Artificial Intelligence Applied to Routing and Energy Prediction of Electric Vehicles

Master's thesis in Complex Adaptive Systems

ELLA GUILADI & JOSEFINE ERIKSSON

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Artificial Intelligence Applied to Routing and Energy Prediction of Electric Vehicle

ELLA GUILADI
JOSEFINE ERIKSSON

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2021

Artificial Intelligence Applied to Routing and Energy Prediction of Electric Vehicle
ELLA GUILADI & JOSEFINE ERIKSSON

© ELLA GUILADI & JOSEFINE ERIKSSON, 2021.

Supervisor: Rafael Basso, Volvo Group Trucks Technology
Examiner: Balázs Adam Kulcsár, Department of Electrical Engineering

Master's Thesis 2021
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Real-time Electric Vehicle Routing connectivity from [1].

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Artificial Intelligence Applied to Routing and Energy Prediction of Electric Vehicle
ELLA GUILADI & JOSEFINE ERIKSSON
Division of Systems and Control
Chalmers University of Technology

Abstract

Dynamic routing of electric commercial vehicles can be a challenging task since besides dealing with a large amount of data, there are also random customer requests that needs to be taken into account when predicting the optimal route. This thesis is based on previous work regarding the Dynamic Stochastic Electric Vehicle Routing Problem (DS-EVRP), however introducing artificial intelligence applied to routing and energy prediction of electric vehicles. A Deep Q-Learning method is proposed to solve the problem instead of using the Q-Learning inspired method as in previous work. The objective is to minimize the expected energy consumption by implementing different variants of Deep Q-Neural Networks by modifying the already existing Reinforcement Learning model. The rollout function used in the existing Reinforcement Learning model is very simple, therefor this thesis also aims to investigate if a different rollout function provides improved results. The main idea is hence to evaluate the choice of rollout function as well as to investigate if Deep Q-learning, with neural networks as function approximators is better than the existing Reinforcement Learning model with Q-tables, when planning a route predicatively with random customer requests. The method is evaluated through simulations that are based on energy consumption data from a real-life traffic model of the city of Luxembourg as well as on a real-life vehicle model. The results indicates that the initial problem formulation and framework was ill-posed for this solution method, resulting in a time-consuming task. The neural network did not outperform the existing Reinforcement Learning model with Q-tables in terms of time and model performance. However, the results indicated increased model performance when changing some initial problem settings as well as tuning of the network architecture and hyperparameters. The Double Deep Recurrent Q-Network also showed to be a less time consuming task with better model performance when using LSTM layers. Further, the results showed that the ACO algorithm generally outperformed the Tabu search. It is also shown that the usage of a rollout function effectively guides the agent in taking favorable actions during training, however resulting in less exploration of the environment. For future work it would therefor be interesting to investigate the neural networks further, especially the Double Deep Recurrent Q-Network, by re-formulating the initial problem formulation as well as perform a more thorough tuning of the network architecture and hyperparameters.

Keywords: Reinforcement Learning, Deep Q-Learning, Deep Q-Neural Networks, Deep Q-Recurrent Neural Networks, Artificial Intelligence, Electric Vehicles, Energy Consumption, Vehicle Routing, Green Logistics, Ant Colony Optimization

Acknowledgements

We would like to thank our examiner Balázs Adam Kulcsár from the Department of Electrical Engineering at Chalmers University of Technology for insightful ideas, comments, feedback and challenging discussions helping us to navigate the direction of entire thesis as well as the writing of the report. We would also like to thank our supervisor, Rafael Basso from Volvo Group Trucks Technology for providing the data as well as valuable scripts and inputs for a crucial base of the thesis. Furthermore, we would also like to thank Volvo Group Trucks Technology for giving us the opportunity and means to conduct this project. Writing this thesis was a valuable and interesting experience for us and even though it was a challenging task, Balázs Adam Kulcsár and Rafael Basso steered us in the right direction in both a theoretical as well as practical sense.

Lastly, we would like to show our greatest gratitude to our family and friends for providing us with the constant encouragement and support throughout this thesis. Without their support and help, the completion of this thesis would not be possible. Thank you!

Ella Guiladi and Josefine Eriksson, Gothenburg, June 2021

Contents

List of Figures **xiii**

List of Tables **xv**

List of Abbreviations **xvii**

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 Outline . . . 2
- 1.2 Aim . . . 3
- 1.3 Literature Review . . . 3
  - 1.3.1 Problem Formulations . . . 3
  - 1.3.2 Solution Methods . . . 4
- 1.4 Contributions . . . 5
- 1.5 Research Questions . . . 6
- 1.6 Limitations . . . 6

2 Theory **7**
- 2.1 The Vehicle Routing Problem . . . 7
  - 2.1.1 Markov Decision Process . . . 7
  - 2.1.2 Dynamic Programming . . . 8
  - 2.1.3 Rollout Functions . . . 9
    - 2.1.3.1 Ant Colony Optimization . . . 9
    - 2.1.3.2 Tabu Search . . . 10
- 2.2 Artificial Neural Network . . . 11
  - 2.2.1 Fully Connected Multi Layer Feed-Forward Neural Network . . 12
    - 2.2.1.1 Activation Functions . . . 13
    - 2.2.1.2 Initialization of Network Parameters . . . 15
    - 2.2.1.3 Loss Functions . . . 17
    - 2.2.1.4 Backpropagation . . . 18
    - 2.2.1.5 Training and Optimization Algorithms . . . 18
  - 2.2.2 Recurrent Neural Network . . . 20
    - 2.2.2.1 Long Short-Term Memory . . . 21
- 2.3 Q-Learning . . . 23
  - 2.3.1 Policy Optimization . . . 25
- 2.4 Deep Q-Learning . . . 25
  - 2.4.1 Deep Q-Network . . . 25

2.4.1.1 Target Network 26
2.4.1.2 Experience Replay 27
2.4.1.3 Double Deep Q-Network 28
2.4.2 Deep Recurrent Q-Network 29

3 Problem Formulation **31**
3.1 DS-EVRP modeled as a Markov Decision Process 31
3.1.1 Objective Function 32
3.1.2 Solution Method 32
3.1.2.1 Deterministic Re-optimization 33
3.2 Modification to Previous Work 33
3.2.1 Deep Reinforcement Learning 33

4 Implementation **35**
4.1 Input Data 35
4.2 State Configuration 36
4.3 Rollout Functions 36
4.4 Double Deep Q-Network 37
4.5 Double Deep Recurrent Q-Network 38
4.6 Performance Evaluation 39

5 Results **41**
5.1 Double Deep Q-Neural Networks 41
5.1.1 Network Architecture 41
5.1.1.1 State Representation 43
5.1.2 Hyperparameters 43
5.2 Double Deep Recurrent Q-Networks 46
5.2.1 Network Architecture 46
5.3 Case Study 49
5.3.1 Double Deep Q-Network 49
5.3.2 Double Deep Recurrent Q-Network 49
5.4 The Rollout Functions 50
5.4.1 Double Deep Q-Network 50
5.4.2 Double Deep Recurrent Q-Network 51
5.4.3 Ant Colony Optimization vs Tabu Search 52

6 Discussion **55**
6.1 Problem Formulation and Implementation 55
6.1.1 State Representation 57
6.1.2 Non-linear Optimization Problems 58
6.2 Double Deep Q-Neural Network 59
6.2.1 Network Architecture 59
6.2.2 Tuning of the Network 60
6.3 Double Deep Recurrent Q-Network 62
6.3.1 Network Architecture 63
6.3.2 Tuning of the Network 63
6.4 Case Study 65

6.5 The Rollout Functions 65
6.5.1 Double Deep Q-Network 66
6.5.2 Double Deep Recurrent Q-Network 67
6.5.3 Hyperparameter Search 68
6.5.4 Ant Colony Optimization vs Tabu Search 69

7 Conclusion **71**
7.1 Research Questions 71
7.1.1 What is the effect of implementing a Deep Q-Learning model compared to the existing Reinforcement Learning model? . . . 71
7.1.2 How does the choice of Deep Q-Neural Network impact the Deep Q-Learning model? 72
7.1.3 What impact has the rollout function and the choice of it? . . 72
7.1.4 Concluding remarks 73

8 Future work **75**

Bibliography **77**

A Appendix 1 **I**

B Appendix 2 **III**

C Appendix 3 **V**

D Appendix 4 **VII**

E Appendix 5 **IX**

List of Figures

2.1 The design of an artificial neuron as well as an vanilla artificial neural network from [24]. 12
2.2 Architecture of a typical multi-layer feed forward neural network from [28]. 12
2.3 The sigmoid and tanh function with their derivatives, illustrating the vanishing gradient problem from [30]. 14
2.4 The ReLU function and its derivative from [30]. 15
2.5 An unrolled Recurrent Neural Network from [48]. 20
2.6 The structure of a LSTM cell network with the different gates and cell state. 22
2.7 Visual comparison of Q-Learning and Deep Q-Learning from [63]. . . 26

5.1 Average episode loss over all trained episodes for a DDQN with 2 hidden layers and 9 neurons in each hidden layer. 42
5.2 The average weights and biases per layer, over all training episodes for a DDQN with 2 hidden layers and 9 neurons in each hidden layer. 43
5.3 Average episode loss over all trained episodes for a Double-DRQN with 1 LSTM layer, 10 units, 2 hidden layers and 64 neurons in each hidden layer. 47
5.4 The average weights and biases per layer, over all training episodes for a Double-DRQN with 1 LSTM layer, 10 units, 2 hidden layers and 64 neurons in each hidden layer. 48
5.5 The average episode loss over all trained episodes for when solely using the ϵ-greedy algorithm, solely using the ACO algorithm and combining the two algorithms during training. 51
5.6 The average episode loss over all trained episodes for when solely using the ϵ-greedy algorithm during training, solely using the ACO algorithm and combining the two algorithms. 52

List of Tables

5.1 This table presents the difference for a DDQN configuration with varying number of hidden layers as well as neurons in each hidden layer. 42

5.2 This table presents the difference for a DDQN with 2 hidden layers, 9 neurons in each layer, using a reduced as well as a full state representation. 43

5.3 This table presents the difference and run-time for a DDQN with 2 hidden layers, 9 neurons in each layer and varying values of τ. 44

5.4 This table presents the difference and run-time for a DDQN with 2 hidden layers, 9 neurons in each layer and varying learning rate α. . . 44

5.5 This table presents the difference and run-time for a DDQN with 2 hidden layers, 9 neurons in each layer and varying the batch size. . . 44

5.6 This table presents the difference for a DDQN with 2 hidden layers and 9 neurons in each layer, using the two weight initialization methods, He weight initialization and Random Uniform initialization. 45

5.7 This table presents the difference for a DDQN with 2 hidden layers and 9 neurons in each layer using the Huber loss function with varying values of δ. 45

5.8 This table presents the difference for a Double-DRQN configuration with 1 LSTM layer and varied number of units, hidden layers and neurons in the hidden layers. 46

5.9 This table presents the difference for a Double-DRQN configuration with 2 LSTM layers, 10 units in each layer and different number of hidden layers as well as neurons in each hidden layer. 47

5.10 This table presents the difference for a Double-DRQN configuration with 1 LSTM layer and 10 units, 2 hidden layers with 64 neurons each and different dropout rates applied to the LSTM layer. 48

5.11 This table presents the difference and run-time for a Double-DRQN configuration with 1 LSTM layer and 10 units, 2 hidden layers with 64 neurons each and different batch sizes. 48

5.12 This table presents the resulting difference when using case number 1 and 15 with 10 customers as well as case 61 and 75 with 20 customers from the input data. The best found DDQN configuration was used. . 49

5.13 This table presents the resulting percentage difference when using case number 1 and 15 with 10 customers as well as case 61 and 75 with 20 customers from the input data. The best Double-DRQN configuration was used. 50
5.14 This table presents the resulting difference when solely using the ϵ-greedy algorithm, when solely using the ACO algorithm and when using a combination of these two, using the best DDQN configuration. 50
5.15 This table presents the resulting difference and run-time when solely using the ϵ-greedy algorithm, when solely using the ACO algorithm and when using a combination of these two, using the best Double-DRQN configuration. 51
5.16 This table presents the resulting difference and run-time (h) for when either using the ACO algorithm or the Tabu search as a rollout function for two different networks. These comparisons are made on the best DDQN and Double-DRQN configurations. Case 1 and 15 with 10 customers as well as 61 and 75 with 20 customers from the input data was used. 53

List of Abbreviations

ACO Ant Colony Optimization. v, xiii, xvi, 2, 4, 5, 7, 9, 33, 35, 36, 38, 41, 46, 49–53, 66–70, 72, 73, 75
ADAM Adaptive Moment Estimation. 19, 20
ANN Artificial Neural Network. 2, 11, 16

BGD Batch Gradient Descent. 18
BPTT Backpropagation Through Time. 21

DDQN Double Deep Q-Network. xiii, xv, xvi, 2, 5, 6, 28, 33, 35–37, 39, 41–45, 49, 50, 53, 55, 57, 59, 62–65, 68, 69, 72, 73, 75
DNN Deep Neural Network. 25–27, 58
DP Dynamic Programming. 2, 5, 7–9
DQL Deep Q-Learning. 2, 5, 25, 26, 28, 29, 71, 73, 75
DQN Deep Q-Network. 2, 25–29, 55, 56
DRQN Deep Recurrent Q-Network. xiii, xv, xvi, 2, 5, 6, 29, 33, 35, 41, 46–51, 53, 55, 62–65, 67–69, 72, 73, 75
DS-EVRP Dynamic Stochastic Electric Vehicle Routing Problem. v, 1–3, 31

EVRP Electric Vehicle Routing Problem. 3–5

FFNN Feed Forward Neural Network. 12, 13, 15, 18, 20

GTT Group Trucks Technology. 1, 2

LSTM Long Short-Term Memory. v, xiii, xv, 2, 16, 21–23, 29, 38, 39, 46–49, 62–65, 73

MAE Mean Absolute Error Loss. 17
MDP Markov Decision Process. 1, 2, 7–9, 29, 31–33, 37, 55, 75
MSE Mean Square Error Loss. 17, 37, 41, 45, 61, 65

POMDP Partially Observable Markov Decision Process. 8

QL Q-Learning. 2, 7, 23–28, 55, 65

ReLU The rectified linear activation function. xiii, 14–16, 37, 58, 62
RL Reinforcement Learning. 5, 7, 23
RNN Recurrent Neural Network. 2, 16, 20, 21

SGD Stochastic Gradient Descent. 18, 19, 27

VRP Vehicle Routing Problem. 2–5, 7, 9, 10, 71, 73, 75, 76

1

Introduction

In collaboration with Volvo Group Trucks Technology (GTT), this master's thesis project will develop methods to improve the already existing Reinforcement Learning model. This thesis is primarily directed to a target group within vehicle routing, with little or some knowledge within Artificial Intelligence. The Sections below will introduce the background, aim, literature review, contributions and research questions as well as limitations of the project.

1.1 Background

Along with the increasing interest in sustainable transport solutions during the recent years, the interest in electric commercial vehicles has also grown significantly. Even though the battery capacity is under continual improvement, it is still one of the vehicles main limitations. Not only is the battery capacity limited by factors such as its size and weight. The energy consumption itself does also depend on various unsure factors such as traffic conditions and factors related to the drivers behaviour.

In order to ensure that the vehicle will not run out of energy during a route, one needs to handle several challenges regarding energy management. This thesis project will focus on one such challenge - the Dynamic Stochastic Electric Vehicle Routing Problem (DS-EVRP), where the dynamical aspect is introduced by random customer requests. At Volvo Group Truck Technology (GTT), the project report *Dynamic Stochastic Electric Vehicle Routing with Safe Reinforcement Learning* [2] has implemented a Safe Reinforcement Learning model to solve this problem. The problem is modeled as a Markov Decision Process (MDP), and the method for the Safe Reinforcement Learning model is based on Q-learning. This existing model will be the base for the thesis project.

A major limitation of Q-learning is however when dealing with environments including continuous and infinite state-action spaces, which can lead to inefficient learning due to the curse of dimensionality when the number of combinations increases quickly. A solution to these limitations can be to discretize the state space and then storing the Q-values corresponding to state-action pairs in a table. However, when dealing with an infinite state-action space, this does still not ensure that the curse of dimensionality can be avoided. Another solution is thereby to apply a function approximator that learns the value function, taking states as inputs

and outputting Q-values for each action. Since deep neural networks are powerful function approximators, it seems logical to try to adapt them for this role. The advantage of a deep neural network is thereby to avoid building a huge Q-table if the number of state-action pairs are quite large as well as being able to include continuous state-action values, thereby avoiding inefficient learning.

Vehicle Routing Problems (VRP) are known for growing exponentially in size with increasing number of nodes, i.e. increasing number of customers and charging stations. It is therefor of interest for companies such as Volvo GTT as well as researchers within the Vehicle Routing field at Chalmers University of Technology, to find suitable functions approximators for such problems, such as different variants of Deep Q-Neural Networks.

1.1.1 Outline

This report is divided into 8 main Sections. In Section 1, the introductory framework of this report is presented. First a background and introduction of the VRP and how it can be approached is presented as well as its relevance for Volvo GTT and Chalmers University of Technology. The aim of the thesis is then presented, as well as a literature review, the contributions in this paper, formulated research questions and limitations that has been made.

In Chapter 2, the theoretical framework of this thesis is presented. First, an introduction to the VRP is presented, including the mathematical modeling framework Markov Decision Process (MDP) as well as some solution methods, such as the Dynamic Programming (DP), Ant Colony Optimization (ACO) and Tabu Search. Thereafter, the theory behind Artificial Neural Network (ANN) is presented by covering the building blocks of the Fully Connected Multi Layer Feed-Forward Neural Networks which are activation functions, initialization of network parameters, loss functions, backpropagation as well as training and optimization algorithms. The theory behind Recurrent Neural Network (RNN) is then covered in a similar manner, by covering the topic Long Short-Term Memory (LSTM). Lastly, the theory concerning Q-Learning (QL) is presented by focusing on Deep Q-Learning (DQL) through Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) as well as Deep Recurrent Q-Network (DRQN).

Chapter 3 presents the problem formulation in terms of the DS-EVRP modeled as a MDP by presenting the objective function and the solution method. Lastly, the modifications to previous work is presented by introducing the deep Q-learning model. In Chapter 4, the implementations of the input data, state configuration, rollout functions, DDQN, Double-DRQN and performance evaluation are explained. The results for the network architecture, state representation and hyperparameters of the DDQN and the Double-DRQN are then further presented in Chapter 5. Moreover, the results from the case study and effects of the choice of rollout function are

also presented in this chapter, followed by an comparison of the two rollout functions.

The discussion is presented in Chapter 6, starting with the discussion of the problem formulation and implementation, covering the state representation and the non-linear optimization problem. Thereafter, the discussion regarding the two networks is presented by analysing their network architecture as well as tuning of the networks. This is followed by a discussion regarding the case study as well as concerning the effects of the different rollout functions for both networks. In Chapter 7, the conclusion is presented by answering the research questions formulated in Section 1.5 as well as some concluding remarks. The thesis ends with Chapter 8, where potential future work within this VRP is introduced.

1.2 Aim

This thesis aims to tackle the Dynamic Stochastic Electric Vehicle Routing Problem (DS-EVRP) by implementing different variants of Deep Q-Neural Networks through Deep Reinforcement Learning by modifying the already existing Reinforcement Learning model, aiming to minimize the expected energy consumption. This is done to evaluate if Deep Q-Learning, with neural networks as function approximators is better than the existing Reinforcement Learning model with Q-tables.

The rollout function used previously in the existing Reinforcement Learning model is very simple and could be improved, therefore this thesis also aims to investigate if a different rollout function provides improved results.

1.3 Literature Review

The classic Vehicle Routing Problem (VRP) aims to find the optimal set of routes for a vehicle to visit a number of customers. Several variations have been studied since it was introduced in the fifties and therefor different solutions has been presented and developed. For an overview of the theory of the VRP, [3] was mainly used. This project is based on previous work, such as the project report [2] and [4], covering Electric Vehicle Routing Problems (EVRP).

This Section presents an overview of the study of VRPs, including differences in problem formulations as well as solution methods.

1.3.1 Problem Formulations

In order for the VRP to be applicable on real world problems with different constraints related to either the vehicle or the surrounding, various modifications have been introduced. This Section will focus on studies concerning VRPs formulated for electric vehicles.

In [5], a general EVRP was investigated. The objective of their proposed problem was similar to a classic VRP, i.e. to find an optimal set of routes that minimizes the total cost as well as the number of electric vehicles needed for that solution. The total cost included both travel time and battery consumption. Furthermore, they considered the effect of vehicle load on battery consumption, which they claim was novel to their study. The results indicated that the effect of vehicle load cannot in fact be ignored, which implies that vehicle load is a factor to take into consideration when optimizing a routing strategy for electric vehicles.

The work in [6] had a different focus area in their investigation, compared to [5]. They investigated the uncertainty in battery consumption of electric vehicles that comes from a number of external factors, such as driver behaviour and road conditions. Their objective was thereby formulated to optimize routes based on an expected energy consumption while ensuring that the battery of a given vehicle will not be depleted during a route as well as by taking external factors into account. Their results showed that there is a trade-off between minimizing energy consumption and the risk of battery depletion when planning routing strategies. However, the right balance can be reached using an optimization method.

In the project report [2], the effect of vehicle load was taken into account as in [5] as well as the uncertainties regarding energy consumption as in [6]. However, in [2] the EVRP was further modified with the addition of stochastic customer requests. The addition of stochastic customer requests made the problem even more complex, since the optimization should take dynamic customer requests into consideration. The results did however show that it is possible to minimize the energy consumption in a safe way, by taking the risk of battery depletion as well as stochastic customer requests into account when planning the routes.

1.3.2 Solution Methods

Such as the wide range of modifications to the classic VRP that have been introduced in literature, there are also a wide range of developed solution methods. This Section presents studies, using some of the most common methods.

In [7], a general VRP was solved using an Ant Colony Optimization (ACO) algorithm. The ACO algorithm is a metaheuristic, which can be used to find approximate solutions for optimization problems. This algorithm is inspired by the behaviour of real ants, who deposit pheromones in order to navigate other ants from their colony to find resources in their surroundings. The results in [7] indicated that after tuning a set of hyperparameters related to the ACO algorithm, the best possible performance was achieved. It can thus be assumed that the ACO algorithm implemented in [7] is an appropriate choice of optimization method for VRPs of similar complexity. However, the problem in [7] was quite simple compared to for example the VRP in [2]. Further, there were no indications in the results on how this solution method will perform on problems with increasing complexity.

Compared to the ACO algorithm used in [7], which finds an approximate solution to a problem, Dynamic Programming (DP) is a method used to find an exact optimal solution. The core of the method relies on the fact that the optimal solution to a problem actually depends on the optimal solution to its subproblems. Therefor, the initial problem is broken down to interrelated subproblems, which are simpler to solve. However, as the complexity of the initial problem increases, so does the number of interrelated problems. In [8], the VRP was modified with the addition of stochastic customer requests, which made it to complex to be solved using the native DP method. Instead, approximate DP methods were used in [8], which for instance made some simplifying assumptions regarding the solutions to the interrelated subproblems. The work presented promising results, indicating that approximate DP methods can be used to solve more complex VRPs where native DP methods are no longer applicable.

