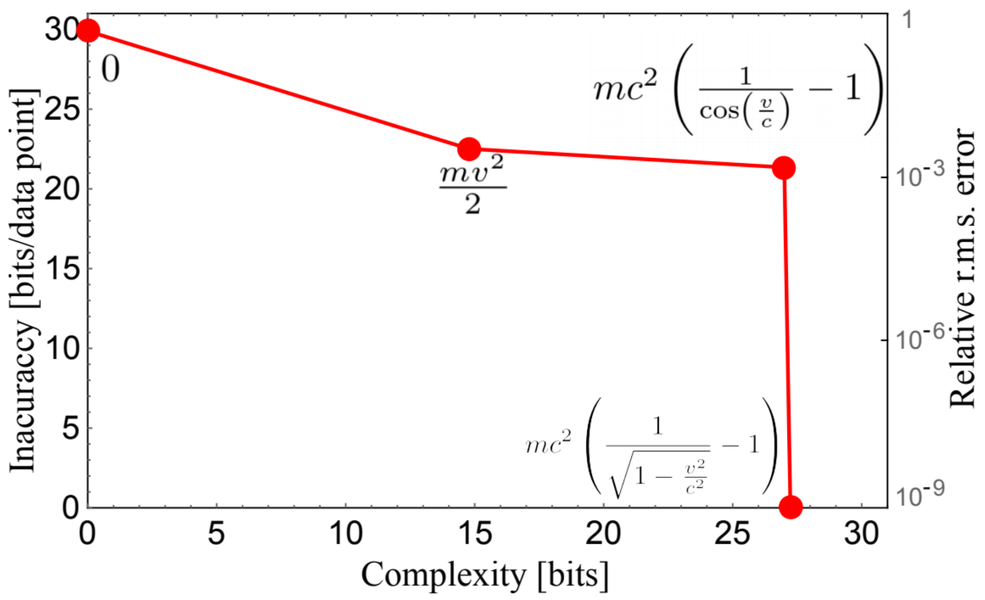
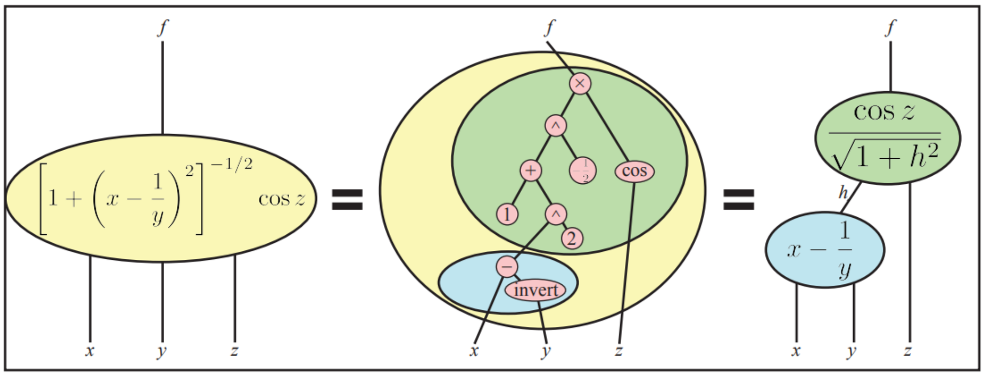


Figure A.1: Pareto-frontier discovered by AI Feynman 2.0 [82].



f
$\left[1+\left(x-\frac{1}{y}\right)^2\right]^{-1/2}\cos z$
x
y
z
invert
cos
$\frac{\cos z}{\sqrt{1+h^2}}$
h
$x-\frac{1}{y}$

Figure A.2: Graph decomposition of a mystery function f [82].

that

$$f(x,y,z) = g[h(x,y),z]. \tag{A.1}$$

The method works by training a neural network f_{NN} to approximate the unknown function f that generated the data. AI Feynman then leverages this neural network approximation to probe the unknown function for modularity. This is the magic sauce of AI Feynman, as searching through possible decompositions by brute force is in practice infeasible. More specifically, it probes the function to discover six different graph decompositions, as shown in Figure A.3.

The most significant contribution of AI Feynman is how it discovers generalized symmetry. Generalized symmetry means that k number of variables only pass through some other scalar function h, as shown in the bottom left panel of Figure A.3. AI Feynman finds these properties in f by analyzing its gradients. Again, the equations and descriptions which will be presented are from [82].

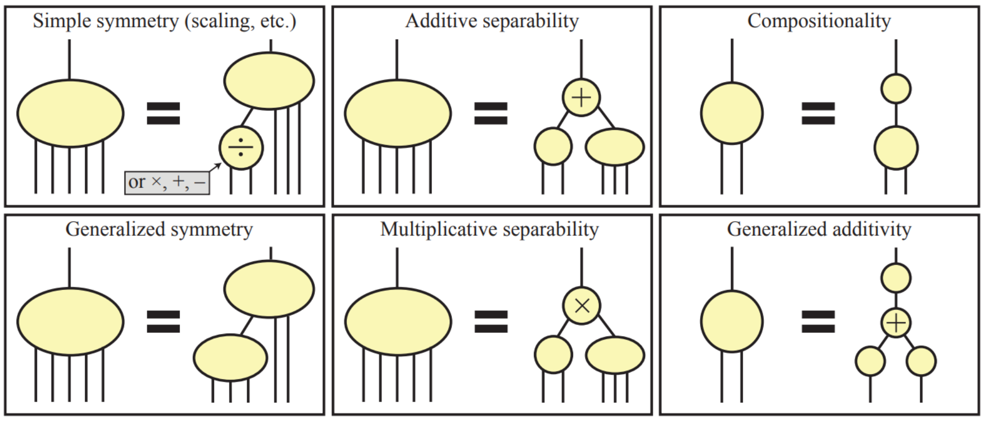


Figure A.3: Possible graph decompositions that AI Feynman can auto-discover [82].

Let the input vector $\mathbf{x} \in \mathbb{R}^n$. The input vector can then be split into two groups $\mathbf{x}' \in \mathbb{R}^k$ and $\mathbf{x}'' \in \mathbb{R}^{n-k}$. If generalized symmetry holds, similarly to equation A.1, $f(\mathbf{x})$ can then be written as

$$f(\mathbf{x}) = f(\mathbf{x}', \mathbf{x}'') = g[h(\mathbf{x}'), \mathbf{x}'']. \tag{A.2}$$

Its derivative found through the chain rule can then be written as

$$\nabla_{\mathbf{x}'} f(\mathbf{x}', \mathbf{x}'') = g_1[h(\mathbf{x}'), \mathbf{x}''] \nabla h(\mathbf{x}'), \quad \text{so} \quad \hat{\nabla_{\mathbf{x}'}} f = \pm \hat{\nabla} h \tag{A.3}$$

where g_1 is the derivative of g with respect to its first argument. These conditions are only valid when $\hat{\nabla_{\mathbf{x}'} f}(\mathbf{x}', \mathbf{x}'')$ which is the direction of the gradient of f with respect to $\mathbf{x}'$, is independent of $\mathbf{x}''$. This independence is an indication that generalized symmetry is present, and the algorithm can be applied recursively to further decompose the graph. A brute force search after this decomposition is much more feasible.

Symbolic regression is an excellent XAI tool, outputting human-interpretable equations. The authors of AI Feynman identified some pitfalls when using symbolic regression, such as overly trusting the formulas extracted from the data and applying the knowledge gained by AI Feynman to untested domains [82]. The method is not quite model-agnostic either, as it requires the basis functions to be mostly differentiable [82].

Appendix B

Integrated Gradients

We have previously discussed interpretable models and model-agnostic post-hoc methods for attributing feature importance. Integrated gradients [84] is another method that deserves a mention, and is a post-hoc method that attributes feature importance by examining the gradients. However, it is not quite model-agnostic, as it requires the model being explained to be differentiable. This assumption holds true however for any model trained using gradient-descent. It is defined as the path integral of the gradients along a straightline path the from a baseline x' to some input x [85].

When applying integrated gradients, a baseline x' needs to be established first. This represents a neutral state and should be tailored to each application, as establishing a good baseline is crucial. For tabular data, the baseline may be an instance where all features are set to zero. For image data, it may be a completely black image. Then, a linear interpolation between x and x' is created. From [85], the integrated gradients can then be defined as

$$\text{IntegratedGrads}_i(x) :: = (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} \tag{B.1}$$

where α is an interpolation constant, i represents a feature, and F is the black-box model. Finding the exact integral may be approximated using a Riemann sum approximation.

The authors of the integrated gradients method identified two axioms that should be satisfied by every attribution method [84]:

Sensitivity(a) For every input and baseline that differ in one feature but have different predictions, then the differing feature should be given a non-zero attribution.

Implementation Invariance For two networks that output the same predictions, regardless of how they are implemented, attributions should be equal.

Integrated gradients was designed with the "completeness" axiom in mind, which implies Sensitivity(a). The completeness axiom says that the attributions add up to difference between the output of F at input x and baseline x' [84]. This axiom is the equivalent of the efficiency axiom for SHAP [57] which was discussed briefly in Section 2.4.2. Several attribution methods such as DeepLIFT, a central part of the Deep SHAP algorithm which is SHAP for neural networks, violates the implementation invariance axiom.