Another area of methods that can be used for more complex VRPs is Reinforcement Learning (RL), which is an area within machine learning. In RL, a decision maker, called agent, learns how to take favorable actions in different situations, described in Section 2.1.1. In [9], a so called pointer network, which is a kind of deep neural network, was trained using policy gradient algorithms to solve a modified VRP. Policy gradient algorithms belong to the policy optimization (Section 2.3.1) subarea within RL, and aims to find the optimal solution by using gradient descent. In [9], the modification of the VRP included an addition of a capacity constraint, compared to the stochastic customer requests in [8]. The results from [9] indicated that the combination of a deep neural network and a policy optimization algorithm is a powerful setup for solving the VRP with capacity constraints. Further, the results did also indicate that this combination performs well on different combinatorial optimization problems with increasing complexity, such as the stochastic VRPs explained in [8].

Lastly, to the best of our knowledge there is no published paper solving the EVRP with stochastic energy and dynamic customer requests using Deep Q-Learning (DQL).

1.4 Contributions

The main contributions presented in this paper are:

- A Deep Q-Learning solution method applied on the VRP including:
  - A DDQN and a Double-DRQN as function approximators
  - A dynamic customer request strategy
  - A training strategy
- Computational experiments to evaluate the proposed approach
- Comparative study between a probabilistic and a deterministic rollout function

1.5 Research Questions

The questions that are aimed to be answered by this thesis project are stated bellow:

- What is the effect of implementing a Deep Q-Learning model compared to the existing Reinforcement Learning model?
- How does the choice of Deep Q-Neural Network impact the Deep Q-Learning model?
- What impact has the rollout function and the choice of it?

1.6 Limitations

Due to the limitations of time and resources in the execution of this thesis project, some restrictions were decided to be made. Depending on the expenditure of time, limitations in what specific configurations that could be evaluated were made during the execution. However, the initial limitations were the following:

- Due to the initial problem formulation in combination with the neural network, the safety constraints in the safe reinforcement learning was not taken into account. This was not taken into account in the sense that the battery was assumed to have capacity that lasts for an entire route, i.e. the vehicle will never have to plan the route based on charging.
- Due to the time constraints, the number of evaluated cases was restricted.
- The tuning of the networks were performed using solely case number 1 with 10 customers.
- Hyperparameter tuning as well as some tuning of initial settings was only conducted on the DDQN, and thereafter applied to the Double-DRQN. Therefor, the tuning of hyperparameters was limited in the case of the Double-DRQN.

2
Theory

This Chapter will cover the underlying theory that this thesis is based on. An introduction to the Vehicle Routing Problem (VRP), including theory about common solution methods will be presented in the first Sections bellow. Thereafter, the main concepts of Artificial Neural Networks will be covered. Lastly, theory regarding the concepts of Q-Learning and deep Q-Learning will be presented.

2.1 The Vehicle Routing Problem

The core of the Vehicle Routing Problem (VRP) is to design optimal pick-up and/or delivery routes for a predefined set of vehicles, all starting from a depot. The customers, which requests the deliveries and/or pick-ups, are geographically scattered within a predefined area. Except for the geographic distance between the customers, restrictions such as the vehicles capacity, time windows and precedence relations between customers have to be considered when optimizing the routes [3]. A commonly used mathematical framework for modeling such optimization problems is the Markov Decision Process (MDP) [10]. The VRP was introduced initially in the fifties, and has since then been formulated in various ways in order to fit the variations of constraints that occur in practice [3].

Except the wide range of variations within the problem formulation, there is also a wide range of solution methods for the problem. One can for example use an exact algorithm, such as Dynamic Programming (DP), or meta-heuristics such as Ant Colony Optimization (ACO) and Tabu Search [3]. Further, one can also use methods within Reinforcement Learning (RL), such as Q-learning (QL). QL will be covered later in Section 2.3.

2.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a stochastic sequential decision process, meaning that it is a model for a dynamic system under the control of a decision maker [11]. The decision maker, often denoted as agent, observes the state of the system at each time a decision can be made. The system is commonly referred to as environment [12]. Based on what the agent observes from the state, it chooses an action from a set of alternatives. When the chosen action is executed, the agent receives an immediate reward which is a measure of the quality of the action. In addition, a probability distribution on the successive environment states is specified. Note

that the agent only receives an immediate reward for an action, and that the action changes the future evolution of possible actions in the environment. The agent must therefor take future consequences into account when deciding on what action to take. The agent's objective is to choose a sequence of actions such that the total reward is maximized. A sequence of actions, from the initial- to the terminal state, is called a policy and is denoted π. Further, the objective is thus to obtain the optimal policy π^* [11].

The time set, T, of the problem can be either finite or infinite, and the time-points, t, when a decision can be made can be either discrete or continuous depending on the problem that is modeled. What information of the environment that is included in the observed state does also depends on the problem and the state values can either be continuous or discrete. For each state $s \in S_t$ at time t, where S_t is the entire set of possible states at that time, there is a corresponding set of possible actions, $A_{s,t}$, for the agent to choose an action from $a \in A_{s,t}$. These actions can also be discrete or continuous. The reward $r_t(s, a)$ for taking action a in time t and state s is defined as in equation 2.1.

$$r_t(s_t, a) = \sum_{s_{t+1} \in S_{t+1}} r_t(s_{t+1}, a) p_t(s_{t+1} \mid s_t, a), \tag{2.1}$$

where $p_t(s_{t+1} \mid s_t, a)$ is the transition probability function, i.e. the probability of ending up in state s_{t+1} after executing action a in state s_t. It is the collection of sets; T, S_t, $A_{s,t}$, $r_t(s_t, a)$ and $p_t(s_{t+1} \mid s_t, a)$ that forms the MDP [11].

In the native MDP described in this Section, it is assumed that the state is fully observable to the agent. When modeling real world problems, this is however not always the case. An extension of the MDP has therefore been introduced, namely the Partially Observable Markov Decision Process (POMDP). POMDPs are similar to MDPs in that sense that there is a set of states, actions, transitions and rewards. The difference of a POMDP from a MDP is however that the agent cannot completely observe the underlying state of the environment. Therefor, to obtain information about the states, the agent makes observations that depends on the current state instead of directly observing it as in the MDP. These observations are then used to form a belief of in what state the environment is in, called belief state. These beliefs are expressed as a probability distribution over the states, i.e. the probability of each observation for each state in the model. Similar to the MDP, the solution of the POMDP is a policy giving the optimal actions for each belief state [13].

2.1.2 Dynamic Programming

Dynamic Programming (DP) are model-based mathematical methods for solving sequential decision problems where the model is fully known [14]. Thanks to the known model an optimal, or exact, solution can be found. The core idea of the methods is to decompose a multistage problem into a sequence of simpler interrelated one-stage problems. This is done in a recursive manner. After the decomposition,

one ends up with the so-called Bellman's equation. The Bellman's equation gives the value of a decision at a certain point in terms of both the payoff from prior decisions and the value of remaining decisions [14]. Since all decisions are interrelated, remaining decisions depend on previous ones as well.

A DP problem is often modeled as a MDP. Applications of DP have been successful in various sequential optimization problems, such as the traveling salesmen problem [15]. However, as the computational power required increases rapidly with the problem size, DP is not suitable for more complex problems [14]. The success of solely using DP in applications of the VRP is thus limited. A combination of deep learning and DP, deep policy DP [16], as well as approximate DP [8], have been proposed as more suitable solution methods for the VRP since they are not as sensitive to the problem complexity.

2.1.3 Rollout Functions

A rollout function uses a given heuristic to start from, in order to create an improved heuristic generating better performance than the native one. This is a commonly used method since it is often simple to implement and provides improved results for optimization problems efficiently [17]. In this Section, two different rollout functions will be described.

2.1.3.1 Ant Colony Optimization

The Ant Colony Optimization (ACO) algorithm is a probabilistic technique used to solve optimization problems that can be summarized as finding good paths through graphs. The algorithm is inspired by the behaviour of real ants, which deposits a chemical substance called pheromone on a path they traverse to inform other ants from the colony which path to choose. The ants in the colony will thereafter choose the path with the highest pheromone level, which often is equal to the most favourable path [18]. There exists different variations of the ACO algorithm, where Ant System, Ant Colony System and Max-Min Ant System are the most commonly used ones. The Ant System was the fist variant introduced in literature [19], and also the one that will be described in detail below.

The characteristic pheromone deposition mechanism is mimicked by the algorithm in a way that in each iteration, the pheromone level is updated by all the m ants that have built a solution in that iteration. The pheromone level on the edge between node i and j is denoted as τ_{ij} and the pheromone level update follows equation 2.2. A commonly used initialization strategy for the pheromone level is to set it to $\tau_{ij} = m/D^{nn}$, where D^{nn} is the nearest neighbour path length [20].

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}. \tag{2.2}$$

In equation 2.2, ρ is the evaporation rate and m the total number of ants in the colony. It has been shown that $\rho = 0.5$ and $m = n$, where n is the number of nodes

in the path to be constructed, results in good performance for many problems [20]. $\Delta\tau_{ij}^k$ is the amount of pheromone deposited by ant k on the edge between node i and j, obtained by equation 2.3.

$$\Delta\tau_{ij}^k = \begin{cases} 1/L_k, & \text{if edge } ij \text{ was traversed by ant } k, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

In equation 2.3, L_k is the total length of the path constructed by ant k in that iteration. It follows from the equation that pheromone is only deposited on an edge traversed by ant k [19].

When an ant constructs a path, it does so by selecting the next node to visit using a stochastic approach. The probability for ant k to choose node j when currently being on node i is given by equation 2.4.

$$p(c_{ij} \mid s_k^p) = \begin{cases} \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{c_{lj} \in N_{(s_k^p)}} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}}, & \text{if } j \in N_{(s_k^p)}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

In equation 2.4, s_k^p is ant k's partial solution, i.e. the path of nodes visited so far. $N_{(s_k^p)}$ is the set of nodes that still can be chosen, α is a factor that control the importance of the pheromone and β is the visibility factor, controlling the importance of the visibility parameter η. For many problems, $\alpha = 1$ and $\beta \in [2, 5]$ provides good performance [20]. The visibility parameter equals $\eta = 1/d_{ij}$, where d_{ij} is the distance between node i and j [19]. The above described path construction procedure is executed by each ant in each iteration. The algorithm runs until a maximum number of iterations is reached, or until a stopping criteria is met [18].

2.1.3.2 Tabu Search

Another common meta-heuristic used to solve sequential optimization problems, such as the VRP, is the Tabu Search algorithm. The main idea of the algorithm is to iteratively explore a space of problem solutions, X. This is done by

1. generating an initial solution $s \in X$, e.g the nearest neighbour solution,
2. iteratively generating sets of neighbouring solutions $V^* \in N(s)$ to s, with V^* being a set of solutions and $N(s)$ the solution neighbourhood to s.
   - For each V^*, find the best solution *s'*,
   - if *s'* is better than s, set *s=s'*.

Each set V^* is obtained by performing different moves on the initial solution s [21]. Such a move can for example be to shift nodes forward [2], or to perform the so called 2-opt algorithm on the initial solution [21]. To shift nodes forward implies to swap the positions of two adjacent nodes by shifting one of them forward. 2-opt is on the other hand a local search algorithm that aims to reorder a solution with crossing paths, to make sure that these does not exists in the final solution. This reordering is done iteratively, by shifting the order of two nodes at a time so that the resulting solution is shorter [22]. The iterative process of generating and evaluating

solutions is stopped either when a certain criteria regarding the length of the current best solution is met, or after a maximum number of iterations [23].

Memories, one short term and one long term, are used to help the algorithm improve the solutions as well as explore the solution space X. The task for the short-term memory is to prevent the algorithm to obtain an old solution and to keep track of common properties of solutions that have been good so far [21]. By keeping track of these common properties of good solutions, the algorithm can be encouraged to use them when generating new solutions. This is called intensification, as it intensifies the search for an optimal solution. The intensification procedure is performed with help of a so called Tabu list, which stores a fixed number of recently made moves. When generating new solutions, solutions that includes moves that occur on the Tabu list will not be considered. However, there is a so-called aspiration criterium corresponding to each move that can make the move bypass the Tabu list if it leads to a good solution. To bypass the Tabu list, the move's aspiration criteria must be higher then a threshold [21]. In this way, moves common to good solutions can be used even if they have been used before. The Tabu list is updated in each iteration, by adding moves generated in that iteration and removing moves older than a maximum number of iterations [23].

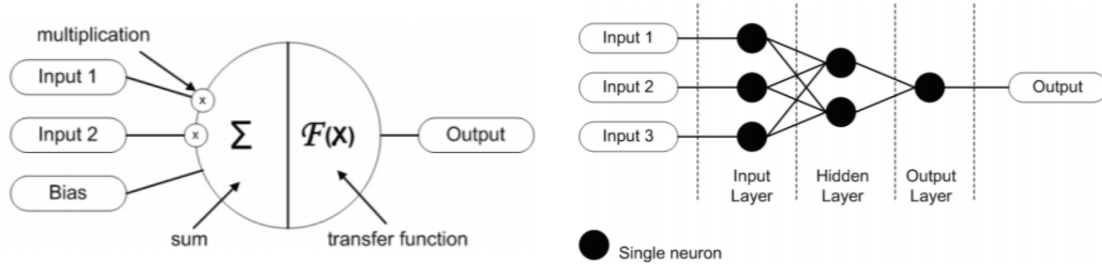
The task for the long term memory is to make the algorithm explore the solution space X by avoiding already obtained solutions. This is called diversification and is executed by a so-called frequency memory, which stores the total number of iterations each solution has obtained [21]. Since solutions that have been obtained more times are less likely to be picked again, the algorithm is in this way encouraged to explore the solution space.

2.2 Artificial Neural Network

An Artificial Neural Network (ANN) is a mathematical model that is inspired by the functionalities of biological neural networks and applied in modern computers. By using the processing of the brain as a basis, it is possible to develop algorithms that can be used to model complex patterns and prediction problems. The basic building block of such a network is the artificial neuron, which is a mathematical model/function [24]. At the beginning of the artificial neuron, each input is weighted, i.e. every input value is multiplied with an individual weight. In the middle Section, there is a sum function that sums up all of the weighted inputs and biases and thereafter processes the sums with an activation function (also called transfer function). At the end Section, an artificial neuron passes the processed information via outputs [24]. The artificial neuron is visualized in Figure 2.1a with its weighted inputs, transfer function, bias and output.

The full potential and computational power of these artificial neurons comes to life when they are interconnected into an ANN, which can be seen in Figure 2.1b. The artificial neurons in the ANN are however not interconnected randomly. Depending

on what kind of problem that is being solved, the architecture varies. After the architecture is chosen, a process of fine-tuning it as well as the parameters included in the network is performed. After fine-tuning, a training phase is initialized where the network learns to solve the type of given problem, before it is actually used for its main purpose [24]. For a more detailed explanation regarding this Section, the reader is encouraged to read the theoretical material in [24], [25] and [26] further.



(a) An Artificial Neuron

(b) An Artificial Neural Network

Figure 2.1: The design of an artificial neuron as well as an vanilla artificial neural network from [24].

2.2.1 Fully Connected Multi Layer Feed-Forward Neural Network

The fully connected multi-layer feed-forward neural networks are popular networks that are used for a variety of problems. These neural networks are capable of modeling complex relations where an output is predicted given a certain input. By giving the network examples of known outputs, it can build a model which is able to learn and predict unknown outputs, by extracting information from the examples. This process is more commonly known as supervised learning [27]. This kind of network have neurons that are arranged in layers, which can be visualised as in Figure 2.2. The first layer takes in inputs and the last one produces the outputs. The middle layers are called hidden layers and can constitute of one or more layers.

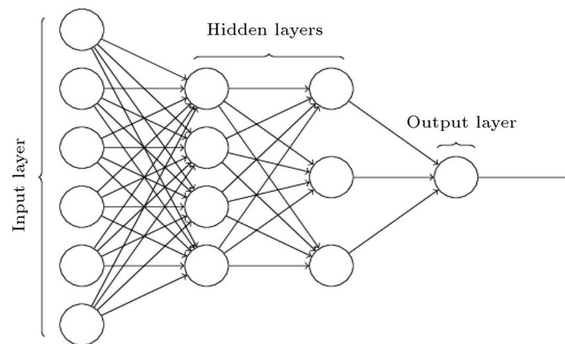


Figure 2.2: Architecture of a typical multi-layer feed forward neural network from [28].

The objective of a fully connected multi-layer feed-forward neural network is to approximate some function $f(x)$. The Feed Forward Neural Network (FFNN) defines

a mapping $y = f(x;\theta)$, with y and x defined as the output respectively the input, and learns the value of the parameters θ that results in the best function approximation. These networks are called feed-forward since the flow of information is in the forward direction, where input x is used to calculate some intermediate function in the hidden layer, which thereafter is used to calculate the output y. Each neuron in one layer is therefore connected to every neuron in the next layer, feeding the information forward from one layer to the next. The input x goes through the layers of neurons, where one neuron i in layer j is defined in equation 2.5, where ϕ is the activation function, N corresponds to the number of input nodes, b is the bias and w is the weights. No feedback connections are included, i.e. there are no outputs from the model that are fed back into itself [25].

$$a_j^i = \phi\left(\sum_{k=0}^{N} w_{jk}^i x_{jk}^i + b_j^i\right), w_j^i, b_j^i \in \theta \tag{2.5}$$

2.2.1.1 Activation Functions

The activation function in the FFNN defines how the weighted sum of the input will be transformed from the neuron into the activation of the neuron or the output for that given input. In order for a FFNN to be able to represent nonlinear mappings, activation functions needs to be inserted in the different layers of the network. There are several different activation functions and the choice of which one to use can be crucial. The activation function in the hidden layers controls how well the model learns during training and the choice of activation function in the output layer decides what kind of prediction the network makes [25]. One important feature of the activation function is that it needs to be differentiable. This is necessary to be able to perform the backpropagation, which enables the computation of the losses w.r.t the weights or in order to optimize the weights using optimization techniques [29].

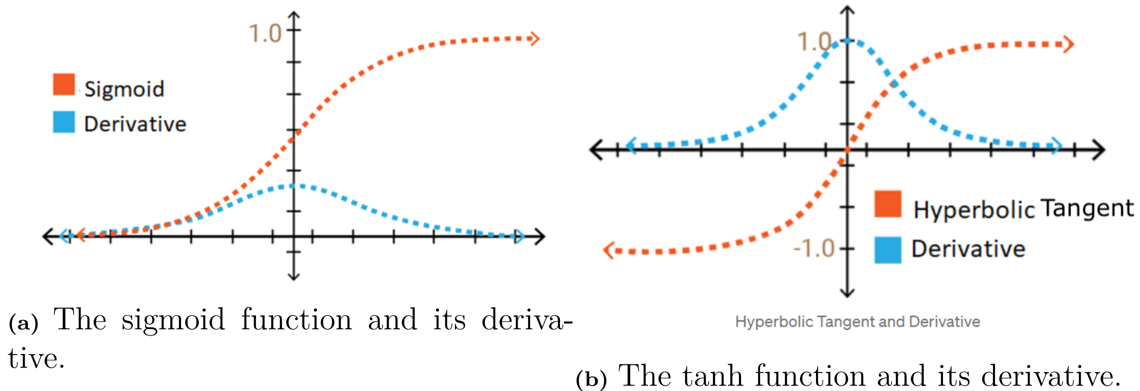
One simple and commonly used activation function is the linear activation function, which can be described with equation 2.6. The function multiplies the input by the weights for each neuron, creating an output signal that is proportional to the input, meaning that no transform is applied at all [29].

$$F(x) = ax \tag{2.6}$$

The derivative of the function in equation 2.6 is not zero, however it is a constant value a independent of the input. This implies that the same value will be used for updating the weights and biases during the backpropagation step, meaning that the losses will not improve. A network that uses linear activation functions is easy to train, however it will also be unable to learn complex patterns from the data. Therefore, the linear activation function is most commonly used in the output layer for problems with the objective to predict a quantity, such as in regression problems [29].

In order for the neuron to be able to learn more complex structures in the data, the activation functions in the hidden layers are preferably nonlinear activation functions. Two widely used activation functions are the sigmoid and the hyperbolic

tangent (tanh) function. The sigmoid function transforms any input value into an output range of $(0, 1)$, so that x-values smaller than -2 are transformed to values close to 0 and x-values larger than 2 are transformed to values close to 1. The tanh function is a similar shaped nonlinear activation function that transforms input values in a similar manner into an output range of $(-1, 1)$.



(a) The sigmoid function and its derivative.

(b) The tanh function and its derivative.

Figure 2.3: The sigmoid and tanh function with their derivatives, illustrating the vanishing gradient problem from [30].

As seen in Figure 2.3a, the derivative values at the edges of the graph are very small and converges to 0. In Figure 2.3b, it is possible to see that the derivative is a bit more steeper than in Figure 2.3a, which might lead to a slightly better performance. However, one can see that the gradient also converges to 0. This is a problem called the vanishing gradient problem [29]. This problem occurs when the gradients are very small, since small gradients lead to the weights and biases of the early layers not being updated accordingly at each training step. These early layers are important in the process of recognizing important features of the input data, which therefor leads to an overall inaccuracy in the network. This is especially a problem in deep neural networks [29].

In order to solve the vanishing gradient problem, it is necessary to find an activation function who does not give rise to small derivatives. The Rectified Linear Unit (ReLU) activation function is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero, which is described by equation 2.7 [29].

$$ReLU(x) = max(0, x) \tag{2.7}$$

From Figure 2.4, one can see that the gradient does not take on small values and therefor will not converge to 0. This makes the ReLU function an appropriate choice for the hidden layers, since it is computationally efficient as well as that it acts and looks like a linear function, but can however capture complex patterns. Most importantly, the vanishing gradient problem can be avoided by using ReLU since the gradients remain proportional to the node activations [29].

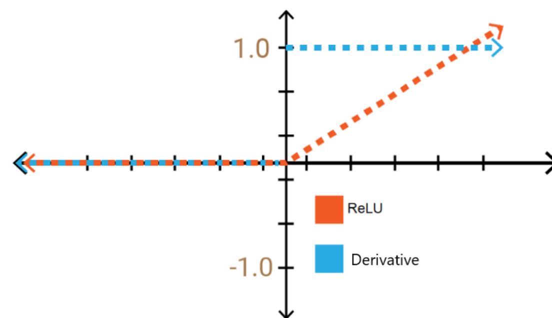


Figure 2.4: The ReLU function and its derivative from [30].