Integrated gradients exploits the fact that we have information about the neural network structure to more efficiently attribute feature importance. In contrast, a Shapley-value based method requires computing the model output on a large number of inputs sampled from the exponentially huge subspace of all possible combinations of feature values [86]. This method may prove to provide good explanations, but was not implemented due to time limitations.

Appendix C

NNRT Implementation

```
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import math

from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score, mean_absolute_error
from sklearn import preprocessing

class NNRT(nn.Module):
    def __init__(self, input_dim=9, output_dim=1, depth=5, lamda=1e-3):
        super(NNRT, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim

        self.depth = depth
        self.lamda = lamda
        self.device = torch.device("cpu")

        self._validate_parameters()

        self.internal_node_num = 2 ** self.depth - 1
        self.leaf_node_num = 2 ** self.depth

        self.a = torch.empty(self.internal_node_num, self.input_dim,\
        requires_grad=True)
        self.b = torch.empty(self.internal_node_num, requires_grad=True)
        self.f = torch.ones(self.leaf_node_num, self.depth + 1)

    def random_initialize(self, X, Y, iterations):
        """
        Initialize a and b matrices with random numbers, then
        calculate the corresponding f matrix
        Choose the a, b, and f matrices with the lowest loss
        """
        with torch.no_grad():
            loss = math.inf
            best_loss = math.inf
            best_a = None
            best_b = None
            best_f = None
            for i in range(0, iterations):
                self.a = torch.rand(
                    self.internal_node_num,
                    self.input_dim,
                    requires_grad=True)
                self.b = torch.rand(self.internal_node_num,
                                    requires_grad=True)

                loss = self._calculate_f(X, Y)

                if loss < best_loss:
                    best_a = self.a.clone().detach()
                    best_b = self.b.clone().detach()
                    best_f = self.f.clone().detach()
                    best_loss = loss

            self.a = nn.Parameter(best_a.clone().detach())
            self.b = nn.Parameter(best_b.clone().detach())
            self.f = best_f.clone().detach()

    def _calculate_f(self, X, Y):
        """
        Update the module with updated vectors f
        """
        model = Ridge(alpha=self.lamda,fit_intercept=False)
        f = torch.empty(self.leaf_node_num, self.depth + 1)
        with torch.no_grad():

            g, rhs = self.forward(X)
            f = model.fit(rhs,Y)
            self.f = torch.tensor(f.coef_).reshape(self.leaf_node_num,
                                                   self.depth + 1)
            loss = self.loss(g, Y)

        return loss

    def forward(self, X):
        """
        Implementation on the data forwarding process for vector of samples.
        """
        batch_size = X.size()[0]
        output = torch.empty(batch_size)
        rhs = torch.empty(batch_size,self.leaf_node_num * (self.depth+1))
        for sample in range(0, batch_size):
            output[sample], rhs[sample] = self._forward(X[sample, :])
        return output,rhs

    def _forward(self, x):
        """
        Implementation on the data forwarding process for one sample.
        """

        leaf_node = 0
        _ancestors = []
        _ancestors_left = []
        _ancestors_right = []

        # Traverse the tree to find the leaf node
        begin_idx = 0
        end_idx = 0
        with torch.no_grad():
            for layer_idx in range(0, self.depth):
                if self.a[begin_idx, :] @ x < self.b[begin_idx]:
                    _ancestors.append(end_idx)
                    _ancestors_left.append(end_idx)
                    end_idx = begin_idx + 2 ** (layer_idx)
                else:
                    _ancestors.append(end_idx)
                    _ancestors_right.append(end_idx)
                    end_idx = begin_idx + 2 ** (layer_idx + 1)

                begin_idx = end_idx
                leaf_node = end_idx - self.internal_node_num  # Remap nodes

        # Solve equation 4 to obtain prediction
        g = 0
        rhs = torch.zeros(self.leaf_node_num, self.depth + 1)
        rhs[leaf_node] = torch.ones(self.depth+1)
        for layer_idx in range(0, self.depth):
            _split = 1
            for ancestor_idx in _ancestors:
                _tmp = 0
                if ancestor_idx in _ancestors_left:
                    _tmp = (
                        self.b[ancestor_idx]
                        - self.a[ancestor_idx,:] @ x
                    )
                if ancestor_idx in _ancestors_right:
                    _tmp = (
                        self.a[ancestor_idx,:] @ x
                        - self.b[ancestor_idx]
                    )
                _split *= torch.abs(_tmp)
            _ancestors.pop()
            g += self.f[leaf_node, layer_idx] * _split
            rhs[leaf_node,layer_idx] = _split

        g += self.f[leaf_node, self.depth]
        rhs[leaf_node,self.depth] = 1
        return g, rhs.flatten()

    def _validate_parameters(self):
        """
        Make sure the parameters are valid
        """
        if not self.depth > 0:
            msg = "The tree depth should be strictly positive."
            raise ValueError(msg.format(self.depth))

        if not self.lamda >= 0:
            msg = (
                "The coefficient of the regularization term should not be"
                " negative."
            )
            raise ValueError(msg.format(self.lamda))
    def loss(self, input, target):
        """
        Calculate loss
        """
        return self._loss(input, target) + self.lamda * (
            torch.linalg.norm(self.a)
            + torch.linalg.norm(self.b)
            + torch.linalg.norm(self.f)
        )

    def _loss(self, input, target):
        return ((target - input) ** 2).sum() / input.data.nelement()



if __name__ == "__main__":
    input_dim = 9  # the number of inputs
    output_dim = 1  # the number of outputs
    depth = 4  # tree depth
    lamda = 0.001  # coefficient of the regularization term
    init_iter = 50 # random initializations
    epsilon = 1e-7 # epsilon
    lr = 0.001  # learning rate
    batch_size = 128  # batch size
    state=2 # state to approximate

    min_max_scaler = preprocessing.MinMaxScaler()

    df_train_x = pd.read_csv('master_training_states_unnormalized.csv',
                             header=0)
    df_train_y = pd.read_csv('master_training_actions_normalized.csv',
                             header=0)
    train_x_scaled = min_max_scaler.fit_transform(df_train_x)
    df_train_x=pd.DataFrame(train_x_scaled,
                            columns=df_train_x.columns)

    df_val_x = pd.read_csv('master_val_states_unnormalized.csv',
                           header=0)
    df_val_y = pd.read_csv('master_val_actions_normalized.csv',
                           header=0)
    val_x_scaled = min_max_scaler.transform(df_val_x)
    df_val_x=pd.DataFrame(val_x_scaled,
                          columns=df_val_x.columns)
```

```
df_test_x = pd.read_csv('master_test_states_unnormalized.csv',
                        header=0)
df_test_y = pd.read_csv('master_test_actions_normalized.csv',
                        header=0)
test_x_scaled = min_max_scaler.transform(df_test_x)
df_test_x=pd.DataFrame(test_x_scaled,
                       columns=df_test_x.columns)

X_train = torch.tensor(df_train_x.iloc[:, :].to_numpy(),
                       dtype=torch.float32)
Y_train = torch.tensor(df_train_y.iloc[:, state].to_numpy(),
                       dtype=torch.float32)
X_val = torch.tensor(df_val_x.to_numpy(),
                     dtype=torch.float32)
Y_val = torch.tensor(df_val_y.iloc[:, state].to_numpy(),
                     dtype=torch.float32)
X_test = torch.tensor(df_test_x.iloc[:, :].to_numpy(),
                      dtype=torch.float32)
Y_test = torch.tensor(df_test_y.iloc[:, state].to_numpy(),
                      dtype=torch.float32)


model = NNRT(input_dim, output_dim, depth, lamda)
model.random_initialize(X_val, Y_val, init_iter)
optimizer = optim.Adam(model.parameters(),lr=lr,eps=epsilon)
criterion = nn.MSELoss()

train_loss = []
test_loss = []
prev_loss = 0
curr_loss = 1
first = True
while abs(prev_loss - curr_loss) >= epsilon:
    permutation = torch.randperm(X_train.size()[0])
    prev_loss = curr_loss

    model.train()
    for i in range(0,X_train.size()[0], batch_size):
        indices = permutation[i:i+batch_size]
        batch_x, batch_y = X_train[indices,:], Y_train[indices]

        optimizer.zero_grad()
        predictions,rhs = model(batch_x)
```

```
            curr_loss = model.loss(predictions, batch_y)
            curr_loss.backward()
            optimizer.step()
            _ = model._calculate_f(batch_x,batch_y)

        model.eval()
        predictions,_ = model(X_val)
        r2 = r2_score(predictions.detach().numpy(),
                      Y_val.detach().numpy())
        mae = mean_absolute_error(predictions.detach().numpy(),
                                  Y_val.detach().numpy())
        loss_diff = abs(prev_loss - curr_loss)
        print(f"Val r squared: {r2} MAE: {mae} Loss-diff: {loss_diff}")
        print("-" * 10)

    model.eval()
    predictions,rhs = model(X_test)
    loss = r2_score(predictions.detach().numpy(),Y_test.detach().numpy())
    mae = mean_absolute_error(predictions.detach().numpy(),Y_test.detach().numpy())
    loss_diff = abs(prev_loss - curr_loss)
    print(f"Test r squared: {loss} MAE: {mae} Loss-diff: {loss_diff}")
```