There is however a problem with ReLU referred to as the dying neuron problem, which results in ReLU neurons becoming inactive. This is a scenario when a large number of ReLU neurons only output values of 0, i.e. a large number of inputs to the ReLU neurons are negative. This problem can however be solved by using different variations of ReLU, that for example adds a slight slope in the negative range. Another solution is to modify initialization procedure of the weight and biases in the ReLU neurons [31].

2.2.1.2 Initialization of Network Parameters

The initialization of the neural network weights and biases can be crucial for the model's ultimate performance, in terms of accuracy and speed [32], [33], [34].

If not initializing the weights correctly the deeper layers might receive inputs with small variances. The activation outputs from each layer may hence result in either vanishing or exploding gradients, i.e. too small or too large gradients, which in turn disturbs or retards the overall convergence process of the network. The aim is therefore to find an appropriate weight initialization which ensures that the variance from the outputs of the different layers are approximately the same [33].

One commonly used weight initializer is the Random Uniform initializer, which generates tensors with a uniform distribution $\in (0, 1)$ [35]. The idea of using random uniform initialization is based on the fact that the weight vector is multiplied with the input vector (equation 2.5). It can therefore be a good idea to keep the values in a small range, however not in a range that would lead to unstable gradients. Another reason for the random uniform initialization is based on the randomness of the initialization. With a constant weight initialization, all the activation outputs from each layer of a network during the initial forward pass will be the same, which is referred to as the symmetric problem [36] and complicates the updating of the weights for the network [37].

The random uniform initialization does however not ensure stable gradients even if it might be better than the constant weight initialization. A popular initialization for multilayered FFNN when using the rectified linear unit (ReLU) as an activation function 2.2.1.1, is the He weight initialization. The He initialization method is

calculated as a random number with a Gaussian probability distribution, a mean of 0 and a variance of $\frac{2}{n}$, where n is the number of input units in to the weight tensor. This method is effective since it tries to keep the variance of all the layers approximately equal as well as takes the non-linearity of activation functions into account. This can be seen when observing the variance, since the weights are initialized by keeping the size of the previous layer in mind, resulting in a controlled initialization, i.e. a faster and more efficient gradient descent [33] [38].

In the case of using linear activation functions, such as the tanh or sigmoid, the Xavier uniform initialization is an appropriate choice. The aim of this initialization is to initialize the weights in a similar way as in He initialization, i.e. by keeping the variance of the activations approximately equal across every layer. By enabling this constant variance, the unstable gradient problem will be reduced. In the Xavier uniform initialization, the biases are initialized as 0 and the weights W_{ij} at each layer are initialized as in equation 2.8 [33].

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \tag{2.8}$$

In equation 2.8, U is a uniform distribution and n is the size of the previous layer.

When dealing with non-linear neural networks such as the Recurrent Neural Networks (RNNs), explained in Section 2.2.2, the orthogonal weight initialization has been found to be successful. One problem with the RNNs is the vanishing and exploding gradients. A method to combat this is for example by using LSTM layers, explained in Section 2.2.2.1, and another method is to use an orthogonal initialization for the weights. As seen in Appendix 3 and explained in Section 2.2.1.4, the gradient is calculated through repeated matrix multiplication. This procedure might lead to a vanishing or exploding gradient, i.e. no training occurs due to no information being backpropagated or training does not converge due to highly fluctuating gradients. However, the eigenvalues of orthogonal matrices have the property that their absolute value always is 1. This means that when using orthogonal matrices to initialize the weights, this property will result in matrices that do not explode or vanish, i.e. such an initialization will thereby lead to stable gradients [39].

In ANNs, the bias can be thought of as having the same role as a constant in a linear function, enabling the straight line to be transposed by the value of the constant. In a neural network the bias instead enables a shift of the activation function with a constant value, allowing the model to be flexible in order to fit the data [40]. The biases are most commonly initialized as zeros, since the symmetric problem can be taken care of by the small random numbers in the weights. When initializing biases in neurons with ReLU activation, it is sometimes preferred to use small constant values (e.g. 0.01) as bias. This is done in order to prevent the dying neuron problem explained in Section 2.2.1.1, by ensuring that all ReLU neurons are active at the beginning. This type of initialization has however been resulting in inconsistent results, which is why the biases are more commonly initialized as zeros [34], [36].

2.2.1.3 Loss Functions

The neural networks are trained with an optimization algorithm which aims to minimize the loss of the model. This requires that the choice of loss function matches the aim of the specific problem, such as classification or regression as well as that the configuration of the output layer matches the chosen loss function. Problems that involves predicting real-valued quantities are called regression problems [26].

One frequently used loss function within regression problems is the Mean Square Error (MSE) loss. MSE is defined as the average of the squared differences between the actual and predicted values, which can be seen in equation 2.9.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} \tag{2.9}$$

In the equation, n is the total number of data points, y is the actual values and $\hat{y}$ is the predicted values. One problem with the MSE can however be that it is sensitive to outliers and is therefore only good to use if the target data is normally distributed [41].

In regression problems where one do not want outliers to play a crucial role, the Mean Absolute Error (MAE) loss should be used. MAE is more robust to outliers than the MSE and is defined as the average of the differences between the actual and predicted values, which can be seen in equation 2.10.

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n} \tag{2.10}$$

However, a problem with MAE is that the magnitude of the gradient only depends on the sign of $y_i - \hat{y}_i$, and not the size of the error. This means that the gradient magnitude can be large even for small errors which can lead to missing minima at the end of training and thereby inhibiting convergence [42].

There can also be cases when neither MSE or MAE gives desirable predictions due to their flaws. Then the Huber loss should be used. The Huber loss is a combination of the MSE and the MAE in that sense that it is less sensitive to outliers in the data as well as that it is able to curve around the minima, which reduces the gradient. The Huber loss can be seen as an absolute error that becomes quadratic when the error is small. The threshold for the error to become quadratic depends on parameter δ, which can be fine-tuned. When $\delta \to 0$ the Huber loss advances towards MSE and when $\delta \to \infty$ it advances towards MAE, which can be seen in equation 2.11.

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases} \tag{2.11}$$

The Huber loss therefore uses good properties from both MSE and MAE, as well as make up for their less advantageous properties. A problem with this loss function is however that the hyperparameter δ needs to be tuned iteratively [43].

2.2.1.4 Backpropagation

Backpropagation is a commonly used algorithm in FFNN for calculating the gradient given a loss function $J(t, y(x))$, where $y(x)$ is the predicted output and t is the target output. Using forward propagation, an output is generated by the neural network. This output is then used for obtaining the scalar loss $J(t, y(x))$ which then propagates through the network backwards retrieving the gradient of the loss w.r.t. each of the parameters (weights and biases) in the network by the chain rule. The gradient is computed one layer at a time, iterating backwards, i.e. starting from the last layer and stopping at the first. The term backpropagation therefore refers to the neurons being updated forward and the errors being updated backwards [26].

The foundation of the backpropagation is built on the chain rule, which is a method for calculating the derivative of the composition of two or more functions. The chain rule is a single step of the backpropagation algorithm [26]. The pseudocode for the forward and backward propagation can be found in Appendix 2 respectively 3.

2.2.1.5 Training and Optimization Algorithms

Backpropagation is as stated in the above Section used for calculating the gradients efficiently. Optimizers on the other hand, are used for training the neural network, using the gradients computed with backpropagation in order to update the parameters and minimize the loss.

One of the most popular and commonly used algorithms to perform optimization is Gradient Descent. The most naive implementation, Batch Gradient Descent (BGD), minimizes an objective function (loss function) J(θ) parameterized by a model's parameters $w, b \in \theta$. This is done by updating the parameters in the opposite direction of the gradient of the objective function w.r.t. to the parameters, until reaching convergence, which can be seen in equation 2.12.

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta). \tag{2.12}$$

The learning rate α determines the stepsize one takes to reach a (local) minimum. One problem with BGD is however that the convergence to a local minimum might be very slow when dealing with large datasets, since BGD computes the gradient using the whole dataset. Another problem is that there is no guarantee that the algorithm will find the global minimum when there are several local optima and non-convex surfaces [44].

One other variant of Gradient Descent is the Stochastic Gradient Descent (SGD). Instead of computing the gradient using the whole dataset, SGD uses random subset selections of data, which can reduce the computation time. This means that SGD executes a parameter update for each training example $x^{(i)}, y^{(i)}$ instead of the entire dataset, as seen in equation 2.13.

$$\theta = \theta - \alpha \cdot \nabla_\theta J\left(\theta; x^{(i)}; y^{(i)}\right). \tag{2.13}$$

SGD performs updates with a high variance that cause the objective function to fluctuate a lot. The heavy fluctuation that is encountered in SGD enables the algorithm to find potentially better local minima. However it might also lead to overshooting, which makes it difficult to find the exact minimum [44]. SGD therefor has a hard time orienting in spaces where the surface curves are much more steep in one dimension than in another, which is common in areas that surrounds a local optima [45]. By introducing momentum into the algorithm, it is possible for SGD to increase the speed of convergence by damping oscillations and accelerate in to the wanted direction. This is done by adding a momentum term γ, which is usually set to 0.9. As seen in equation 2.14, the modification of the update vector v_t at the current time step depends on both the current gradient and the change in the update vector in the previous step v_{t-1} [46].

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_\theta J(\theta), \\ \theta &= \theta - v_t. \end{aligned} \tag{2.14}$$

This method is inspired by classical mechanics of motion such as letting a ball roll down a hill, gathering momentum and increasing its speed until reaching its final velocity (if $\gamma < 1$). This is applied to the parameter updates by enabling the momentum term to increase the updates for dimensions whose gradients point in the same directions and to reduce updates for dimensions whose gradients point in the opposite direction, indicating that γ decides how quickly the impact of previous gradients will disappear. This results in an accelerated convergence as well as decreased oscillations [44].

One of the most widely used optimization algorithms for stochastic gradient descent for training deep neural networks is however the Adaptive Moment Estimation (ADAM) optimization algorithm. In order to not get stuck in a local optima, ADAM uses the estimates of the first and second moments of the gradients in order to adapt individual learning rates for each parameter of the neural network. This means that it adapts the learning rate of parameters based on the averaged first moment (the mean), as well as using the second moments of the gradients (the uncentered variance). The momentum thereby enables the algorithm to pass local optima and accelerate into better ones. This can be visualized in equation 2.15, where ADAM stores an exponentially decaying average of past gradients (like in momentum) as well as keep an exponentially decaying average of past squared gradients [47].

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\, g_t, \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2. \end{aligned} \tag{2.15}$$

In equation 2.15 $g_t = \nabla_{\theta_t} J(\theta_t)$, m_t is the estimates of the first moment and v_t is the estimates of the second moment of the gradients. β_1 and β_2 are the exponential decay rates of the first and second moment estimates and are commonly set to be close to 1. Based on equation 2.15, both m_t and v_t are initialized as vectors of zeros, leading to the moment estimates having a tendency to be biased towards zero. This problem is however taken care of by computing bias-corrected first and second moment estimates, such as in equation 2.16.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t},$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{2.16}$$

These bias-corrected estimates are then used in the ADAM update rule, in equation 2.17.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t. \tag{2.17}$$

In equation 2.17, ϵ is a small positive constant in order to avoid division by zero and is commonly set to 10^{-8}. Another big advantage of ADAM is that it is relatively insensitive to the choice of hyperparameters, making the fine-tuning less complicated. This makes ADAM a suitable optimization algorithm for problems with sparse gradients and non-stationary settings [47]. The pseudocode for the ADAM algorithm can be found in Appendix 1.

2.2.2 Recurrent Neural Network

A FFNN can only take in an input vector of a fixed size. This is a shortcoming for the network in that sense that it inhibits the usage of sequenced input data with no predetermined size. In sequenced data, one single input of that sequence is related to the other inputs and therefor they will depend on each other. It is thereby necessary to find a network that can capture this relationship, since feed-forward networks are not able to [48].

When feed forward neural networks include feedback connections, i.e when a feedback loop is inserted in to the network, then they are called Recurrent Neural Networks (RNNs). RNNs therefor allow information to be preserved over a sequence of a sample, by having an internal memory. In difference to the feed forward neural networks, RNNs remember what they have learned from previous inputs while simultaneously generating outputs. The outputs are in turn not only determined by weights that are applied to the input but also by a hidden state that is based on prior inputs/outputs, which are further used to process the next input [48].

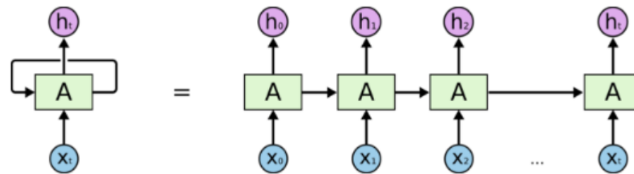


Figure 2.5: An unrolled Recurrent Neural Network from [48].

In Figure 2.5 it is possible to see clearly how the feedback connections occur. The input x_0 from a sequence of inputs is fed in to the network, outputting the hidden state vector h_0, which then together with x_1 constitute the input for the next step

and so on. By proceeding in this way, the relationship between the inputs is remembered during training. The equation for the current state and output can be seen in equation 2.18.

$$h_t = \phi\left(w_{hh}h_{t-1} + w_{xh}x_t\right), \\ \mathrm{y_t} = \mathrm{w_{hy}h_t}. \tag{2.18}$$

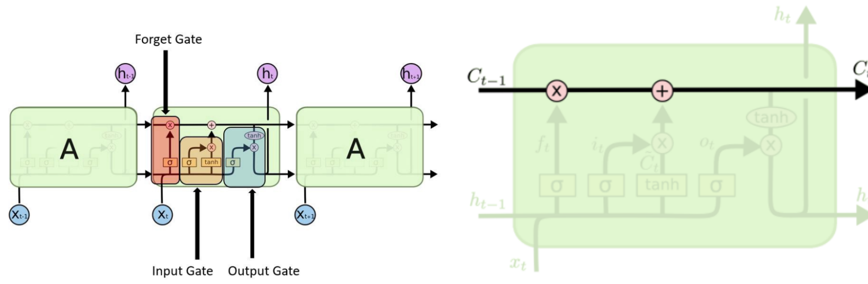
In equation 2.18, h_t is the hidden state vector for input x_t, ϕ is the activation function, w_{hh} is the weight for the previous hidden state, w_{xh} is the weight at the current input state, y_t is the output state and w_{hy} is the weight at the output state.

A problem with RNN can however appear through training. Due to the recurrent nature of the network, one needs to use backpropagation through time (BPTT), instead of the backpropagation used for feed forward neural networks. BPTT means that the RNN is unfolded over time before backpropagation is used to calculate the gradient, making it into a deep neural network [49]. These unrolled networks tend to be very deep, which often results in the either the vanishing gradient problem, explained in Section 2.2.1.1, or the exploding gradient problem. The exploding gradient problem arises when large error gradients accumulate, resulting in large updates of the network weights during training, which in turn results in an unstable network that is unable to learn accordingly [50]. The vanishing and exploding gradients are commonly occurring in RNNs due to the difficulty of capturing long-term dependencies. This challenge arises since the gradient in early layers is the product of gradients from all the later layers, which can be exponentially decreasing/increasing w.r.t. the number of layers. The more layers, the longer the product becomes, resulting in an unstable gradient [51].

2.2.2.1 Long Short-Term Memory

One way to take care of the exploding gradient problem is through gradient clipping. Gradient clipping is a method that rescales the values of the gradients with a threshold value, i.e. the gradient is either clipped or set to that threshold value if the gradient is larger than that threshold. By doing so the gradients are prevented from increasing exponentially [50].

Another way to deal with the unstable vanishing and exploding gradients is by using the Long Short-Term Memory (LSTM) networks. LSTM networks are modified versions of RNNs that are able to capture long-term dependencies in sequenced data. The network is then trained through backpropagation [52]. There are three different gates in a vanilla LSTM network cell: the forget gate, the input gate and the output gate, which can be seen in Figure 2.6a. The cell state of a LSTM network cell is vizualised in Figure 2.6b as the horizontal line that passes through the top of the cell.



(a) The forget, input and output gate in a LSTM cell from [48].

(b) The previous C_{t-1} and current C_t cell state of a LSTM network cell from [53].

Figure 2.6: The structure of a LSTM cell network with the different gates and cell state.

The forget gate is responsible for deciding which information that should be ignored, and which that is important for the cell state. This is done by passing the information from the current input x_t and hidden state from previous time step h_{t-1} through the sigmoid function. The sigmoid function in turn returns a number between 0 and 1 for each number in the cell state, where 0 means to discard everything and 1 means to keep everything. The equation for the forget gate is presented in equation 2.19, where σ is the sigmoid function, W_f is the weight matrix between the input and forget gate and b_f is the bias [48], [53].

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right). \tag{2.19}$$

In the next step, the input gate i_t determines which information that will be stored in the cell state by first deciding which values to update through a second sigmoid layer and secondly with a tanh layer that outputs a vector of new candidate values $\tilde{C}_t$. These values are weighted after their importance in a range between $(-1, 1)$, where -1 equals being less important [53].

$$\begin{aligned} i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right), \\ \tilde{C}_t &= \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right). \end{aligned} \tag{2.20}$$

In the next step, the previous cell state C_{t-1} needs to be updated into the current one C_t. The previous cell state is therefore multiplied by the forget gate f_t in order to forget irrelevant information. The product is thereafter added to $i_t * \tilde{C}_t$, resulting in new candidate values scaled by their importance, i.e. by how much each state value was updated [53]. This can be seen in equation 2.21, where $*$ stands for element-wise multiplication.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t. \tag{2.21}$$

Lastly, the output gate of the LSTM cell decides what to output. The output will be based on the current cell state, however a filtered version of it with a sigmoid layer, as in equation 2.22.

$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right). \tag{2.22}$$

In order to get the final output the current cell state C_t is passed through a tanh function and then multiplied with the output of the sigmoid layer o_t element-wise, as seen in the equation 2.23 [53].

$$h_t = o_t * \tanh(C_t) . \tag{2.23}$$

The LSTM thereby solves the problem of unstable gradients in order to capture long-term dependencies by being able to protect and control the cell state using the different gates.

Due to the structure of a LSTM cell, each cell contributes with many learnable parameters to the network [54]. With a large amount of learnable parameters, the network has a great capacity to learn the features of the input data. However, this also makes the network prone to overfitting [55], which means that it learns too much of the details during training. This will further make the network perform worse on new unseen data [56]. In order to prevent overfitting, one can add dropout layers to the model between existing layers. The dropout is thereby applied to outputs of the prior layer that are fed into the next layer. Dropout works by randomly and temporarily "dropping out" or removing neurons in the network during training, by setting the value of them to zero. Dropout also uses a dropout rate, which is a tuneable parameter that decides the probability of randomly dropping out neurons in the network. [57].

2.3 Q-Learning

Q-Learning (QL) is one of the major sub-fields within model-free Reinforcement Learning (RL). Model-free RL algorithms searches for the optimal policy by trial-and-error experience and does therefore not need full information of the model [58].

The QL algorithm provides the agent with the capability of learning how to act optimally in a Markovian domain i.e. a problem modeled as a MDP [59], which is explained in Section 2.1.1. The agent learns by evaluating the immediate reward obtained from taking a certain action in a certain state, together with an estimate of the value of the state it ends up in after executing the action. Equation 2.24 defines the value of state s given a policy π [59].

$$V^{\pi}(s) = \Re_s(\pi(s)) + \gamma \sum_j P_{s,j}[\pi(s)]V^{\pi}(j), \tag{2.24}$$

where $\Re_s(\pi(s))$ is the expected immediate reward the agent receive after taking action a, i.e the mean of the immediate reward $r(s,a)$. $V^{\pi}(j)$ is the value of the next state j which the agent moves to with transition probability $P_{s,j}[\pi(s)]$. The discount factor $\gamma \in (0,1)$ is multiplied with the cumulative future rewards $\sum_j P_{s,j}[\pi(s)]V^{\pi}(j)$, and describes how important future rewards are to the current state [59]. With a $\gamma = 1$, the agent will choose its actions based on the cumulative future rewards while

with a $\gamma = 0$ it will instead choose actions based on only the immediate reward.

After trying all possible actions in all possible states repeatedly, the agent has determined the optimal policy π^* which is the optimal sequence of actions in terms of the total discounted reward. The value of sate s following policy π^* is defined in equation 2.25 [59].

$$V^{\pi^*}(s) = \underset{a}{max}\{\mathfrak{R}_s(a) + \gamma \sum_j P_{s,j}[a] V^{\pi^*}(j)\}. \tag{2.25}$$

As mentioned earlier, the QL algorithm does not require full information of the model to find the optimal policy. The objective is instead to obtain $V^{\pi^*}(s)$ and $\pi^*(s)$ without knowing $\mathfrak{R}_s(a)$ or $P_{s,j}[a]$ [59]. This is done by estimating the expected discounted reward for taking action a in state s and following policy π thereafter. Equation 2.26 defines the equation for these estimates which are called Q-values.

$$Q^{\pi}(s,a) = \mathfrak{R}_s(a) + \gamma \sum_j P_{s,j}[\pi(s)] V^{\pi^*}(j). \tag{2.26}$$

The agent's experience consists of a sequence of distinct stages, denoted as epochs from now on. These epochs are the equivalents of the time steps described in Section 2.1.1. In an epoch n, the agent:

1. observes the current state s_n,
2. executes action a_n,
3. evaluates action by receiving an immediate reward r_n,
4. observes the next state $j_n = s_{n+1}$,
5. adjusts the previous epoch's Q-values with a learning rate α_n.

The immediate reward r_n that the agent receives after executing action a in state s is probabilistic with mean $\mathfrak{R}_{s_n}(a_n)$. To obtain the optimal Q-values, $Q^{\pi^*}(s,a)$, for action a in state s, the agent has to execute action a in state s repeatedly and each time adjust the estimate of the corresponding Q-value. The adjustment of the Q-values follows equation 2.27, which is derived from the Bellman equation.

$$Q_{n+1}(s,a) = \begin{cases} Q_n(s,a) + \alpha_n[r_n + \gamma V_n(s_{n+1}) - Q_n(s,a)], & \text{if } s = s_n \text{ and } a = a_n \\ Q_n(s,a), & \text{otherwise.} \end{cases} \tag{2.27}$$

In equation 2.27, $V_n(s_{n+1})$ corresponds to the value of state s_{n+1} assuming the agent follows the current best policy thereafter, thus $V_n(s_{n+1}) = \underset{a_{n+1}}{max} Q_n(s_{n+1}, a_{n+1})$. Only Q-values corresponding to state-action pairs visited in epoch n are adjusted. All Q-values are stored in a table, with one Q-value for each state-action pair. In order for each state-action pair to have an individual position in the table, both state and action values have to be discrete and of low dimension. The optimal Q-values for action a in state s, $Q^{\pi^*}(s,a)$, are obtained when equation 2.27 has converged [59].

In order for the agent to find the optimal policy π^* by estimating the optimal Q-values $Q^{\pi^*}(s,a)$, it has to both explore the environment to discover new states and revisit already visited states to adjust the estimated Q-values. If the agent follows the greedy policy, i.e. choose actions it thinks will yield the highest expected future reward, $\underset{a_{n+1}}{max} Q_n(s_{n+1}, a_{n+1})$, it will most of the time revisit already visited states and not discover as many new states. It might therefore end up with a sub-optimal solution.

Instead of only following the greedy policy, one can use a so called ϵ-greedy algorithm which handles the trade-off between exploration and exploitation [60]. When exploring, or discovering new states, the agent chooses what action to execute randomly. The agent exploits when it follows the greedy policy since it then uses available information to choose an action. The trade-off between exploration and exploitation is handled by the probability ϵ [60]. The pseudocode for the ϵ-greedy algorithm is presented in Appendix 4.

2.3.1 Policy Optimization

Another model-free approach can be to use a policy optimization algorithm, such as a policy gradient algorithm [61]. The objective of a policy gradient algorithm is to maximize the expected reward directly by taking small steps in the direction of the policy gradient. The policy gradient is the derivative of the expected reward with respect to the policy parameters.

2.4 Deep Q-Learning

A disadvantage with the vanilla QL described in Section 2.3 is the Q-table, which requires a discrete and low dimensional state-action space. When solving problems that approaches real-world complexity, a low dimensional state-action space is not frequently encountered [62]. In order to be able to use QL in such situation, one has to derive an efficient representation of the environment and without losing too much information, decrease the dimensions of the state-action space. This can be done by for example discretizing state and action values or excluding some state parameters from the state representation [2].

2.4.1 Deep Q-Network

Another solution is to use a so called Deep Q-Network (DQN) [62], which uses a Deep Neural Network (DNN) to approximate the Q-values instead of storing them in a Q-table. Without the Q-table, one is not restricted to have a low dimensional and discrete state-action space since the network can take high-dimensional continuous data as input. The DQN takes the current state as input and returns estimates of the Q-values for all possible actions. The method of combining a DNN with QL is called Deep Q-Learning (DQL) [63]. In Figure 2.7 the difference between QL and

DQL is visualized.

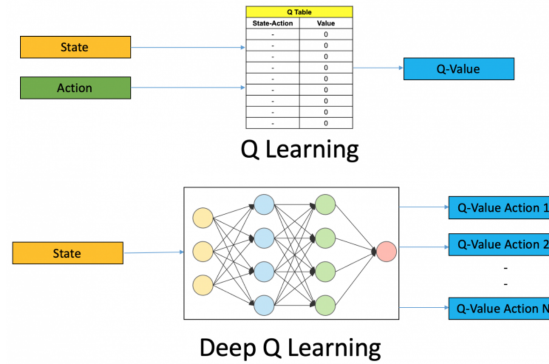


Figure 2.7: Visual comparison of Q-Learning and Deep Q-Learning from [63].

The DNN in the visualisation of the DQL-model in Figure 2.7 is a fully connected multi layer feed-forward neural network, which is described in detail in Section 2.2.1. This is also the kind of network that is referred to when mentioning DNN in this Section.

The objective is the same for DQL as for QL, i.e to find the optimal policy π^* by estimating the optimal Q-values, $Q^{\pi^*}(s,a)$. However, instead of iteratively adjusting the estimates until the optimal ones are received, the optimal Q-values are received by iteratively updating the network parameters in order to minimize a loss function [62]. How to choose an appropriate loss function for the specific problem can be read about in Section 2.2.1.3. The network parameters are updated and the loss function is minimized using an optimization algorithm, see Section 2.2.1.5.

2.4.1.1 Target Network

As described in Section 2.2.1, the DNN want's examples of known outputs, or target outputs, when building a model to predict unknown outputs for a given input. Using a DQN, this would mean that optimal Q-values should be known for each state. This is however not the case since the aim is to estimate these values. Instead, one estimates both the prediction and target output using the DNN, with the target output being as in equation 2.28

$$t_n = r_n + \gamma \underset{a_{n+1}}{max} Q_n(s_{n+1}, a_{n+1}; \phi), \tag{2.28}$$

and the prediction being $y_n = Q_n(s_n, a_n; \theta)$. The parameters θ and ϕ represents the parameters of the prediction respectively the target network. A problem with this is however that the target output is re-estimated in each training iteration. Using both continuously changing predictions and target outputs to update the network parameters can easily cause unstable training [62]. To overcome this instability problem, one can use two networks instead of one, where one estimates the prediction and

the other estimates the target output. These networks should have the same architecture, but the parameters of the network estimating the target output, often referred to as target network, should be frozen [62]. The parameters of the network estimating the predictions, sometimes referred to as online or prediction network, are copied over to the target network, either all at once every C training iterations or smoothly each training iteration [64], [62]. The smooth update follows the rule in equation 2.29. The parameter $\tau \in (0, 1)$ regulates how much of the parameters from the prediction network that should be copied over to the target network each training iteration.

$$\phi \leftarrow (1 - \tau)\phi + \tau\theta. \tag{2.29}$$

With a τ equal to 1, all parameters from the prediction network are copied over to the target network in each iteration [64], i.e. the parameters in the target network are the same as the parameters in the prediction network. The idea of using another network to predict the targets is however to stabilize the training by not having the same parameters in both networks. One does therefor not want to update the target network too much every iteration, i.e not use a too large τ. In contrast, with a τ equal to 0, none of the target network parameters are updated in an iteration, meaning that the initial parameters are kept during the entire training. A too small τ does therefor slow down the learning and might prevent the algorithm to converge to an optima since the target network parameters will rarely be updated [64]. This implies that τ is a hyperparameter that has to be tuned for each problem.

2.4.1.2 Experience Replay

In the approach of QL described in Section 2.3, the Q-values are updated after each epoch, i.e on each experience in sequential order. However, since the DNN in DQN uses supervised learning, it assumes that the inputs are independent of earlier outputs and comes from an identical distribution. These conditions are violated when training with the data in sequential order, due to the data being correlated. Another reason for these conditions being violated is that the agent learns simultaneously as exploring. This results in data that will not come from an identical distribution [65]. To avoid this, one can store each experience in a buffer and then sample randomly from the buffer. This method is called experience replay [66]. In this way, correlations within the data that comes from the agent's sequenced experience are avoided. Ensuring that the data is no longer correlated is also essential for some optimization algorithms, such as SGD which requires independent and identically distributed data in order to assure convergence [67]. Using experience replay also improves the sample efficiency of the algorithm as it enables data to be reused multiple times instead of being thrown away immediately [68].

The replay buffer is a fixed-size buffer that holds experiences collected by the agent in the most recent epochs. Data corresponding to one experience, i.e one epoch n, includes the current state s_n, the chosen action a_n, the obtained reward r_n and the

state the agent ends up in after executing the chosen action s_{n+1}. The buffer is typically implemented as a pipe, in which the oldest experience is removed when a new one is collected [68].

The size of the batches that should be sampled from the replay buffer in each training iteration is a hyperparameter that has to be tuned for each problem. The same holds for the number of epochs between each training iteration [69]. The pseudocode for the DQL algorithm is presented in Appendix 5.

2.4.1.3 Double Deep Q-Network

It has been shown that the implementations of both QL (Section 2.3) and DQL (Section 2.4) overestimates Q-values under certain circumstances [70], [71]. In the beginning of the Q-learning process, the current Q-values might not be representative of the optimal ones. To use the maximum of the current Q-value estimates, $\underset{a_{n+1}}{max} Q_n(s_{n+1}, a_{n+1})$, when updating the Q-values as in equation 2.27, can introduce a maximization bias in the learning for actions [70]. This can occur since the samples used to decide the best action. i.e. the highest expected rewards, are the same as the ones used to estimate the action values. Therefore, this enables the agent to choose the non-optimal action in a given state since it has the maximum Q-value, due to the overestimation problem [70].

To overcome this problem in QL when using Q-tables, it is proposed to use the Double Q-Learning (Double QL) algorithm. The general idea of Double QL is to decouple the maximum operation in the Q-value update (equation 2.27) to action selection, and then separately evaluate the chosen action. This means that one uses two separate estimates of the Q-values, one for selecting the action using the maximum operation and one for evaluating the value of the chosen action [70]. The two Q-value estimates are denoted Q_A and Q_B. Further, the action selection follows $a^* = \underset{a_{n+1}}{max} Q_{A,n}(s_{n+1}, a_{n+1})$ and the action evaluation would follow $Q_{B,n}(s_{n+1}, a^*)$. Which of the two Q-value estimates Q_A and Q_B should be used for action selection and action evaluation is chosen randomly for each update iteration. For more detailed information about the Double QL algorithm, the reader is encouraged to read [70].

As mentioned in the first paragraph of this Section, the above described overestimation problem is a problem for DQL using DQN as well. The proposed solution in this case is to use Double Deep Q-Network (DDQN), which is inspired by the Double QL algorithm. When using a DQN, the maximum operation is done by the target network when estimating the target output as in equation 2.28. However, to separate the maximum operation to action selection and action evaluation, as proposed in the Double QL algorithm, the target output is instead defined as in equation 2.30 for a DDQN [71].

$$t_n = r_n + \gamma Q(s_{n+1}, \underset{a_{n+1}}{max} Q(s_{n+1}, a_{n+1}; \theta); \phi). \tag{2.30}$$

In equation 2.30, the parameters θ and ϕ indicates which network parameters were used to estimate the Q-values, with θ referring to the prediction network and ϕ the target network. The estimation of the prediction using the prediction network, as well as how the network parameters are updated, is unchanged from DQL [71].

2.4.2 Deep Recurrent Q-Network

Deep Q- Learning (DQL) and Deep Q-Networks (DQN) has been successfully applied to a variety of complex tasks. In Section 2.1.1, regarding fully observable MDPs, it has been shown that it is suitable to use DQN when there is full knowledge of the environment even if the transition and reward probabilities are unknown. This setup is however rare in real-world environments, i.e. it is not common that the full state of the environment can be provided or determined [72]. It is however more common that real-world environment dynamics are captured by a Partially Observable Markov Decision Process, also explained in Section 2.1.1, where only an observation about the current state is available instead of all information about it. It can therefor be appropriate to use a different method than the DQN, since it is necessary for the agent to register and remember what it has seen in the past and thereby learn a policy.

A Deep Recurrent Q-Network (DRQN) is similar to the DQN, except with one or several LSTM cells added into the architechture of the DQN. The LSTM layer will include recurrence over time, i.e. the output for the observation o_t from the network will rely on a history of formerly seen observations [52]. A DRQN is trained by feeding a sequence of observations in to the network. It is not longer possible to use random batches of experience as in DQN, due to the network aiming to understand sequential dependencies. However, in order to maintain the random sampling as well as take sequential data into account, sequences of experiences are stored and sampled randomly before being fed into the network [73].

Since the LSTM cell includes the information regarding the history of observations, only the current observation needs to be fed into the network for prediction. The network will then be able to change its output depending on the pattern of observations that is has collected. At the start of each new episode, the hidden state of the LSTM cell is zeroed in order to maintain the independence of batches between episodes [72]. By using a LSTM layer, it is thereby possible for the DRQN to identify which sequences of actions that provides the maximum reward and then perhaps be able to converge faster than the DQN.

3
Problem Formulation

The formulation of the Dynamic Stochastic Electric Vehicle Routing Problem (DS-EVRP) aims to find a route for an electric vehicle that can change it's route dynamically based on randomly upcoming customer requests. This problem formulation is based on the report [4] as well as the project report [2].

Based on these reports, the problem is formulated in that sense that the vehicle departs and returns to the same depot during a route. During the route two types of customer requests can appear, where all locations of the customers are known. Either deterministic requests, which are previously known or stochastic requests, which are received with a given probability during the route. It is further assumed that all requests are accepted by the vehicle. The vehicle is electric and the amount of energy to drive a distance is random with a known probability distribution. The battery capacity is lowered by the consumed energy. In the project report, the vehicle could plan dynamically for charging during the route when reaching a certain battery limit. However in this thesis, the battery limit is set to a high value which is sufficient for an entire route. The charging of the battery as well as the charging stations in the environment are therefor not taken into consideration.

In the following Sections the model of the DS-EVRP, the solution method and the deterministic re-optimization for it is presented. Further, modification to the previous work in form of deep reinforcement learning is introduced.

3.1 DS-EVRP modeled as a Markov Decision Process

The problem is modeled as a MDP and can be visualized as a graph of nodes. The nodes correspond to customers, charging stations and the depot, which are connected through a set of paths. The state parameters describing the state of the environment in epoch n are:

- the current vehicle node position, i_n,
- the battery level, b_n,
- current payload, u_n,
- set of active customer requests, $\bar{C}_n$,
- set of visited customers, $\tilde{C}_n$.

The state in epoch n, described by the above parameters, is further denoted as s_n. In s_n, the vehicle chooses which node to visit next among a set of accessible nodes. These nodes includes the active customer requests, if there are any. If no customer requests are left, the vehicle returns to the depot. As mentioned above, the battery limit is enough to hold for an entire route. The nodes corresponding to the charging stations will therefor never be included in the set of accessible nodes. Further, the node that the vehicle decides to visit next after i_n, is denoted j_n. When the vehicle leaves node i_n, epoch n is complete. The energy consumed to drive between node i_n and j_n is denoted e_{i_n,j_n}, and is subtracted from the battery level b_n after the transition. It is obtained using equation 3.1, where W is the trucks curb weight, u_n is the payload in epoch n and $\alpha_{i,j}$ and $\beta_{i,j}$ are energy coefficients obtained from the input data, see Section 4.1.

$$e_{i_n,j_n} = \alpha_{i_n,j_n}(W + u_n) + \beta_{i_n,j_n} \tag{3.1}$$

The energy $e_{i,j}$, is also the reward for choosing node j, starting from node i. The weight of the vehicle payload u_n is further updated when the vehicle has arrived to the new node j_n, by adding the weight of the collected goods. Finally, j_n is removed from the set of active requests $\bar{C}_n$, and added to the set of visited customers $\tilde{C}_n$. It is assumed that each customer can at most make one request. After K epochs, no new requests are received, where $n \leq K$.

For the MDP to be terminated, all customer requests must be finished and the vehicle has to be back at the depot. It is assumed that the MDP is fully observable, and the continuous parameters, battery and weight, makes the state space infinite.

3.1.1 Objective Function

The objective of the problem in this thesis is to minimize the expected energy consumption by planning the routes efficiently. The objective function is defined in equation 3.2, where e_{i_n,j_n} is defined in equation 3.1.

$$\min_{e} \mathbf{E} \sum_{n} e_{i_n,j_n}. \tag{3.2}$$

There are no constrains on the objective function, which makes the problem a non-constrained optimization problem.

3.1.2 Solution Method

A deterministic re-optimization approach was presented in the project report [2]. In order to solve the re-formulated problem, this approach is compared with a Deep Reinforcement Learning algorithm, more specifically Deep Q-Learning. This method is further used with a probabilistic rollout function instead of a deterministic one, as in the project report.

3.1.2.1 Deterministic Re-optimization

The deterministic re-optimization approach is used in order to solve the MDP. This is done by using the deterministic information collected from state s_n at epoch n, to take an action j_n by only taking current requests $\bar{C}_n$ into account to determine the route. In order to estimate the total energy consumption, the expected value of energy e is used.

In each epoch n, the best action j_n is determined through a probabilistic rollout function, the Ant Colony Optimization algorithm (explained in Section 2.1.3.1), which considers the current requests.

3.2 Modification to Previous Work

The following changes are made to the previous work from the project report [2]:

- Instead of the Q-learning based reinforcement learning method introduced in the report, a deep Q-learning based reinforcement learning method is introduced, including a deep Q-neural network.
- Since a neural network is used, consideration of the battery capacity by inserting charging stations into the route became a much more complex task. Therefor, the battery capacity is set to a high limit, preventing battery depletion and thereby not taking the safety constraints into account. This further implies that the charging stations in the environment are never used when a route is planned.
- The constrains on the objective function are changed in this thesis in comparison to previous work. Since the safety constraints are not longer taken into account, the risk of the battery being under a certain level is no longer a constraint on the problem, as stated in equation 3.2.
- In the report, the state space was discretized to be able to build the Q-tables. In this thesis however, the state space is continuous, due to the use of a neural network as a function approximator. Further, the state space is modified using different and fewer state parameters in the state space.
- The rollout function used in the project report was the Tabu search 2.1.3.2. In this thesis, another rollout function is implemented, the Ant Colony Optimization algorithm (ACO), which is further compared to the Tabu search.

3.2.1 Deep Reinforcement Learning

Deep reinforcement learning is the method used in order to forecast future requests and predict the energy consumption to improve decision taking. The networks used in the deep reinforcement learning method are the Double Deep Q-network (DDQN) and the Deep Recurrent Q-network (DRQN), where the implementations of them are further explained in Sections 4.4 and 4.5.

4

Implementation

The implementations made when conducting this thesis were highly based on the theory presented in Chapter 2 as well as the problem formulation described in Chapter 3. The code was written in Python, using Keras from the open-source software library TensorFlow when implementing the neural networks. Other libraries were used as well, including Numpy, math and random, among others.

Due to the time constraints when conducting this thesis, it was necessary to limit the tuning of different combinations of hyperparameters in the rollout functions as well as the parameters and architecture of the network. Therefore, the tuning was conducted solely on case number 1 with 10 customers from the input data, presented in Section 4.1, for both DDQN and Double-DRQN, in order to find the best model configuration. Thereafter, this configuration was used throughout the entire project, including when evaluating other implementations, such as the other cases with 20 customer.

In the following Sections the input data used in this project is described, followed by an explanation of the state configuration. Thereafter, the implementations of the ACO algorithm, the DDQN and Double-DRQN are presented. Finally, the implemented performance evaluation method is described.

4.1 Input Data

The problem can be solved for various numbers of customers, however the data that was used includes cases for 10 respectively 20 customers. There are 25 different cases for each of the two customer numbers. The data for each case includes parameters for the initial state, probabilities for new customer requests, parameters used to calculate the energy it takes to drive between nodes, and lastly a set of constants that defines some of the problem- and vehicle characteristics.

The state parameters included in the initial state are the same as the ones described in Section 3.1, i.e $s_0 = (i_0, u_0, b_0, \tilde{C}_0, \bar{C}_0)$. For all cases in the initial state s_0, the vehicle is in the depot, i.e $i_0 = 0$ and the payload is zero, i.e $u_0 = 0$. The vehicle is also initially fully charged, but the battery capacity differs for the two different numbers of customers. For the 10 customers cases, the battery capacity is $b_0 = 20$ kWh and for the 20 customers cases $b_0 = 25$ kWh. There are initially no customers visited, i.e $\tilde{C}_0$ is empty and the initial set of customer requests $\bar{C}_0$ is different for all

cases.

Each customer c has an individual probability of sending out a request in epoch n, $p_c n = \sqrt[n]{1 - p_c}$, where p_c is the total probability of sending out a request for customer c. p_c is different among customers and cases.

The energy coefficients, $\alpha_{i,j}$ and $\beta_{i,j}$ used in equation 3.1 includes information about for example speed, topography and traffic. The coefficients are derived from the experiments in [4]. These experiments uses customers that are located in the city center of Luxembourg, two charging stations, and one depot. The customers are located randomly, but the location of the charging stations, respectively the depot are the same for all cases. An additional energy parameter, denoted $matrixCost$, is further included in the input data. This parameter was used in the rollout function. The values in this parameter does not take variations in the payload into account, but only the average vehicle weight.

Finally, the following constants are included in the input data:

- the curb weight of the truck, $W = 10700$ kg
- the vehicle's total battery capacity, $B = 20$ kWh for the 10 customer cases respectively $B = 25$ kWh for the 20 customer cases
- the number of customers in the case, i.e $ncust = 10$ or $ncust = 20$
- the weight of the goods to be picked up, which varies among customers and cases
- the limit after which no new customer requests are received, $K = 5$ for the 10 customers cases respectively $K = 10$ for the 20 customers cases

4.2 State Configuration

All state parameters mentioned in Section 3.1, i.e $s_n = (i_n, u_n, b_n, \tilde{C}_n, \bar{C}_n)$, were used to model the problem in an epoch n. However, an evaluation of two different state representations that were fed into the DDQN, was performed. One of them includes all state parameters and the other includes only the current vehicle position i_n and the current weight of the vehicle $u_n + W$.

4.3 Rollout Functions

The ACO algorithm was implemented as described in Section 2.1.3.1. The number of ants was set to $n_ants = 5$ for the 10 customer cases and $n_ants = 10$ for the 20 customer cases. The maximum number of iterations was set to $n_iterations = 10$. Remaining parameters, i.e the evaporation rate ρ, the factor that control the importance of the pheromone α, and the visibility factor β were set to 0.5, 1 and 2 respectively. The pheromone level τ was initialized as $\tau_0 = \frac{n_ants}{D^{nn}}$, where D^{nn} is the energy consumed for the nearest neighbour tour. This problem aims to minimize the energy consumption and not the path length. Therefor, the parameter L_k in

equation 2.2 represents the energy consumed on the route constructed by ant k instead of the length of the route.

The Tabu search algorithm (Section 2.1.3.2) was implemented in the same way as in the project report [2], using the same hyperparameter setup as well. This setup was chosen in order to perform a comparative study accordingly.

4.4 Double Deep Q-Network

The environment of the problem was implemented as the MDP described in Section 3.1, with the state representation presented in Section 4.2. The Double Deep Q-Network (DDQN) was implemented as described in Section 2.4.1.3 as well as in appendix 5, however using equation 2.30 as target function instead. The replay buffer size was set to $M = 10^6$ and training was performed in each epoch. Furthermore, the size of the batch m that is sampled in each training iteration was varied between $32, 64$ and 96. For this problem, the discount factor (described in Section 2.3) was set to $\gamma = 1$, since the agent should choose its actions based on the cumulative future reward rather than the immediate reward.

Different configurations of the network architecture were implemented, with varying numbers of fully connected hidden layers and neurons in each layer. The following combinations of different numbers of hidden layers and neurons in each layer were implemented: $1, 2$ and 3 hidden layers respectively 9 and 32 neurons in each hidden layer. The number of neurons in the input and output layers were set to the number of states included in the state representation respectively to the total number of possible actions. The ReLU function was used as an activation function in each hidden layer as well as in the input layer. A linear activation function was used in the output layer.

Two different weight initialization methods were evaluated for the weights in the hidden layers, namely He weight initialization and Random Uniform weight initialization. The biases were initialized as zeros and the weights in the input layer were initialized using Xavier Uniform initialization. The target network parameters were updated smoothly using equation 2.29. Different values of the parameter τ were tested, namely: 0.01,0.05 and 0.1. To see if the network parameters converge with the different configurations, an average of each parameter in each layer was saved and plotted when the training was finished.

The two loss functions Huber loss and MSE loss were tested. The average episode loss was saved and plotted after the training. For the Huber loss, the following values of the δ hyperparameter were tested: 1, 30 and 50. Further, the ADAM algorithm was used as the optimizer, evaluated with the following values of α: $5 \cdot 10^{-5}$, 10^{-4} and $5 \cdot 10^{-3}$.

Finally, three different action selection procedures were implemented. These proce-

dures were either to solely use the ϵ-greedy algorithm, to solely use the ACO rollout function or a combination of them both. When the ACO algorithm is used for action selection, the most favorable action according to the algorithm is chosen as explained in Section 2.1.3.1. Similarly, the action selection process when using the ϵ-greedy algorithm is explained is Section 2.3.

When combining the two methods, the rollout function was used in the first $\frac{1}{3}$ episodes and the ϵ-greedy algorithm in the remaining ones. The ϵ-greedy algorithm was implemented as the algorithm in appendix 4. When choosing an action randomly, only actions that were included in $\tilde{C}_k$ and excluded from $\bar{C}_k$ could be picked. Similarly, when an action was chosen from the greedy policy, Q-values corresponding to actions that were included in $\bar{C}_k$ and excluded from $\tilde{C}_k$ were set to $-inf$. Furthermore, as charging of the battery was not taken into consideration, the Q-values corresponding to actions that are charging stations were also set to $-inf$. This was done in order to make sure that these actions will not correspond to the maximum Q-values and will thereby never be chosen.

The probability ϵ in the ϵ-greedy algorithm was initially set to $\epsilon = 1$. Furthermore, a decay schedule was implemented so that in each episode, ϵ decayed with 0.001. This decay proceeded until ϵ reached 0.01. When reaching this lower threshold, the decay schedule stopped at the constant value 0.01. When choosing an action using the rollout function, the most favorable action according to the rollout function was chosen, given the current state.

As previously mentioned, when evaluating the performance on cases with 20 customers, the best found configuration for case 1 with 10 customers was used.

4.5 Double Deep Recurrent Q-Network

The Double Deep Recurrent Q-Learning model was implemented as the native Double Deep Q-Learning model described in previous Section 4.4, but with two important modifications which will be described in this Section. First off, instead of the Double Deep Q-Network, a Double Deep Recurrent Q-Network was used. In a Double Deep Recurrent Q-Network the principle of using two different networks to overcome the bias maximization problem described in Section 2.4.1.3, is combined with the principle of Deep Recurrent Q-Networks described in Section 2.4.2. The neural network in this Double Deep Recurrent Q-Network is a combination of one or more LSTM-layers and one or more fully connected layers.

Different network architectures were implemented. Combinations of the number of LSTM-layers, fully connected layers, number of units in each LSTM-layer as well as the number or neurons in each fully connected layer were varied. The number of LSTM layers was either 1 or 2, and the number of units in each layer were $5, 10$ or 15. Further, the number of fully connected layers was either $0, 1$ or 2 and the number of neurons in each fully connected layer was either $9, 32$ or 64. The effect of using

dropout to avoid overfitting was also investigated by adding a dropout layer after the final LSTM layer. The following different dropout rates were tested: 0, 0.1, 0.2 and 0.5, where a dropout rate equal to 0 implies that no dropout is performed. The weights in the LSTM layers were initialized with the default initialization methods, the Xavier Uniform initialization for the weights applied to the inputs and Orthogonal initialization for the remaining weights in the LSTM cell. Further, the biases were also initialized as the default choice, i.e. as zeros in all gates except the forget gate which biases were initialized as ones. The default activation functions, described in Section 2.2.2.1, were applied in the LSTM cell.

The second important difference in this implementation is how the data was stored and sampled from the replay buffer. Since the LSTM expects to be fed with sequenced data, the data from each epoch in an episode was stored as a sequence in the experience replay buffer. Each such sequence contains all data corresponding to one complete route. In one training iteration a batch of sequences, i.e a batch of complete routes, was sampled randomly from the experience replay buffer. This means that a batch size of 8 will contain 8 complete routes that are each randomly drawn from the replay buffer. In contrast, the implementation of experience replay used in DDQN uses a batch of samples of independent actions instead. Since each sequence should contain data corresponding to an entire route, training was only performed when a route was completed, i.e when an episode was terminated. The following different batch sizes were tested: 2, 4 & 8.

When evaluating the performance on cases with 20 customers, the best found configuration for case 1 with 10 customers was used with the exception of using twice as many units in the LSTM layers.

4.6 Performance Evaluation

In order to get results that were comparable with those obtained in the project report [2], the same performance evaluation strategy was implemented in this work. The strategy is that one compares the performance of the deep Q-learning model with the performance of the rollout function, using the average total reward per route as a performance measure. The total reward for one route is the sum of all rewards obtained after executing actions in that route. This strategy was implemented as follows:

1. the deep Q-learning model was trained for E episodes,
2. S complete routes were simulated using the trained deep Q-learning model and the performance was measured by calculating the average total reward per route,
3. S complete routes were simulated using the rollout function and the performance was measured by calculating the average total reward per route,
4. the percentage difference between the performance of the two methods was calculated as in equation 4.1.

$$\text{difference} = \left(\frac{\text{average total reward deep Q-learning model}}{\text{average total reward rollout function}} - 1 \right) \cdot 100 \qquad (4.1)$$

Note that this strategy was repeated for every evaluation, i.e. the rollout function was re-run for every evaluation. This was done in order to take the stochastic nature of the process into account.

Since the objective of this problem is to minimize the expected energy consumption, as explained in Section 3.1.1, the reward is defined as the energy consumed to drive between nodes. This means that the lower reward, the better the model performs. As seen in equation 4.1, a negative value of the difference therefor indicates that the deep Q-learning model outperforms the rollout function and vice versa. Depending on which model that was evaluated and for how many episodes the rollout function was used during training, E and S were set differently.

5

Results

This Chapter presents the results obtained when trying to answer the research questions formulated in this thesis. Case 1 with 10 customers from the input data (Section 4.1) was used for training in all results, except the results presented in the case studies in Section 5.3.2, 5.3.1 and in Table 5.16. The reduced state representation, explained in Section 4.2, was used when obtaining all results except when comparing the reduced state representation to the full one in Section 5.1.1.1. The performance of the different networks and their configurations are evaluated using the difference in average total reward (equation 4.1) between the deep Q-Learning model and the rollout function, as well as by analyzing the convergence of the network parameters. The percentage difference was further used to compare with the obtained percentage differences in the project report [2]. Note that the lower the percentage difference, the better the model performance, as explained in 4.6.

In the following Sections, the results for the DDQN, the Double-DRQN, the case study as well as for the rollout functions are presented.

5.1 Double Deep Q-Neural Networks

In this Section, the results concerning the DDQN network architecture and hyperparameter tuning are presented. In all trials where nothing else is stated, the ACO algorithm was used as a rollout function throughout the entire training. Further, the network was trained for $E = 10000$ episodes and the performance was evaluated on $S = 5000$ simulations.

5.1.1 Network Architecture

Different network architectures were implemented and analysed in order to find the best setup. In Table 5.1, one can see the obtained results from using a network with varying numbers of hidden layers as well as neurons in each hidden layer. The hyperparameters were set to the following values: $\alpha = 10^{-4}$, $\tau = 0.05$ and the batch size was set to 32. Further, the weights in the hidden layers were initialized using He-initialisation, the biases were initialized as zeros and MSE was used as loss function.

Table 5.1: This table presents the difference for a DDQN configuration with varying number of hidden layers as well as neurons in each hidden layer.

Difference	Nr hidden layers	Nr neurons
9.00 %	1	9
16.81 %	1	32
8.42 %	2	9
16.90 %	2	32
8.30 %	3	9
8.55 %	3	32

From the results in Table 5.1, it can be observed that the overall best performance was obtained with 9 neurons in each layer. The results do however not indicate a big difference in performance for the different numbers of hidden layers with 9 neurons in each layer. The following trials were therefore chosen to be evaluated using a network with 2 hidden layers and 9 neurons in each layer. The run-time for the chosen configuration was 3.5 hours.

In Figure 5.1, the average episode loss over all training episodes is presented for the network with 2 hidden layers and 9 neurons in each layer. As seen in the figure, the loss decreased quickly and thereafter converged.

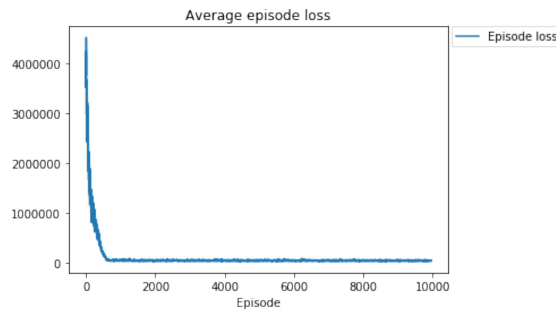
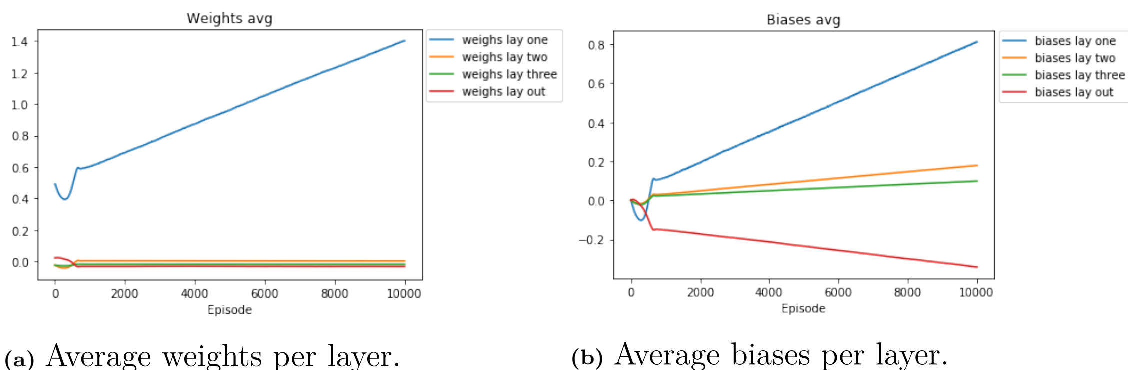


Figure 5.1: Average episode loss over all trained episodes for a DDQN with 2 hidden layers and 9 neurons in each hidden layer.

Figure 5.2a and 5.2b visualizes the average of the weights and biases in each layer over all training episodes. In Figure 5.2a, one can see that all weights converged, except the ones in the input layer. Further, Figure 5.2b shows that no biases converged for this setup.



(a) Average weights per layer.

(b) Average biases per layer.

Figure 5.2: The average weights and biases per layer, over all training episodes for a DDQN with 2 hidden layers and 9 neurons in each hidden layer.

5.1.1.1 State Representation

The full state representation was compared with a reduced one, as explained in Section 4.2. The results for this comparison are presented in Table 5.2. Furthermore, the trials were performed using a network with 2 hidden layers and 9 neurons in each hidden layer, as stated in Section 5.1.1. Finally, the same hyperparameter setup as in the trials performed in 5.1.1 was used.

Table 5.2: This table presents the difference for a DDQN with 2 hidden layers, 9 neurons in each layer, using a reduced as well as a full state representation.

Difference	State Representation
14.60 %	full
8.42 %	reduced

From the results in Table 5.2, it can be observed that the reduced state representation provided the best model performance. Further, the run-time for the reduced representation was 3.5 hours compared to the run-time for the full representation, which was 3.9 hours.

5.1.2 Hyperparameters

The results of the tuning of hyperparameters are presented in this Section. The hyperparameter setup that was finally chosen to proceed with is the one that provided the lowest percentage difference as well as the one that enabled network parameter convergence.

Table 5.3 presents the results of using the network configuration obtained in 5.1.1 with varying values of τ.

Table 5.3: This table presents the difference and run-time for a DDQN with 2 hidden layers, 9 neurons in each layer and varying values of τ.

Difference	τ	Run-time (h)
9.10 %	0.01	3.5
8.42 %	0.05	3.5
8.37 %	0.1	4.2

The results in Table 5.3 indicates that the model performed approximately equally good for all values of τ. The run-time was further shorter for the two lower values of τ. For the following trials, $\tau = 0.05$ was chosen.

In Table 5.4, the results from using the same network as above with varying values of the learning rate α are presented.

Table 5.4: This table presents the difference and run-time for a DDQN with 2 hidden layers, 9 neurons in each layer and varying learning rate α.

Difference	α	Run-time (h)
8.20 %	$5 \cdot 10^{-5}$	4.1
8.42 %	$1 \cdot 10^{-4}$	3.5
16.95 %	$5 \cdot 10^{-3}$	3.9

It can be observed from the results in Table 5.4, that the percentage difference was approximately the same for $\alpha = 5 \cdot 10^{-5}$ and $\alpha = 1 \cdot 10^{-4}$, however with a lower run-time for $\alpha = 1 \cdot 10^{-4}$. Further, with $\alpha = 5 \cdot 10^{-3}$ the performance was significantly worse. Therefore, $\alpha = 1 \cdot 10^{-4}$ was used in following trials.

The performances of using varying batch sizes with the so-far best network configuration are presented in Table 5.5.

Table 5.5: This table presents the difference and run-time for a DDQN with 2 hidden layers, 9 neurons in each layer and varying the batch size.

Difference	Batch Size	Run-time (h)
8.42 %	32	3.5
9.13 %	64	4.4
8.72 %	128	4.2

The percentage differences in Table 5.5 indicates that the network performed approximately equally well with all batch sizes. Further, the smallest batch size provided the lowest run-time. Therefore, the batch size 32 was chosen to conduct further trials with, since it would result in the most time-efficient code.

Table 5.6 presents the percentage differences when using the two weight initialization methods, He-initialization and Random Uniform initialization, for the weights

in the hidden layers.

Table 5.6: This table presents the difference for a DDQN with 2 hidden layers and 9 neurons in each layer, using the two weight initialization methods, He weight initialization and Random Uniform initialization.

Difference	Weight initialization
8.42 %	He-initialization
9.20 %	Random Uniform

From the results in Table 5.6, it can be observed that the model performance did not differ significantly between the two initialization methods. Further, the run-time was slightly (0.4 hours) shorter for the trial using the He initialization. The He-initialization method was chosen to perform the upcoming trials with.

In Table 5.7, the results of using the Huber loss function with different values of δ are presented. The network was trained for $E = 20000$ episodes and the performance was evaluated on $S = 10000$ simulations. This choice of increasing the number of episodes and simulations was based on results showing minimal to no difference in performance between the two loss functions when the default $E = 10000$ and $S = 5000$ was used.

Table 5.7: This table presents the difference for a DDQN with 2 hidden layers and 9 neurons in each layer using the Huber loss function with varying values of δ.

Difference	δ
9.68 %	1
11.44 %	10
13.61 %	50

It can be noticed from the results in Table 5.7 that the percentage difference increased with increasing δ. All differences in the table are higher than when the MSE loss function was used, which resulted in a difference of 8.42 % (Table 5.1). Further, the run-time per episode for the trials in Table 5.7 was approximately the same as when the MSE loss function was used. For this setup of episodes and simulations the run-time was thereby approximately 7 hours. Therefor, the MSE loss function was chosen to proceed with in remaining trials.

The results presented in this Section conclude the best network configuration for the DDQN to include 2 hidden layers and 9 neurons in each hidden layer. Further, $\tau = 0.05$, $\alpha = 10^{-4}$, a batch size of 32 as well as He weight initialization and the MSE loss function, corresponded to the configuration providing the best performance.

5.2 Double Deep Recurrent Q-Networks

In this Section, the results concerning the network architecture using a Double-DRQN are presented. The hyperparameters α and τ, the choice of loss function and the initialization of weights and biases were based on the hyperparameter search from Section 5.1.2. The same setup was therefor used. In all trials where nothing else is stated, the ACO algorithm were used as rollout function throughout the entire training. Furthermore, the network was trained for $E = 30000$ episodes, the performance was evaluated on $S = 5000$ simulations, and the batch size was set to 4.

5.2.1 Network Architecture

Different network architectures were implemented and analysed to find the best setup. In Table 5.8, one can see the results of using a Double-DRQN with 1 LSTM layer and different combinations of units, hidden layers, and neurons in the hidden layers.

Table 5.8: This table presents the difference for a Double-DRQN configuration with 1 LSTM layer and varied number of units, hidden layers and neurons in the hidden layers.

Difference	Nr units	Nr hidden layers	Nr neurons
16.57 %	5	2	9
9.60 %	10	2	9
16.79 %	15	2	9
13.56 %	5	2	32
13.72 %	10	2	32
13.75 %	15	2	32
13.22 %	5	2	64
8.22 %	10	2	64
13.48 %	15	2	64
10.13 %	10	0	0
14.01 %	10	3	64
16.98 %	10	1	64

It can be observed from Table 5.8 that the lowest percentage difference was obtained with the following configurations: a network with 10 units in the LSTM layer and 2 hidden layers with 9 neurons each, a network with 10 units in the LSTM layer and 2 hidden layers with 64 neurons each as well as a network with 10 units in the LSTM layer and no hidden layers.

In Table 5.9, the results of using a Double-DRQN with 2 LSTM layers, 10 units in each layer, different numbers of hidden layers and neurons in each hidden layer are

presented. The number of hidden layers and neurons in each hidden layer that were chosen for the trials were based on the results giving the best performance in Table 5.8, stated in the above paragraph.

Table 5.9: This table presents the difference for a Double-DRQN configuration with 2 LSTM layers, 10 units in each layer and different number of hidden layers as well as neurons in each hidden layer.

Difference	Nr units	Nr hidden layers	Nr neurons
13.83 %	10	2	64
14.62 %	10	2	9
13.96 %	10	0	0

The results in Table 5.9 indicates that the performance did not improve by adding a second LSTM layer. From Table 5.8 and 5.9, the best network configuration was chosen to be a network with 1 LSTM layer with 10 units, 2 hidden layers and 64 neurons in each hidden layer. This configuration was therefore the one chosen to use in following trials. The run-time for this final configuration was approximately 2.3 hours.

In Figure 5.3 the loss over all training episodes is visualized. From the figure, one can see that the loss decreased and reached an approximate convergence. Further, the fluctuations decreased at a lower value of the loss.

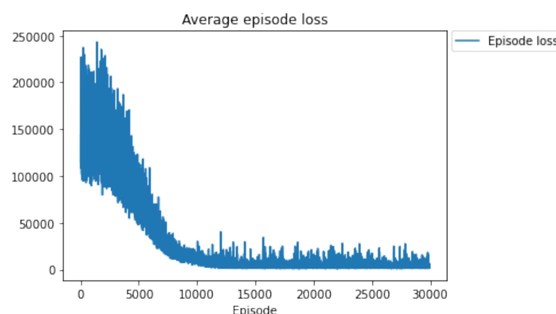
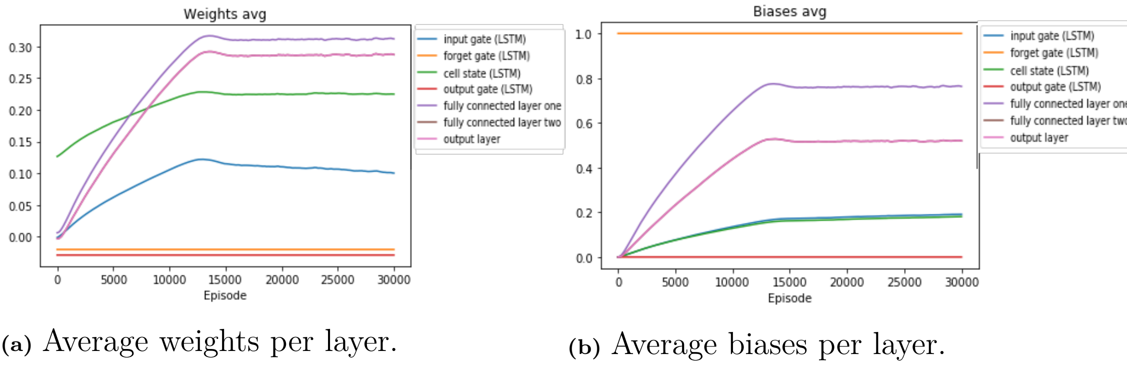


Figure 5.3: Average episode loss over all trained episodes for a Double-DRQN with 1 LSTM layer, 10 units, 2 hidden layers and 64 neurons in each hidden layer.

In Figure 5.4a and 5.4b, one can see the average of the weights and biases over each layer in all training episodes. From the figure, one can see that all network parameters converged, expect the ones in the forget and output gate of the LSTM layer, which were not updated at all.



(a) Average weights per layer.

(b) Average biases per layer.

Figure 5.4: The average weights and biases per layer, over all training episodes for a Double-DRQN with 1 LSTM layer, 10 units, 2 hidden layers and 64 neurons in each hidden layer.

In Table 5.10, the performances obtained when training the best network configuration with different dropout rates applied to the LSTM layer, are presented. The result of using no dropout, i.e. dropout rate 0, is included for comparative reasons.

Table 5.10: This table presents the difference for a Double-DRQN configuration with 1 LSTM layer and 10 units, 2 hidden layers with 64 neurons each and different dropout rates applied to the LSTM layer.

Difference	Dropout rate
8.22 %	0
8.45 %	0.1
8.02 %	0.2
8.35 %	0.5

It can be observed from Table 5.10 that the percentage difference was approximately the same for all dropout rates. The run-time did not differ significantly for any of the dropout rates either. Therefore, no dropout was applied in the following trials.

In Table 5.11, the performances for a configuration with 1 LSTM layer and 10 units, 2 hidden layers and 64 neurons using different batch sizes are presented.

Table 5.11: This table presents the difference and run-time for a Double-DRQN configuration with 1 LSTM layer and 10 units, 2 hidden layers with 64 neurons each and different batch sizes.

Difference	Batch Size	Run-time (h)
13.61 %	2	2.5
8.22 %	4	2.3
13.33 %	8	2.6

From the results in Table 5.11, it can be observed that the best performance as well as the shortest run-time was obtained when the batch size was equal to 4, meaning

that the network was fed with data containing 4 complete routes. A batch size of 4 was therefore chosen to use when performing the following trials.

Based on the results in this Section, the best configuration for the Double-DRQN includes 1 LSTM layer with 10 units and 2 hidden layers with 64 neurons each. Further, the best performance was obtained without any dropout applied to the LSTM layer as well as with a batch size equal to 4.

5.3 Case Study

In this Section, the results concerning the conducted case study are presented. The best found configurations of the DDQN (Section 5.1) respectively the Double-DRQN (Section 5.2) were used.

5.3.1 Double Deep Q-Network

In Table 5.12, the results from using case 1 and 15 with 10 customers as well as case 61 and 75 with 20 customers from the input data are presented. The best found DDQN configuration from Section 5.1 was used as well as the ACO algorithm as the action selection method during training. The network was trained for $E = 10000$ episodes and the performance was evaluated on $S = 5000$ simulations.

Table 5.12: This table presents the resulting difference when using case number 1 and 15 with 10 customers as well as case 61 and 75 with 20 customers from the input data. The best found DDQN configuration was used.

Case 1	Case 15	Case 61	Case 75
8.42 %	21.60 %	38.84 %	26.10 %

The results in Table 5.12 indicates that that the performance varied between the different cases. Furthermore, the best performance was obtained for case 1 with 10 customers and the worst for case 61 with 20 customers. The run-time for these are presented in Table 5.16.

5.3.2 Double Deep Recurrent Q-Network

The results from using case 1 and 15 with 10 customers as well as cases 61 and 75 with 20 customers from the input data are presented in Table 5.13. The best found Double-DRQN configuration from Section 5.2 was used, with the exception that the number of units in the LSTM layer was doubled for the 20 customer cases. The ACO algorithm was used as the action selection procedure during training, the network was trained for $E = 30000$ episodes and the performance was evaluated using $S = 5000$ simulations.

Table 5.13: This table presents the resulting percentage difference when using case number 1 and 15 with 10 customers as well as case 61 and 75 with 20 customers from the input data. The best Double-DRQN configuration was used.

Case 1	Case 15	Case 61	Case 75
8.22 %	21.89 %	40.11 %	30.11 %

The results in Table 5.13 indicates that the percentage difference varied between the different cases. Case 1 with 10 customers obtained the best performance, while case 61 with 20 customers obtained the worst performance. The run-time for these cases are presented in Table 5.16.

5.4 The Rollout Functions

In this Section, the effects of using solely the ACO algorithm, the ϵ-greedy algorithm and a combination of the two methods as action selection procedures are evaluated, using both the DDQN and Double-DRQN. Further, the two rollout functions, i.e. the ACO algorithm and the Tabu search are compared using the two different networks as well as different cases from the input data.

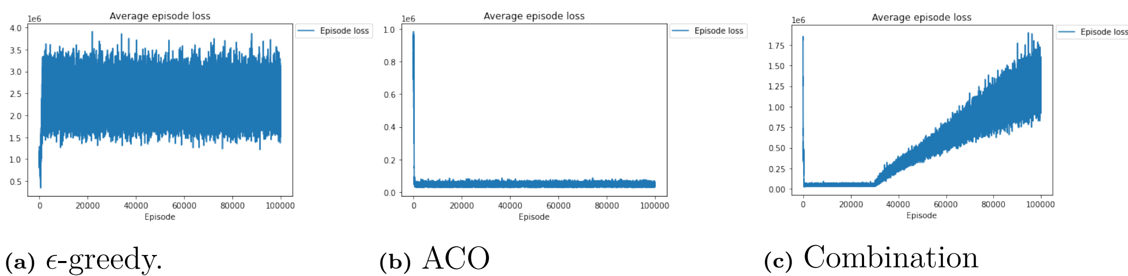
5.4.1 Double Deep Q-Network

The results in terms of performance from training using solely the ϵ-greedy algorithm, the ACO algorithm and a combination of them with the found best configuration are presented in Table 5.14. The combination was configured by using the ACO algorithm for the first 1/3 of the training episodes and then using solely the ϵ-greedy algorithm for the rest. For these setups, the network was trained for $E = 100000$ episodes and the performance was evaluated on $S = 15000$ simulations.

Table 5.14: This table presents the resulting difference when solely using the ϵ-greedy algorithm, when solely using the ACO algorithm and when using a combination of these two, using the best DDQN configuration.

	ϵ-greedy	ACO	Combination
Difference	16.83 %	7.98 %	15.23 %
Run-time (h)	44.6	36.7	41.2

From the results in Table 5.14, it can be observed that the best performance was obtained using solely the ACO algorithm. Furthermore, the performance was approximately equally bad when using solely the ϵ-greedy algorithm as well as when using a combination of the two methods.



(a) ϵ-greedy. (b) ACO (c) Combination

Figure 5.5: The average episode loss over all trained episodes for when solely using the ϵ-greedy algorithm, solely using the ACO algorithm and combining the two algorithms during training.

The results in Figure 5.5a, 5.5b and 5.5c indicates that the loss behaved significantly differently between the three trials. When solely the ϵ-greedy algorithm was used (Figure 5.5a), there was a high variance in the loss during the entire training and the loss did not decrease or converge. In comparison, when solely the ACO algorithm was used (Figure 5.5b), the loss decreased quickly early in the training followed by convergence and the variance was low throughout the training. For the final trial, when a combination of the two methods was used (Figure 5.5c), the loss decreased initially and converged before it increased again. It can further be seen that when the loss started to increase, the variance also increased.

Concerning the convergence of the network parameters, none of the biases as well as the weights of the input layer trained using solely the ϵ-greedy algorithm, converged during training. The parameters of the network trained using solely the ACO algorithm behaved similarly to the parameters in Figure 5.2a and 5.2b, i.e. all weights converged except the weights in the input layer converged. Further, no biases converged. Furthermore, when the combination of the two methods was used, the parameters behaved initially similarly as in Figure 5.2a and 5.2b. However, after the change in action selection method, all parameters with a non-converging behaviour changed more drastically than before.

5.4.2 Double Deep Recurrent Q-Network

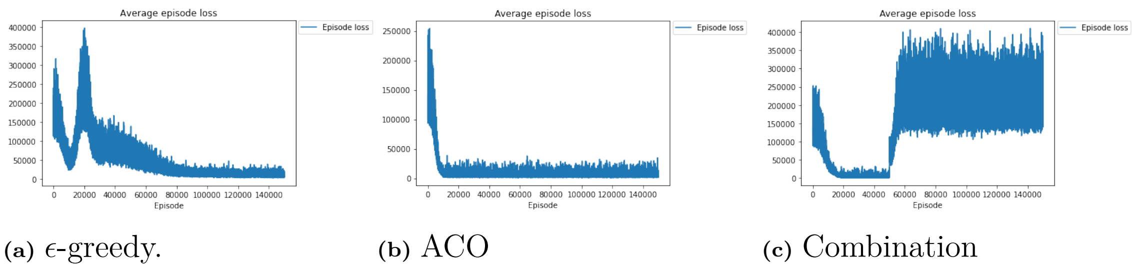
The results obtained from performing the same action selection procedures as in the previous Section (Section 5.4.1), using the best Double-DRQN configuration from Section 5.2, are presented in Table 5.15. The network was trained for $E = 150000$ episodes and the performance was evaluated using $S = 15000$ simulations.

Table 5.15: This table presents the resulting difference and run-time when solely using the ϵ-greedy algorithm, when solely using the ACO algorithm and when using a combination of these two, using the best Double-DRQN configuration.

	ϵ-greedy	ACO	Combination
Difference	13.61 %	13.41 %	17.60 %
Run-time (h)	18.7	11.5	18.1

The results in Table 5.15 indicates that the network performed equally well when it was trained solely using the ϵ-greedy algorithm as when it was trained solely using the ACO-algorithm. However, when a combination of the two algorithms was used, the performance was worsened. The run-time did also differ between the three different trials, with a significantly shorter run-time of 11.5 hours for the trial using only the ACO algorithm, compared to over 18 hours for the other two cases.

In Figure 5.6a, 5.6b and 5.6c one can see the average loss over all trained episodes, using the configuration described above.



(a) ϵ-greedy. **(b)** ACO **(c)** Combination

Figure 5.6: The average episode loss over all trained episodes for when solely using the ϵ-greedy algorithm during training, solely using the ACO algorithm and combining the two algorithms.

From Figure 5.6a, 5.6b and 5.6c it can be noticed that the behaviour of the loss varies over the training for the different configurations. It can be observed that the loss approximately converged (fluctuating behavior of the loss) for the trials when either ϵ-greedy or the ACO algorithm was used solely during training, i.e Figure 5.6a and 5.6b. For the trial when a combination of the two methods was used (Figure 5.6c), the loss did however not converge. In all cases, all network biases and weights converged.

5.4.3 Ant Colony Optimization vs Tabu Search

The results from comparing the two rollout functions, the ACO algorithm and the Tabu search, are presented in this Section. In Table 5.16, the percentage difference in total reward and run-time was used as a comparative measure for the investigation of the two network configurations when using different rollout functions.

Table 5.16: This table presents the resulting difference and run-time (h) for when either using the ACO algorithm or the Tabu search as a rollout function for two different networks. These comparisons are made on the best DDQN and Double-DRQN configurations. Case 1 and 15 with 10 customers as well as 61 and 75 with 20 customers from the input data was used.

Type of network	ACO	Tabu seach	Case Nr
DDQN	8.42 %	9.21 %	1
Run-time	3.5	3.5	
DDQN	21.60 %	28.81 %	15
Run-time	4.3	4.4	
DDQN	38.84 %	21.82 %	61
Run-time	8.9	16.0	
DDQN	26.10 %	55.67 %	75
Run-time	7.6	11.7	
Double-DRQN	8.22 %	14.57 %	1
Run-time	2.3	3.5	
Double-DRQN	21.87 %	25.50 %	15
Run-time	2.8	3.3	
Double-DRQN	40.11 %	21.84 %	61
Run-time	12.9	22.9	
Double-DRQN	30.11 %	51.375 %	75
Run-time	6.5	11.2	

From the results in Table 5.16, it can be observed that the performance for the 10 customer cases was slightly better when the ACO algorithm was used as rollout function compared to when the Tabu search was used. However, for the 20 customer cases, the ACO algorithm outperformed the Tabu search in case 75 but not in case 61. Further, the run-time was approximately the same when using the ACO algorithm and the Tabu search for the 10 customer cases. The run-time for the 20 customer cases, using the Tabu search rollout function, was significantly longer than for the ACO algorithm.

6
Discussion

This Chapter will discuss the results presented in Chapter 5, in combination with the problem formulation and implementations from Chapter 3 and 4. The discussion is also highly based on the theory from Chapter 2. In the following Sections, general challenges regarding the problem formulation and implementation will be discussed. Furthermore, discussions regarding the configuration of the DDQN as well as the Double-DRQN are covered. This is followed by investigations of the performance of the two networks using different cases in a case study. Finally, the results of using different action selection methods are compared and analysed in the rollout function Section.

6.1 Problem Formulation and Implementation

A challenge that has been recurrent during this thesis project is the fact that the problem was initially posed in the project report [2]. The problem, including factors such as how the environment is modeled and which parameters that were included in the state space, was constructed to be solved using a QL inspired reinforcement learning model with Q-tables. The MDP is for example formulated specifically for a Q-table approach by using data with overlapping information that is numerically close in a large state-action space. This is not optimal for neural networks since it makes it difficult for the network to generalize. Optimally, the MDP could be formulated in a way so that the environment contains a wider range of information, perhaps by using other or several parameters.

As mentioned in Section 2.4, when using a DQN instead of a Q-table, the state-action space of the problem is not restricted to be low-dimensional and discrete. However, if the problem is constructed specifically for a Q-table implementation, a DQN might not be a better choice. If the environment of the problem can be accordingly represented by a low-dimensional and discrete state-action space which can be mapped by a Q-table, a Q-table approach would probably perform better. This is due to that with enough training and exploration, each possible state-action pair will probably be revisited several times and the Q-value estimates (the Bellman function 2.27) in the Q-table will converge, as explained in Section 2.3. Using this Q-table, the optimal policy can be obtained. The strength of using a DQN, i.e to be able to generalize a large environment and select advantageous actions in unseen states, is not utilized in such situation and instead some of the flaws of using a neural network may be encountered. One such flaw is their tendency to converge to a

local optima, explained in Section 2.2.1.5, due to for example a complex non-convex solution space with several local optima. Another flaw is the large parameter space that is usually encountered in neural networks, requiring extensive hyperparameter tuning, which can be a time-consuming task.

In the project report [2], the continuous weight parameter and the set of visited customers were assumed to be unnecessary for the agent to learn what action to take in a certain situation. These state parameters were therefore not included in the state representation. Furthermore, the continuous battery parameter was discretisized. Without losing to much information of the environment, the dimension of the state-action space was thus reduced and discretisized, becoming well suited for a Q-table. With a more complex environment, consisting of more continuous and/or discrete state parameters, the environment would not be easily adjusted to suit a Q-table approach. Then the strength of using DQN would come to use, since it enables a more complex environment which can be used to capture the real-world complexity of the problem. This might have been the case if the problem was not constructed with a Q-table based solution method in mind.

Another upcoming challenge during this project is the fact that the code for this thesis is based on a Python translation of the Matlab code implemented in the project report [2]. Since the implementations in the Matlab code was based on over a year of work, it would take too much time to re-implement the entire code in a completely Python-optimal manner. Therefor, the existing Matlab code was translated into Python. A lot of time was then spent to optimize the translation, however the existing implementations restricted what could be done. For example, memory allocation is performed differently in Matlab compared to Python, which increased the run-time when implementing the code in Python. This resulted in a Python code that was much slower than the Matlab version even before it was extended with implementations of neural networks. Training and tuning the hyperparameters of the models has therefore been a time-consuming challenge.

In addition to the fact that the translation led to a potentially slower code than if it would have been re-implemented for Python specifically, the existing implementations did also later restrict how the extensions could be implemented. The extensions were not only supposed to integrate with the existing implementations, but also yield a better performance. If the code was implemented specifically for this problem setting, the use of neural networks as Q-value estimator and Python as a programming language could have been taken into consideration initially in all implementations. This could potentially have resulted in a reduced run-time and better performance.

As stated in Section 4.6, the rollout function was re-run for every evaluation in order to capture the stochastic nature of the process. This indicates that the percentage difference will differ for each identical run and therefor an average of several identical runs as a baseline might have resulted in a more reliable and generalized result. However, the results between identical runs differed at most 1-2 %, which

was evaluated to be reliable results at this stage due to the time limitations.

6.1.1 State Representation

As previously mentioned, when using a neural network as a Q-value function approximator, the dimension of the state-action space is no longer a restriction. The state- and action values can furthermore be continuous. The initial thought was therefore to include all state parameters of the problem in the state representation that is fed into the network. In order to see if this strategy was favorable or if including unnecessary state parameters worsens the performance of the model, another state representation was tested as well. In the later configuration, only the current vehicle position and weight were included in the state representation. Since the reward depends on the vehicle weight, as seen in equation 3.1, the vehicle weight is assumed to be of relevance for the agent when deciding on what action to take in a specific state. Further, the current vehicle position indicates where in the route the vehicle is, and can be assumed to be of relevance when choosing what action to take next. Q-values that corresponds to actions that have no current request or that have already been visited, are set to a value that the agent will not choose, as described in Section 4.4. Therefore, the sets of current requests and already visited customers are not included in the state representation that is fed into the network. Furthermore, as the battery capacity is not assumed to be a restriction for the vehicle in this problem, the battery level is considered irrelevant for the agent when deciding what action to take next.

From the results in Section 5.1.1.1 and table 5.2, it can be seen that the performance of the model worsens and the run-time increases when more state parameters are included in the representation. One reason for this result might be that an increase in the state parameters leads to an increase of the dimension of the state space. Furthermore, some of the state parameters are numerically similar, i.e close in the state space, and others have completely different numerical range. When dealing with a high dimensional space and numerically similar parameters as well as parameters covering entirely different ranges, an thorough exploitation and exploration strategy is essential. Such a strategy was aimed to be reached with the ϵ-greedy algorithm, however after analysing the result it seems that this algorithm might not be sufficient. Perhaps it can be of interest to develop an algorithm that randomly chooses between exploration and exploitation when taking actions. After each episode, feedback can be received to the agent on how effective the exploitation and exploration phase were, and the process of choosing between them is adjusted with some probability depending on the feedback.

As explained in Section 2.4.1.2, using correlated and numerical similar state parameters as input to the network can be a problem when using supervised learning methods, such as the DDQN. To solve this problem, one therefore uses an experience replay buffer which avoids correlations between sequenced samples. However, when dealing with a high-dimensional state space, such as the one including all of our state parameters, it is difficult to recognize similarities and patterns within the state pa-

rameters. This implies that the dynamics of the state parameter might be hard to capture in a high-dimensional state space by only sampling with an experience replay buffer. Further, using a high dimensional state space might also lead to overlapping information among the state parameters. Thereby, it is of interest to find the minimal state space representation that requires model information. Therefore, a reduced state representation was evaluated. The results from Section 5.1.1.1 implies that a reduced state representation in fact improves the network performance, which strengthens the arguments of the problem with a high dimensional state space.

However, even with a reduced state representation, there can still occur overlapping information between state parameters as well as dynamics within the state parameters that are difficult to capture. In that case, it might not be enough to use sampling with a vanilla experience replay buffer. To ensure that the dynamics of the model are captured and that the samples aren't correlated, a more complex experience replay buffer would be interesting to implement and explore.
From the results in Section 5.1.1.1, it is clear that the state representation has a great influence on the model performance. When evaluating combinations of different state representations, high correlations between these and the parameters in the reward function were observed. It would therefore be of interest to evaluate parameters such as speed, time, and distance in the state representation, since these are included in the energy coefficients explained in Section 4.1 and can therefore help the network see connections between states and rewards.

6.1.2 Non-linear Optimization Problems

As seen in Section 2.2.1.1, ReLU is an activation function used to achieve a non-linear transformation of the data. This is the activation function that is used in all the layers of our network, except the output layer. In order to solve the unstable gradient problem explained in Section 2.2.1.1, as well as allow the model to create complex mappings, one needs to use such a non-linear activation function with DNN. Since ReLU applies a non-linear transformation to the input data, i.e. a batch of states, the problem of optimizing the network to map the state space becomes non-linear. This problem should however not be confused with the problem of minimizing the expected energy consumption, as in equation 3.2, which is linear.

Non-linear optimization problems are very complex due to their sensitivity in the choice of initial conditions, i.e. weights and biases. If we start our ADAM optimization algorithm with ill-posed initial conditions it is easy to end up in a local optima. It is also impossible to know what kind of initial conditions that will guarantee a path to the global optima, especially when dealing with such a large parameter space as in our case. When taking larger steps in the local optima, by increasing the learning rate α, it is however possible to get out of that sub-optimal solution and perhaps find an optimal one. The trade-off is though when the step size is chosen as too large, which might lead in us overshooting the minima and thereby missing the optimal solution. To be able to solve such a nonlinear optimization problem, time

is of essence since it requires some extensive analysis regarding how to choose the initial conditions when dealing with a large parameter space.

6.2 Double Deep Q-Neural Network

A difficulty when tuning the DDQN is that the parameter space is large and requires fine-tuning for each individual parameter. When tuning one parameter it might also require tuning of another, which has been a nested and complex task. If more time was available, it would be of interest to examine combinations of the network architecture more, as well as the learning rate, the loss function, the regulation parameter τ and the initialization of the weights and biases.

6.2.1 Network Architecture

For the network architecture, a decision had to be made concerning the number of hidden layers as well as how many neurons that should be in each of these layers, as mentioned in Section 2.2. Usually, two or fewer layers are sufficient when dealing with simpler, not linearly separably data sets. Adding layers will enable the network to learn more complex relationships, however it would also increase the complexity of the model, i.e. lead to increased processing time and overfitting [74]. When choosing the number of hidden neurons, there are some guidelines to be used. One general rule of thumb is to set the number of hidden neurons to be between the size of the input layer and the output layer [75]. When using too many neurons in the hidden layers the computation time increased and the results indicated that the network was overfitting. However, choosing too few neurons in the hidden layers presented tendencies of underfitting from the network. Therefor, these results from trial and error were used in combination with theoretical guidelines, to find the best configuration for our problem formulation.

Different configurations of hidden layers and nodes were thereby tested, where the best configuration is presented in Section 5.1.1. This result indicates that some trial and error was required and that the choices of the number of hidden layers and neurons were highly dependent on the problem and use case. As seen in the result from table 5.1, the model performed best using 9 neurons in each layer with no big difference in performance when using 1, 2 or 3 hidden layers. As indicated previously, the best network configuration was only tuned based on 1 test case with 10 customers. Therefore, 2 hidden layers were chosen to proceed with, since 1 hidden layer might not enable the network to successfully learn all complex relationships and 3 hidden layers might over-complicate the model in other cases. The resulting loss for this model configuration can be seen in Figure 5.1, where a clear non-fluctuating decrease followed by convergence of the loss function is seen. This indicates that the training is converging, implying that the network is learning.

Lastly, when tuning the network configurations, the most important takeaway is to write the code as efficient as possible, in order to have the time to tune the network

with some guidelines as well as through trial and error.

6.2.2 Tuning of the Network

The regulation parameter τ was tuned as well, which is presented in table 5.3. From the table it is clear that the model performance is not sensitive to the value of τ. Therefore, the midmost value $\tau = 0.05$ was chosen to conduct the following trials with. This choice was based on the arguments in Section 2.4.1.1, implying that a too large value can destabilize the learning by updating the target network parameters too much in each iteration. The highest value did also result in the longest run-time. Furthermore, a too low value can slow down the learning and prevent the algorithm from converging to an optima.

Different initial learning rates, α were also explored to some extent in Section 5.1.2. This was done by exploring different initial α and adapting individual learning rates for each parameter in the network through the ADAM optimization algorithm, explained in Section 2.2.1.5. However, the network showed to be very sensitive to the choice of α and seemed to perform best when $\alpha = 5 \cdot 10^{-5}$ and $\alpha = 1 \cdot 10^{-4}$, presented in table 5.4. These learning rates are however seemingly small, which is intended to enable the model to learn a global optimal set of weights by avoiding to overshoot the minima. However, if the learning rate is too small for the specific task, the training might be too slow leading to the α never converging due to getting stuck in a sub-optimal solution. Therefor, the slightly higher value, $\alpha = 1 \cdot 10^{-4}$ was chosen for further trials. This learning rate did also result in the shortest run-time. Configuring the learning rate was a challenging and time-consuming task due to its large impact on the result and would be interesting to evaluate further, if more time was available.

The batch size is iterated in Section 5.1.2 since it impacts how quickly the model learns and how stable the learning is, which is presented in table 5.5. With a larger batch size, the loss function will be less fluctuating through updates, due to the estimates being based on a large amount of data. However, this leads to slower learning, requiring a lot of memory for processing. On the other hand, using a too small batch size might lead to the gradients being based on noisier estimates of biased data, since the training samples are randomly drawn from the dataset. A smaller batch size will therefore lead to a noisier and more fluctuating learning, which either can help the learning escape from a local optima or can extend the time for model convergence by overshooting global optima. In the Table 5.5, it is however clear that the model is not that affected by the choice of batch size. This might be a result of using experience replay, explained in Section 2.4.1.2. Since each experience is stored in a replay buffer and then sampled randomly from the buffer, each training sample becomes a random sample. Due to this randomness, there is a high probability that the samples used for training are representative of the problem, leading to parameter updates in the right direction. Due to the low impact of the batch size in this case as well as since an increased batch size also results in increased processing time, the smallest batch size of 32 was chosen for upcoming trials.

The evaluated loss functions were the MSE loss and the Huber loss as seen in the results from Section 5.1.2. In order to evaluate the Huber loss, the hyperparameter δ was tuned, which is explained in Section 2.2.1.3. The results of using different values of δ is presented in Table 5.7 and shows a decrease in the model performance when increasing δ. As explained in Section 2.2.1.3, if $\delta \rightarrow 0$, the Huber loss is advancing towards the MSE loss. This indicates that in order to fit the model to the data, outliers as well as high errors in the prediction needs to be taken into consideration. From these evaluations it can be concluded that the network performs better when using the MSE loss. This conclusion might have been even more clear if the amount of episodes would have been increased further, which was not possible due to the long processing time. These arguments for choosing the MSE loss over the Huber loss are further based on the fact that the Huber loss is a mixture of a linear and quadratic loss, explained in Section 2.2.1.3. Switching between these losses can result in a non-smooth jump for the loss function, meaning that the gradient can suddenly jump and disappear as well. Such a behavior can lead to an unstable gradient problem, which in turn results in an unstable learning and inhibits network parameter convergence, as explained in Section 2.2.1.

As stated in Section 6.2.1, the result in Figure 5.1 indicated training convergence. However, Figure 5.2a and 5.2b, presenting the convergence of the network shows that even though almost all weights are converging, no biases are. This result was not expected due to the converging behavior of the loss in Figure 5.1. From Figure 5.2a and 5.2b, one can see that the parameters with the most dominating values are the ones with the most diverging behavior. This includes the weights in the input layer as well as all biases. The reason for this non-converging behaviour of solely the input weights might be due to its initial values. The weights in the input layer are initialized by the Xavier uniform initialization method, whose values depend on the number of neurons in the layer (equation 2.8). The fewer neurons, the higher values. The same holds for the He initialization, see Section 2.2.1.2. With only 2 neurons in the input layer, the values might be initialized too large. These parameters will thereby dominate the output values from each neuron in each layer, described in equation 2.5. Further, during the learning phase, the policy will be non-stationary due to the Q-values continuously being updated until the optimal policy is found. This non-stationary behaviour cannot be captured by converging parameters, which might be the reason for the parameters not converging resulting in a continuously growing value throughout training, as seen in Figure 5.2a and 5.2b.

As mentioned in the above paragraph, the weights in the input layer were initialized by the default initialization method, i.e. the Xavier initialization. This choice was based on some trial and error as well as on some research regarding previous implementations of the initialization methods in the input layers. Since the He initialization of the weights did not result in an improved performance and the Xavier initialization seemed to be the most commonly used method for the input layer, this choice seemed appropriate. Two different weight initialization methods were thereafter evaluated for the hidden layers, namely the Random Uniform and He weight initialization. The results are presented in Table 5.6. From the results it can be

noticed that the performance does not differ significantly between the two methods. However, the He initialization method has advantageous characteristics when using the activation function ReLU, as stated in Section 2.2.1.2 as well a slightly better performance. Further, the run-time was slightly shorter for the He initialization compared to the Random Uniform initialization method. Therefore, the He initialization was chosen to conduct future trials with. The biases were initialized as zeros in all trials, since it is the most commonly used method providing consistent results as stated in Section 2.2.1.2.

Furthermore, the results showed that when improving the combination as well as fine tuning the parameters, the prediction capabilities of the network improved. It is therefor probably possible to find the optimal configuration of these parameters, which would result in increased performance and efficient learning. This implies that there is potential to find a deep Q-neural network that outperforms the results from the Q-tables.

6.3 Double Deep Recurrent Q-Network

The idea of using a Double-DRQN was to see if it is possible to capture the long term dependencies that occurs in the routing problem by training on sequenced data. This was implemented by training on batches of complete routes instead of on independent actions from several different routes, in order to optimize the action selection, as described in Section 4.5.

As described in Section 2.2.2.1, LSTM layers are commonly used to handle problems concerning unstable gradients. This motivated the use of LSTM layers in the sense that it might increase the convergence possibilities for the network. Another reason to use the Double-DRQN with LSTM layers is due to the fact that training is performed less frequently than in the DDQN. In the DDQN, training occurs after each epoch, i.e after each action is taken. This training data is continuously saved in the experience replay buffer and sampled randomly in a batch for training. However, in the case of the Double-DRQN, training occurs only after each episode, i.e. after each complete route, containing at most 10 actions. This training data is saved in a buffer as well, where a batch of routes are sampled randomly for training. The training process in the DDQN will therefor be much slower and time-consuming task than in the Double-DRQN. By using LSTM layers, it was thereby possible to train the network for more episodes in less time than in the DDQN, enabling the network parameters to fully converge as well as to minimize the time inefficiency. As seen in the result from Section 5, one can see that the Double-DRQN with one LSTM layer performs equally good as the DDQN with more parameters converging as well as with 1.5 times faster run-time and 3 times more trained episodes.

6.3.1 Network Architecture

Regarding the network architecture, various combinations of LSTM layers and fully connected layers were evaluated as seen in Table 5.8. The architecture of a Double-DRQN is highly based on the problem formulation and use case, such as the DDQN in Section 6.2. In order to yield the best possible performance, some guidelines were used in combination with trial and error. For the LSTM layers, both the number of layers and the number of units in each layer has to be chosen. Depending on the complexity of the features to detect in the input sequences, it is usually enough to use one or two LSTM layers. The more layers, the more complex is the training since each layer increases the number of trainable parameters in the network. Since the number of units in LSTM reflects how much data is kept in memory at a time, it is often adjusted after the length of the input sequences.

In order to compare the impact of using 1 or 2 LSTM layers, the 3 best configurations when using 1 LSTM layer were implemented using 2 LSTM layers. The results in Table 5.8 and 5.9 shows that the best performance is obtained when only 1 LSTM layer is used. A reason for this might be that the features in the input sequence which should be detected by the LSTM layers are not that complex. Adding additional LSTM layers does therefore only make the model more complex to train and might lead to overfitting, which can be seen in the worsened performance when comparing Table 5.8 and 5.9. Regarding the number of units to have in the LSTM layer, it can be observed from the results in Section 5.2.1 that the best performance is obtained when 10 units are used. Since the results are obtained using case number 1 with 10 customers, the routes are at a maximum of length 10. This strengthens the argument in the paragraph above, that the optimal number of units depends on the length of the input sequences.

As explained in Section 6.2, adding fully connected layers will enable the network to learn more complex relationships, which is the case in the Double-DRQN as well. It is also necessary to have a fully connected layers as the output layer in order to get a single value output rather than a sequenced output. Therefor, fully connected layers were added after the last LSTM layer. This architecture also increases the complexity of the model in the same way as when adding more LSTM layers. One therefore often tries to keep the number of additional fully connected layers as few as possible. Regarding the choice of number of neurons in each hidden layer, the same assumptions as discussed in Section 6.2 was used. From the results in Section 5.8, it can be observed that the optimal performance is obtained using 2 hidden layers with 64 neurons in each layer. These results can be assumed reasonable as they are in line with the arguments in the paragraph above.

6.3.2 Tuning of the Network

The average episode loss over all trained episodes is presented in Figure 5.3, showing a decrease in the loss until training has reached approximate convergence. This indicates that the network is learning. However, one can also see fluctuations in the loss, which might be due to the fact that the targets are continuously updated

during training as described in Section 2.4.1.1. Another measure to indicate how well the network is learning is the convergence of the weights and biases, presented in Figure 5.4a and 5.4b. The result in these Figures indicates that the network is learning from training, since most of the network parameters are in fact converging. As explained in Section 6.2.2, the policy is non-stationary during training. In the DDQN case this non-stationary behavior could not be captured by converging parameters. However, in the Double-DRQN case, the LSTM layer enables the network to take long-term dependencies into account, which might be the reason for why the behavior of the policy can be captured by most of the network parameters. This in turn results in convergence of these parameters, seen in Figure 5.4a and 5.4b. The values of the parameters in the forget gate as well as in the output gate of the LSTM layer does however not seem to be updated. A potential reason for this might be the fact that the discount factor γ is set to 1 for this problem, which is explained in Section 4.4. The motivation for using this value of the discount factor was that the agent should choose its actions based on the cumulative future reward rather than the immediate reward. However, this implies that all rewards are equally important, i.e. nothing from the input sequence should be forgotten. Thereby, the initial parameter values will not be forgotten, meaning that the corresponding parameters will not be updated.

As mentioned in Section 2.2.2.1, LSTM tend to overfit the data due to its large amount of learnable parameters. This problem can be prevented using a dropout layer applied to the final LSTM layer. From the results in Table 5.10 it can however be observed that the performance is not improved by adding a dropout layer after the LSTM layer. This indicates that there are probably no overfitting caused by the LSTM layer, i.e. any occurring overfitting can be assumed to be caused by the fully connected layers.

From the results in Table 5.11, it can be noticed that the performance of the Double-DRQN is quite sensitive to the batch size. The best performance as well as the shortest run-time was obtained with a batch size of 4 sequences. The same arguments that were used for DDQN in Section 6.2 was used for choosing the batch size in the Double-DRQN. The results from Table 5.11 is inline with this argumentation, where the midmost batch size gives the best model performance. A reason for this observed sensibility to the batch size might be the fact that the Double-DRQN is trained on samples of entire routes instead of on samples of independent actions, as in the case of DDQN. If some of the routes in a sample are not representative of the problem, the network parameters might be updated in the wrong direction. With larger batches the updates will be less fluctuating since they are based on more data. However, if the data that the updates are based on is not representative of the problem, larger batches might lead to worsened performance as the algorithm easier converges to a local optima.

As discussed in previous Section 6.2, tuning of the network hyperparameters and architecture was a time consuming challenge in this project. Because of the time constraints, the effect of using different loss functions, different values of the regulari-

sation parameter τ, different learning rates α and different initialization methods for weights and biases was not evaluated as thoroughly for the Double-DRQN. Instead, the parameter setup that resulted in the best performance of the DDQN was used, which is explained in Section 6.2.2. For example, the MSE loss was chosen as a loss function over the Huber loss, based on the performance of the DDQN, explained in Section 6.2. The argument for this choice was partly due to the hypothesis of the Huber loss tendencies of enabling unstable gradients. However, in the case of the Double-DRQN, 1 LSTM layer was included in the network configuration, which takes care of the unstable gradient problem. Thereby, the Huber loss might have been the better choice for the Double-DRQN. Choosing this loss function would however require some extensive tuning of hyperparameter δ, which was too time consuming to investigate further. With more time, it would therefor have been of interest to explore if there exist a better network configuration and hyperparameter setup specifically for the Double-DRQN.

6.4 Case Study

When comparing the results from the case study in Section 5.3.1 from Table 5.12 and in Section 5.3.2 from Table 5.13 with the results obtained in the project report [2], it can be observed that the Double-DRQN as well as the DDQN performs worse than the QL inspired method in all cases. There are several possible reasons for this overall bad performance. For example, the challenges regarding the problem being initially formulated to be solved using a Q-table based approach as well as finding a suitable state representation, which was discussed in Section 6.1 and 6.1.1. It can also be concluded that the results vary depending on the type of case, which was expected and is the case in the project report as well. However, the problem might change to much in the case of 20 customers. The best configurations of the networks were chosen based on one case, due to the time constrains explained in Section 4. Therefor, the network might not generalize well when training on different cases, especially in the case of 20 customers. It would thereby be of interest to tune the networks based on several cases, thereby finding a configuration that might generalize better to unseen cases as well as to a slightly changed problem, when including more customers.

6.5 The Rollout Functions

The rollout functions are mainly used in order to guide the agent in choosing favourable actions given current states. This process of finding optimal solutions, as described in Section 2.1.3.1 and 2.1.3.2, proceeds throughout the learning. In each epoch, this therefore results in less exploration tendency for the agent, i.e the agent uses available information rather than randomness to choose an action. This procedure arises from not using the ϵ-greedy algorithm, described in Section 2.3. Since we are dealing with a high dimensional space and numerically similar parameters as

well as parameters covering entirely other ranges, some exploitation and exploration might be necessary to predict favourable actions in an unvisited state as discussed in Section 6.1. If the agent only learns from actions in a close neighborhood in the state space, the unexplored state space will be large. Therefor, the agent will have limited knowledge of a more general action selection, concerning a larger state space. Furthermore, by only using the rollout functions, the predictions regarding dynamic customer request is not taken into consideration. This leads to only using an expected energy when planning the route and therefor basing the prediction of the routes on static customer requests.

In the project report [2], the reinforcement learning method outperforms the rollout function, which can be concluded by analyzing the values of the difference. However, in that case, the prediction is based on both dynamic customer requests as well as battery constraints. This requires prediction based on samples from a probability distribution function to account for unknown dynamic customer requests and to dynamically plan charging of the battery. This is done by the reinforcement learning method in contrast to the rollout function. In this thesis, the rollout functions either outperforms or performs equally well as the deep Q- learning method. One reason for this might be that the battery constraints are not taken into account, which requires less dynamic prediction abilities.

6.5.1 Double Deep Q-Network

From the results in Table 5.14, it can be observed that the performance when using the ACO algorithm slightly increased compared to when training on fewer episodes, presented in Table 5.1. It can also be noticed that using this algorithm resulted in the lowest run-time. When analysing the behaviour of the loss in Figure 5.5b, one can see that the loss decreased quickly and thereafter converged. However, as stated in Section 5.4.1, no biases converged and all weights except the one in the input layer converged. This indicates that the non-stationary behaviour of the policy cannot be captured by converging parameters, discussed in Section 6.2.2, even with an increased amount of episodes. However, with enough training an optimal and stationary policy would perhaps be reached, which would eventually enable convergence of the network parameters.

When only using the ϵ-greedy algorithm, i.e not using the rollout functions at all, more randomness is introduced in the action selection processes. From the results in Table 5.14, one can see that not using the rollout function at all results in the highest difference, i.e the worst performance of the network. In Figure 5.5a, it can be seen that the loss clearly does not converge and has a wildly fluctuating behavior. Further, almost none of the network parameters converged. This might be due to the agent exploring the environment to a larger extent, which makes it more difficult and time-consuming to find a good policy. In order for the training, parameters and loss to be able to converge, it is probable that the amount of episodes needs to be increased even more. As seen in Table 5.14, the run-time for solely using

the ϵ-greedy algorithm was 44.6 hours, which is approximately 8 hours longer than when solely using the ACO algorithm. With the setup used in this thesis, there was therefor not enough time to investigate the hypothesis of further increasing the episodes. However, if one would succeed to train for a larger amount of episodes without suffering from the time inefficiency, the ϵ-greedy algorithm might perform equally well or perhaps even outperform the rollout function.

To get some guidance regarding favourable actions by the rollout function as well as explore the environment to a greater extent, a combination of these methods was investigated, explained in Section 4.4. When a combination is applied, the rollout function is used for the first 1/3 episodes and the ϵ-greedy algorithm for the last episodes. In Figure 5.5c one can see where the guidance of the rollout function cease, due to an increase in the loss. The fluctuating behaviour of the loss function in Figure 5.5c could potentially be explained by drastic changes in the variance of the loss function. These changes might be due to the switch from one action generating method to another during training. Since the network is fed with independent actions from several different routes generated by both the ACO algorithm as well as the ϵ-greedy algorithm, the variance in the loss might fluctuate to a large extent. Further, since the target network is updated slowly and is initially trained on samples solely from the ACO algorithm, the loss will increase when the network starts to train on samples from the ϵ-greedy algorithm. Therefor, this might be a potential reason for the fluctuating behavior and high variance in the loss, seen in Figure 5.5c.

6.5.2 Double Deep Recurrent Q-Network

When increasing the amount of episodes and simulations, the Double-DRQN seemed to perform worse when solely using the ACO algorithm. This can be seen in the difference in the average total reward, presented in Table 5.15. With 30000 episodes and 5000 simulations, the difference was 8.22% (Table 5.8) in comparison to when using 150000 episodes and 15000 simulations, resulting in a difference of 13.41% (Table 5.15). However, when solely using the ϵ-greedy algorithm, it is clear from Figure 5.6a that the loss decreases and converges when using 150000 episodes in comparison to the loss at 30000 episodes. This implies that the ϵ-greedy algorithm requires more episodes for the loss to be able to converge.

A reason for the worsened performance in the case of using solely the ACO algorithm might be that the algorithm does not explore the state space in a large extent. The amount of variating data or states that are generated, might thus be limited. The network would therefor be fed with quite similar data during the entire training, and the larger amount of episodes, the more similar states would be generated. This could potentially be compared to training a network on a dataset not representing the problem accordingly, leading to the network not generalizing well to new data. Such training tends to cause overfitting. To train the Double-DRQN for too many episodes solely using the ACO algorithm might thus lead to overfitting of the network, which could explain the worsened performance that occurred when increasing

the number of training episodes. This analysis is however difficult to validate due to the lack of validation data when dealing with a regression task.

For the case when solely using the ϵ-greedy algorithm in the Double-DRQN, the model performance slightly improves with increasing episodes, as seen in Figure 5.6a. This result was expected, since it increases the opportunities for the network to both explore and exploit the environment to a larger extent. As seen in Figure 5.6b, the loss converges after approximately 10000 episodes when solely using the ACO algorithm while it takes approximately 80000 for the loss to converge in the case of ϵ-greedy as in Figure 5.6a. However, the ϵ-greedy algorithm now performs equally well as the ACO algorithm overall, which was not the case with fewer episodes. This result confirms the hypothesis that the ϵ-greedy algorithm can perform equally well or perhaps better than the rollout function when increasing the training episodes. When using the Double-DRQN, this task became less time consuming, which thereby enables the improvement of the ϵ-greedy algorithm. With more time, it would therefor be of interest to tune the network further, using solely the ϵ-greedy algorithm.

The combination of the ϵ-greedy algorithm and ACO algorithm resulted in the highest difference, i.e. the worst performance of the Double-DRQN (Table 5.15). The behaviour of the loss function clearly shows an explosion as well as a fluctuating behavior when the change from the ACO algorithm to the ϵ-greedy algorithm occurs after 50000 episodes, as seen in Figure 5.6c. Following the arguments in Section 6.5.1, the loss might have decreased and converged if the amount of episodes was increased even further. However, the idea of the combination was to provide the network with "good" training samples based on the reward from the ACO algorithm, thereby improving its overall performance, which was not the case. The same argument as in Section 6.5.1 can be applied here, i.e. the fluctuating behaviour of the loss function in Figure 5.6c could potentially be explained by drastic changes in the variance of the loss function. However, in this case for the Double-DRQN, the loss has an even more fluctuating behavior than in the DDQN, as seen in Figure 5.5c. A reason for this can be that the variations in the variance is higher in batches of routes than in independent actions from several different routes. Further, when using a combination of the ACO algorithm and ϵ-greedy algorithm, the variance in the batches (i.e. the sequences of routes) might increase when switching between the two action generating methods. Since we use a experience replay buffer, the network will be fed with a mixture of routes generated by the two methods after the switch. Such a setup might thereby result in the fluctuating behavior and high variance of the loss function, seen in Figure 5.6c.

6.5.3 Hyperparameter Search

It is clear from Section 2.1.3.1 that the ACO-algorithm includes a set of tuneable parameters that should be optimized for each individual problem to yield best possible performance. As further explained in Section 2.1.3.1, it is suggested to use equal number of ants as nodes in the route to construct. There are at maximum

10 nodes in the route for the 10 customer case, so the first assumption might be to use 10 ants. However, as the ACO algorithm only optimizes a route based on known customers and since not all requests are known at forehand, the number of current nodes in a route will not be 10 in the beginning of an episode. Further, the algorithm optimizes the route from the current vehicle position including currently known customer requests. This means that even though more customer requests will be known after some epochs, the length of the route to optimize will still not be 10, since the current vehicle position has progressed. Since the number of ants and iterations highly impact the run-time of the algorithm [76], these were set as low as possible to yield good performance without suffering from a high run-time. Therefore, 5 ants were used instead of 10 and the number of iterations were set to 10, in the 10 customer cases. This argument was used in the 20 customer cases as well, setting the number of ants to 10. However, the number of iterations was not adjusted in the 20 customer cases. The choice of iterations in the 10 customer cases was mainly based on trial and error. The rest of the parameters, i.e α, β, ρ and the initialisation of τ were set to the recommended values presented in Section 2.1.3.1.

6.5.4 Ant Colony Optimization vs Tabu Search

As seen in Table 4.1, the ACO algorithm performs better than Tabu search for all cases with 10 customers for both the DDQN and the Double-DRQN. It can also be concluded that both Tabu search and the ACO algorithm performs individually best in case 1 with 10 customers, for both networks.

Further, the ACO algorithm performs worse than the Tabu search for the cases with 20 customers (Table 4.1) for both networks, which might be due to the tuning of hyperparameters solely being performed on case 1 with 10 customers, as explained in Section 6.5.3. The hypothesis of using the ACO algorithm over the Tabu search was based on that the Tabu search might be a too simple rollout function, which could provide a worse performance on a more complex problem. Therefor, we expected the ACO algorithm to perform slightly better or equally well as the Tabu search in the less complex case with 10 customers, and to outperform the Tabu search in the more complex case with 20 customers. This was based on that in comparison to the Tabu search (which iteratively searches for an improved solution), the ACO algorithm can rapidly find good solutions thanks to the pheromone feed-back mechanism. In theory, this implies that in each iteration, the ACO algorithm should generate the best possible routes, while the Tabu search iteratively tries to improve the current one. Therefor, the ACO algorithm was expected to yield both a shorter run-time as well as a better overall performance compared to the Tabu search.

In the tuned case with 10 customers, this theory seems to be correct in most cases, which can be observed from Table 5.16. However, we did not expect the Tabu search to outperform the ACO algorithm in the more complex case with 20 customers, since it requires optimizing a longer route. As stated previously in this Section, the reason for this result might be simply due to the tuning not being adequate for the case of 20

customers. Further, when analysing the run-time for the different rollout functions and cases in Table 5.16, one can see a significant increase in run-time for the 20 customer cases when using the Tabu search. As stated in Section 4.3, the number of ants was modified for the 20 customer cases, however not the number of iterations. This might be a reason for the poor result of the ACO algorithm in these cases, as well as the low run-time. By not increasing the number of iterations in the more complex case, exploration in the solution space might be insufficient. Thereby, it would of interest to increase the number of iterations in order to enable the ants to find an optimal solution and see if the ACO algorithm can in fact outperform the Tabu search in the more complex cases.

7

Conclusion

Even though the resulting model did not perform better or even as good as the Q-tables in previous work, the project is still considered a success due to its findings. The discussion in chapter 6 is used as a baseline here to draw conclusions regarding the results from Chapter 5.

7.1 Research Questions

In order to conclude the results and discussions made in Chapter 5 and 6, the research questions that we aim to answer when conducting this thesis as well as some concluding remarks are used.

7.1.1 What is the effect of implementing a Deep Q-Learning model compared to the existing Reinforcement Learning model?

Comparing the results in Section 5 with the results obtained in the project report [2], it can be concluded that the existing reinforcement learning model performs better than the different DQL-models for all cases and configurations. Using the neural network instead of the existing reinforcement learning model also resulted in a significantly increased run-time.

One reason for the overall bad performance using DQL might be the problem being formulated to be solved using a Q-table based approach as well as being specifically adapted for a Matlab based solution. Another reason for the poor performance might also be due to the time consuming task of tuning the hyperparameters and configurations of the networks. Since using the neural networks as function approximators resulted in a remarkably higher run-time, several limitations regarding the configuration and tuning of the networks were applied.

Furthermore, it was possible to successfully implement a DQL-model, which in fact provided a function approximator that was able to learn more complex environments. These environments may contain continuous and infinite state-action spaces, which the existing reinforcement learning model was not able to capture. This open up possibilities within the VRP, since it is a problem with growing complexity.

7.1.2 How does the choice of Deep Q-Neural Network impact the Deep Q-Learning model?

Comparing the results in Section 5.1 and 5.2, one can conclude that the best configurations of the two different networks performed approximately equally well in terms of the percentage difference (equation 4.1). However, the per episode run-time was significantly lower for the Double-DRQN compared to the DDQN.

It can further be concluded that the network parameters of the Double-DRQN converged in a higher extent then the ones in the DDQN. From the discussion in Section 6.2.2, the reason for this might be due to difficulties in capturing the non-stationary policy with the network parameters of the DDQN. However, using a Double-DRQN enabled parameter convergence, due to the time efficiency as well as the ability of taking long-term dependencies into account, thereby managing to capture the non-stationary policy. Further, when comparing the loss function for the different trials of the Double-DRQN to the ones for the DDQN, there is a big difference in the fluctuating behavior of the function. This is due to the different sampling methods in the two networks, where the sampling in the Double-DRQN case introduces higher variations in the variance during training.

The hyperparameter setup, the loss function as well as the initialization of the weights and biases for the Double-DRQN was solely based on the parameter search performed in the DDQN, due to the time constraints. However, the results presented an overall better performance in the Double-DRQN compared to the DDQN, which implies that with a more thorough tuning of the Double-DRQN, an increased performance can be expected.

7.1.3 What impact has the rollout function and the choice of it?

The results in Chapter 5 indicate that the ACO algorithm successfully guides the agent in choosing favorable actions during training. This action selection method accelerated the training and thereby enabled the use of fewer episodes during training. By not using the rollout function, more episodes were needed. However, this enabled a wider exploration of the environment.

From the results in Section 5.4.1 and 5.4.2, it can be concluded that the best action selection algorithm depends on the network. For the Double-DRQN, when increasing the amount of episodes, guidance by the ACO algorithm does not longer provide the best performance, due to lack of exploration. Further, for the DDQN, the ACO algorithm resulted in the best performance. However, it can be concluded that the amount of episodes during training might not be enough to draw conclusions regarding the best action selection method for DDQN.

The results in Section 5.4.3 indicate that the ACO algorithm performs better than

the Tabu search for all cases with 10 customers. However, for the 20 customer cases, the ACO algorithm outperformed the Tabu search in only one of the two cases. These results were expected since the hyperparameters of the ACO algorithm were tuned solely on case 1 with 10 customers. It can thereby be concluded that the ACO algorithm is an appropriate choice of rollout function for this VRP. However, for the 20 customer cases, a more thorough hyperparameter tuning would be needed in order to outperform the Tabu search on all cases.

7.1.4 Concluding remarks

To summarize the conclusion, it was possible to apply the DDQN and the Double-DRQN as function approximators instead of using the existing reinforcement learning model for this VRP. The existing reinforcement learning model outperforms the DQL-models at this stage, however this might be due to the inital problem formulation and the configuration and tuning of the networks, which therefor should be investigated further. The Double-DRQN outperformed the DDQN in terms of run-time and parameter convergence, but resulted in an approximately equal performance concerning the values of the difference. This might be due to the gradient stabilizing abilities as well as the capturing of the long term dependencies that is enabled by the LSTM layer. Lastly, the ACO algorithm generally outperforms the Tabu search, which can be a result of the Tabu search performing the action selection iteratively.

8

Future work

As previously mentioned, the problem for this thesis was initially posed in the project report [2], which has been a common feature for several challenges during this process. In future work, it would therefore be interesting to see if the performance could be improved by re-formulating the problem, i.e. the MDP, making it more suitable for a DQL application. Another interesting investigation would be to analyse other reduced combinations of state representations. Since it is of interest to investigate a different problem formulation, the reward function and the parameters in it would be formulated differently as well. This would thereby require a completely different state, which would be taken into account when re-formulating the problem.

With more time, it would further be of interest to tune the ACO algorithm, the different combinations of the network parameters, the hyperparameters and the network architecture more thoroughly as well as for several cases. By being able to try several different cases, the generalization capabilities of the network might improve and it might be possible to draw some conclusions on behavioural trends. Furthermore, the tuning in this thesis was solely based on one case with 10 customers in order to find the best network configuration. It would therefor be of interest to implement some kind of grid search algorithm for the hyperparameters to be able to tune several parameters simultaneously. With such an algorithm, performing the search manually can be avoided, enabling to easier draw parallel conclusions from such results as well as taking the time efficiency into account.

It was further concluded that the Double-DRQN performed approximately equally well as the DDQN, however with a lower run-time. The recurrent networks within the VRP seems to be unexplored territory, since most research is based on other neural networks such as pointer networks and convolution networks. It would therefor be interesting to specifically analyse the Double-DRQN further in order to see how one can utilize the recurrent abilities and long term dependencies within the VRP. The VRP can further be seen as a sequential decision making problem. Therefore it would also be interesting to implement an encoder-decoder architecture to the network, since this is an effective framework for solving such kind of problems. Other interesting findings in previous work within the VRP presented the use of policy optimization algorithms such as a policy gradient method instead of Q-learning, to find an optimal policy. It would therefore be of interest to combine such a policy optimization with a recurrent network to evaluate if the network might perform better when not using Q-Learning.

Lastly, implementing neural networks to solve the VRP is a relatively unexplored field with a high potential in interesting findings of effective solution methods. Therefore, the results and conclusions in this thesis should be used as a baseline for further research within this field.

Bibliography

[1] Violino, B. (2020, October 28). IoT and AI boost Volvo Trucks vehicle connectivity. Network World. https://www.networkworld.com/article/3587404/volvo-trucks-boosts-vehicle-connectivity-with-ai-and-iot.html

[2] Basso, R., Kulcsár, B., & Sanchez-Diaz, I. Dynamic Stochastic Electric Vehicle Routing with Safe Reinforcement Learning. Submitted for publication.

[3] Laporte, G. (2007). What you should know about the vehicle routing problem. Naval Research Logistics (NRL), 54(8), 811-819.

[4] Basso, R., Kulcsár, B., & Sanchez-Diaz, I. (2021). Electric vehicle routing problem with machine learning for energy prediction. Transportation Research Part B: Methodological, 145, 24-55.

[5] Lin, J., Zhou, W., & Wolfson, O. (2016). Electric vehicle routing problem. Transportation Research Procedia, 12, 508-521.

[6] Pelletier, S., Jabali, O., & Laporte, G. (2019). The electric vehicle routing problem with energy consumption uncertainty. Transportation Research Part B: Methodological, 126, 225-255.

[7] Othman, W. A. F., Wahab, A. A. A., Alhady, S. S., & Wong, H. N. (2018). Solving Vehicle Routing Problem using Ant Colony Optimisation (ACO) Algorithm. International Journal of Research and Engineering, 5(9), 500-507.

[8] Novoa, C., & Storer, R. (2009). An approximate dynamic programming approach for the vehicle routing problem with stochastic demands. European journal of operational research, 196(2), 509-515.

[9] Nazari, M., Oroojlooy, A., Snyder, L. V., & Takáč, M. (2018). Reinforcement learning for solving the vehicle routing problem. arXiv preprint arXiv:1802.04240.

[10] Pratama, B. P. (2020, October 28). Understanding Markov decision process: The framework behind reinforcement learning. Medium. https://towardsdatascience.com/understanding-markov-decision-process-the-framework-behind-reinforcement-learning-4b5166f3c5b4

[11] Puterman, M. L. (1990). Markov decision processes. Handbooks in operations research and management science, 2, 331-434.

[12] Ashraf, M. (2018, April 11). Reinforcement learning demystified: Markov decision processes (Part 1). Medium. https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690

[13] Kamalzadeh, H., & Hahsler, M. (2019, December 8). POMDP: Introduction to partially observable Markov decision processes. The Comprehensive R Archive Network. https://cran.r-project.org/web/packages/pomdp/vignettes/POMDP.html

[14] Sniedovich, M. (2010). Dynamic programming : Foundations and principles, second edition. ProQuest Ebook Central https://ebookcentral.proquest.com

[15] Held, M., & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied mathematics, 10(1), 196-210.

[16] Kool, W., van Hoof, H., Gromicho, J., & Welling, M. (2021). Deep Policy Dynamic Programming for Vehicle Routing Problems. arXiv preprint arXiv:2102.11756.

[17] Bertsekas, D. P. (2010). Rollout algorithms for discrete optimization: A survey. Handbook of Combinatorial Optimization, D. Zu and P. Pardalos, Eds. Springer.

[18] Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. IEEE computational intelligence magazine, 1(4), 28-39.

[19] Dorigo, M. (2007). Ant colony optimization. Scholarpedia, 2(3), 1461.

[20] Wahde, M. (2008). Biologically inspired optimization methods: an introduction. WIT press.

[21] Glover, F., & Taillard, E. (1993). A user's guide to tabu search. Annals of operations research, 41(1), 1-28.

[22] Hougardy, S., Zaiser, F., & Zhong, X. (2020). The approximation ratio of the 2-Opt Heuristic for the metric Traveling Salesman Problem. Operations Research Letters, 48(4), 401-404.

[23] Liang, F. (2020, July 27). Optimization Techniques — Tabu search. Medium. https://towardsdatascience.com/optimization-techniques-tabu-search-36f197ef8e25

[24] Krenker, A., Bešter, J., & Kos, A. (2011). Introduction to the artificial neural networks. Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech, 1-18.

[25] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1, No. 2). Cambridge: MIT press.

[26] Mehlig, B. (2019). Artificial neural networks. arXiv preprint arXiv:1901.05639.

[27] Smits, J. R. M., Melssen, W. J., Buydens, L. M. C., & Kateman, G. (1994). Using artificial neural networks for solving chemical problems: Part I. Multi-layer feed-forward networks. Chemometrics and Intelligent Laboratory Systems, 22(2), 165-189.

[28] Salimi Asl, A., Erdem, A., & Rafighi, M. (2017). Applying a multi sensor system to predict and simulate the tool wear using of artificial neural networks. Scientia Iranica, 24(6), 2864-2874.

[29] Sharma, S. (2017). Activation functions in neural networks. towards data science, 6.

[30] Little, Z. (2020, May 18). Activation Functions (Linear/Non-linear) in Deep Learning —. Medium. https://xzz201920.medium.com/activation-functions-linear-non-linear-in-deep-learning-relu-sigmoid-softmax-swish-leaky-relu-a6333be712ea

[31] Lu, L., Shin, Y., Su, Y., & Karniadakis, G. E. (2019). Dying relu and initialization: Theory and numerical examples. arXiv preprint arXiv:1903.06733.

[32] Yam, J. Y., & Chow, T. W. (2000). A weight initialization method for improving training speed in feedforward neural network. Neurocomputing, 30(1-4), 219-232.

[33] Kumar, S. K. (2017). On weight initialization in deep neural networks. arXiv preprint arXiv:1704.08863.

[34] Meyerowitz, G. (2018, June 11). Bias Initialization in a Neural Network - Glen Meyerowitz. Medium. https://medium.com/@glenmeyerowitz/bias-initialization-in-a-neural-network-2e5d26fed0f0

[35] Team, K. (n.d.). Keras documentation: Layer weight initializers. Https://Keras.Io/Api/Layers/Initializers/. Retrieved April 16, 2021, from https://keras.io/api/layers/initializers/

[36] Stanford - Spring 2021. (2021b). CS231n Convolutional Neural Networks for Visual Recognition. CS231n: Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/neural-networks-2/

[37] Afrin, S. (2020, June 19). Weight Initialization in Neural Network, inspired by Andrew Ng. Medium. https://medium.com/@safrin1128/weight-initialization-in-neural-network-inspired-by-andrew-ng-e0066dc4a566

[38] Yadav, S. (2020, January 17). Weight Initialization Techniques in Neural Networks. Medium. https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78

[39] Hu, W., Xiao, L., & Pennington, J. (2020). Provable benefit of orthogonal initialization in optimizing deep linear networks. arXiv preprint arXiv:2001.05992.

[40] Gebel, Ł. (2020, November 19). Why We Need Bias in Neural Networks - Towards Data Science. Medium. https://towardsdatascience.com/why-we-need-bias-in-neural-networks-db8f7e07cb98

[41] Wang, Z., & Bovik, A. C. (2009). Mean squared error: Love it or leave it? A new look at signal fidelity measures. IEEE signal processing magazine, 26(1), 98-117.

[42] Willmott, C. J., & Matsuura, K. (2005). Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. Climate research, 30(1), 79-82.

[43] Hastie, T., Tibshirani, R., & Friedman, J. (2009). The elements of statistical learning: data mining, inference, and prediction. Springer Science & Business Media.

[44] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

[45] Sutton, R. S. (1986). Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks. Proceedings of the Eighth Annual Conference of the Cognitive Science Society, : Hillsdale, NJ: Erlbaum.

[46] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural networks, 12(1), 145-151.

[47] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[48] Mittal, A. (2021, March 20). Understanding RNN and LSTM - Aditi Mittal. Medium. https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e

[49] Guo, J. (2013). Backpropagation through time. Unpubl. ms., Harbin Institute of Technology, 40, 1-6.

[50] Grosse, R. (2017). Lecture 15: Exploding and vanishing gradients. University of Toronto Computer Science.

[51] Yu, Y., Si, X., Hu, C., & Zhang, J. (2019). A review of recurrent neural networks: LSTM cells and network architectures. Neural computation, 31(7), 1235-1270.

[52] Sundermeyer, M., Schlüter, R., & Ney, H. (2012). LSTM neural networks for language modeling. In Thirteenth annual conference of the international speech communication association.

[53] Olah, C. (2015, August 27). Understanding LSTM Networks – colah's blog. Https://Colah.Github.Io/Posts/2015-08-Understanding-LSTMs/. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[54] Karakaya, M. (2020, December 16). LSTM: Understanding the number of parameters. Medium. https://medium.com/deep-learning-with-keras/lstm-understanding-the-number-of-parameters-c4e087575756

[55] Vignesh, S. (2020, July 24). The perfect fit for a DNN. Medium. https://medium.com/analytics-vidhya/the-perfect-fit-for-a-dnn-596954c9ea39

[56] Hawkins, D. M. (2004). The problem of overfitting. Journal of chemical information and computer sciences, 44(1), 1-12.

[57] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

[58] AI, S. (2019, February 18). Reinforcement learning algorithms — an intuitive overview. Medium. https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc

[59] Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3-4), 279-292.

[60] Kuleshov, V., & Precup, D. (2014). Algorithms for multi-armed bandit problems. arXiv preprint arXiv:1402.6028.

[61] Peters, J., & Bagnell, J. A. (2010). Policy Gradient Methods. Scholarpedia, 5(11), 3698.

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

[63] A hands-on introduction to deep Q-learning using OpenAI gym in Python. (2020, April 27). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[64] Kobayashi, T., & Ilboudo, W. E. L. (2021). t-soft update of target network for deep reinforcement learning. Neural Networks, 136, 63-71.

[65] Hernandez-Leal, P., Kartal, B., & Taylor, M. E. (2019). A survey and critique of multiagent deep reinforcement learning. Autonomous Agents and Multi-Agent Systems, 33(6), 750-797.

[66] Zhang, S., & Sutton, R. S. (2017). A deeper look at experience replay. arXiv preprint arXiv:1712.01275.

[67] TORRES.AI, J. (2020, December 18). Deep Q-network (DQN)-II. Medium. https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c

[68] Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M., & Dabney, W. (2020, November). Revisiting fundamentals of experience replay. In International Conference on Machine Learning (pp. 3061-3071). PMLR.

[69] Liu, R., & Zou, J. (2018, October). The effects of memory replay in reinforcement learning. In 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton) (pp. 478-485). IEEE.

[70] Hasselt, H. (2010). Double Q-learning. Advances in neural information processing systems, 23, 2613-2621.

[71] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 30, No. 1).

[72] Hausknecht, M., & Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. arXiv preprint arXiv:1507.06527.

[73] Juliani, A. (2018, June 21). Simple Reinforcement Learning with Tensorflow Part 6: Partial Observability and Deep Recurrent Q-Networks. Medium. https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-6-partial-observability-and-deep-recurrent-q-68463e9aeefc

[74] Stathakis, D. (2009). How many hidden layers and nodes?. International Journal of Remote Sensing, 30(8), 2133-2147.

[75] Heaton, J. (2020, July 20). The Number of Hidden Layers. Heaton Research. https://www.heatonresearch.com/2017/06/01/hidden-layers.html

[76] Stützle, T., & Hoos, H. H. (1996). Improving the Ant System: A detailed report on the MAX–MIN Ant System. FG Intellektik, FB Informatik, TU Darmstadt, Germany, Tech. Rep. AIDA–96–12.

A

Appendix 1

Algorithm 1 Pseudocode ADAM algorithm [47]

Require: A small constant for numerical stability ϵ, set to default $1e^7$
Require: β_1,$\beta_2 \in [0, 1)$, the exponential decay rate for the 1st and 2nd moment estimates. Set to default, which is 0.9 respectively 0.999
Require: Stepsize α, set to default 0.001
Require: Stochastic objective function $f(\theta)$, with parameters θ
Require: Initial parameter vector θ_0

$m_0 \leftarrow 0$ (Initialize 1$^{\text{st}}$ moment vector)
$v_0 \leftarrow 0$ (Initialize 2$^{\text{nd}}$ moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t *not converged* **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t (\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$ (Update parameters)
end
return θ_t (resulting parameters)

B

Appendix 2

Algorithm 2 Pseudocode Forward Propagation algorithm [26]

Input: N-dimensional data, $\mathbf{x}^\mu = [x_1^\mu, \ldots, x_N^\mu]^T$

for $l := 1 \leftarrow$ *# Hidden layers* **do**

  for $j := 1 \leftarrow$ *# Neurons in layer l* **do**

    if *l=1* **then**

      $V_j^\mu = \phi(\sum_k w_{jk} x_k^\mu - b_j)$

      - V_j^μ is the activation from neuron j
      - ϕ is an activation function
      - w_{jk} is the weight between x_k and neuron j
      - b_j is the bias for neuron j

    end

    else

      $V_j^\mu = \phi(\sum_k w_{jk} V_k^\mu - b_j)$

      - V_j^μ is the activation from neuron j
      - V_k^μ is activations from the neurons in layer *l-1*
      - w_{jk} is the weight between neuron j in layer l and neuron k in layer *l-1*

    end

  end

end

Output: $y_i^\mu = \phi(\sum_j W_{ij} V_j^\mu - B_i)$

- V_j^μ is the activation from the neuron j in the last hidden layer
- W_{ij} is the weight between neuron i in the output layer and neuron j in the last hidden layer
- B_i is the bias for neuron j

C
Appendix 3

Algorithm 3 Pseudocode Backward Propagation algorithm [26]

for $m := 1 \leftarrow$ *# Neurons in output layer* **do**

1: $\delta W_{mn} = \alpha \frac{\partial J(t,y(x))}{\partial W_{mn}} = \alpha \sum_{\mu=1}^{p} \Delta_m^\mu V_n^\mu$
2: $\delta B_m = -\alpha \frac{\partial J(t,y(x))}{\partial \Theta_m} = -\alpha \sum_{\mu=1}^{p} \Delta_m^\mu$
3: with $\Delta_m^\mu = (t_m^\mu - O_m^\mu) g'(\sum_n W_{mn} V_n^\mu - B_m)$

- *g'* is the gradient of the activation function ϕ
- y_m^μ is the output from neuron m
- V_n^μ is the activation of neuron n in layer *l+1*, from forward propagation
- t_m^μ is the target output from neuron m
- α is the learning rate
- δW_{mn} is the error term for the weight between neuron m in the output layer and neuron n in layer *l+1*
- δB_m is the bias of neuron m in the output layer
- J is the loss function

end

for $l :=$ *# Hidden layers* $\leftarrow 1$ **do**

for $m := 1 \leftarrow$ *# Neurons in layer l* **do**

1: $\delta w_{mn} = \alpha \frac{\partial J(t,y(x))}{\partial w_{mn}} = \alpha \sum_{\mu=1}^{p} \delta_m^\mu V_n^\mu$
2: $\delta b_m = -\alpha \frac{\partial J(t,y(x))}{\partial \theta_m} = -\alpha \sum_{\mu=1}^{p} \delta_m^\mu$
3: with $\delta_m^\mu = \sum_i \delta_i^\mu w_{im} g'(\sum_n w_{mn} V_n^\mu - b_m)$

- δ_i^μ is the error term from neuron i in previous layer *l-1*
- V_n^μ is the activation for neuron n in layer *l+1*, from forward propagation
- δw_{mn} is the error term for the weight between neuron m in layer l and neuron n in layer *l+1*
- δb_m is the error term for the bias of neuron m in layer l

end

end

for $l := 1 \leftarrow$ *# Hidden layers* **do**

for $m := 1 \leftarrow$ *# Neurons in layer l* **do**

Update hidden weights and biases as:

1: $w_{mn} = w_{mn} + \delta w_{mn}$
2: $b_m = b_m + \delta b_m$

end

end

for $m := 1 \leftarrow$ *# Neurons in output layer* **do**

Update output weights and biases as:

1: $W_{mn} = W_{mn} + \delta W_{mn}$
2: $B_m = B_m + \delta B_m$

end

D

Appendix 4

Algorithm 4 Pseudocode Epsilon-greedy algorithm [60]

Require: A random number $r \in (0, 1)$
Require: A trade-off factor or probability $\epsilon \in (0, 1)$, set to a value appropriate to the problem
if $r < \epsilon$ **then**
  select action a randomly
end
else
  select action a based on the Q-values as: $\underset{a}{max} Q(s, a)$
end

E

Appendix 5

Algorithm 5 Pseudocode Deep Q-Learning algorithm [62]

Initialize: replay buffer D, of size M
Initialize: prediction network, with parameters θ
Initialize: target network, with the same architecture as the prediction network and parameters $\phi = \theta$
for *Episode* $i := 1 \rightarrow I$ **do**
  Initialize: initial state s_1
  for *Epoch* $n := 1 \rightarrow N$ **do**
    1. Choose action a_n, based on for example the ϵ-greedy policy
    2. Execute action a_n, observe state s_{n+1} and receive reward r_n
    3. Store the transition (s_n,a_n,r_n,s_{n+1}) in D
    4. Sample random batch (s_j, a_j,r_j,s_{j+1}) of transitions from D, with $j = 1, ..., m$ where m is the batch size
    5. Set target

$$t_j = \begin{cases} r_j, & \text{if the episode terminates at step j+1,} \\ r_j + \gamma \underset{a_j+1}{max} Q(s_{j+1}, a_{j+1}; \phi), & \text{otherwise.} \end{cases} \tag{E.1}$$

    6. Perform an optimization step on $J(t_j - Q(s_j, a_j; \theta))$
    7. Update target network parameters ϕ
  end
end

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se