MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

“Kazakh-British Technical University” JSC

School of Mathematics and Cybernetics

ADMITTED TO DEFENCE
Head of School of Mathematics
and Cybernetics,
Ph.D., Assistant Professor
_____ L. O. Sarybekova
“____”_____ 20___

EXPLANATORY NOTE
TO GRADUATION PROJECT (work)

Theme: “Self-learning neural network based traffic signal controller for an isolated intersection and construction of a new clustering algorithm in unsupervised machine learning”

Supervisor

Professor,

Doctor of Math and Physics Sciences

_____ N. S. Dairbekov

“____”_____ 20___

Student

_____ R. R. Mussabayev

“____”_____ 20___

Major: 5B070500

“Mathematical and Computer Modeling”

Norms compliance monitor

Manager

_____ A. A. Oirat

“____”_____ 20___

Almaty, 2018

MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

“Kazakh-British Technical University” JSC

School of Mathematics and Cybernetics

Speciality 5B070500 “Mathematical and Computer Modeling”

APPROVED
Head of School of
Mathematics and Cybernetics,
Ph.D., Assistant Professor
_____ L. O. Sarybekova
“____”_____ 20___

DIPLOMA PROJECT ASSIGNMENT

Student: R. R. Mussabayev

Project title: “Self-learning neural network based traffic signal controller for an isolated intersection and construction of a new clustering algorithm in unsupervised machine learning”

Approved by the KBTU order: № _____ dated “_____” _____

Submission deadline: _____

List of issues addressed in the diploma project or its brief content:

1. Reinforcement learning
2. Markov decision processes
3. Problem statement and proposed solutions
4. Experiments, implementation and improvement prospects
5. Construction of a new clustering algorithm

List of graphical documents:

1. Figures (19)
2. Tables (2)

Consultations regarding the project with indication of respective reasons:

Section	Consultant (academic degree, title)	Timeline	Signature
Main	N. S. Dairbekov Professor, Doctor of Math and Physics Sciences	12.01 – 15.03	
Special	N. S. Dairbekov Professor, Doctor of Math and Physics Sciences	16.03 – 18.05	

Date of assignment receipt "____" _____ 20___

Head of School of
Mathematics and Cybernetics _____ L. O. Sarybekova
(signature)

Project supervisor _____ N. S. Dairbekov
(signature)

Student _____ R. R. Mussabayev
(signature)

Date "____" _____ 20___

MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

"Kazakh-British Technical University" JSC

School of Mathematics and Cybernetics

APPROVED
Head of School of
Mathematics and Cybernetics,
Ph.D., Assistant Professor
_____ L. O. Sarybekova
"____"_____ 20___

THE PLANNED PERFORMANCE SCHEDULE OF GRADUATION PROJECT

Student: R. R. Mussabayev

Major: "Mathematical and Computer Modeling"

Theme of graduation work: "Self-learning neural network based traffic signal controller for an isolated intersection and construction of a new clustering algorithm in unsupervised machine learning"

Supervisor: Professor, Doctor of Math and Physics Sciences, N. S. Dairbekov

Type of work	Time period
• Choosing the theme; • Drafting a work schedule for the implementation of the graduation project with specific deadlines.	*January, 2018*
• Gathering information: - Review of the literature; - Review of the online sources.	*January, 2018*

• Receiving and discussing the graduation work tasks; • Determining the purposes and tasks of the graduation work, statement of the problem; • Survey of the existing analogs of differential geometry on discrete spaces; • Attempt to devise a new analog of differential geometry on discrete spaces.	*February, 2018*
• Working on the direct method of training on history; • Implementation of the direct method in Python; • Working on the clustering algorithm; • Implementation of the clustering algorithm in MATLAB.	*March, 2018*
• Learning about reinforcement learning; • Working on the Shallow Q-Network (SQN) algorithm; • Implementation of the SQN algorithm in Python; • Working on the Average Reward Network (ARN) algorithm; • Implementation of the ARN algorithm in Python; • Preliminary defense of the graduation project.	*April, 2018*
• Preparation of the explanatory note; • Getting feedback reviews.	*May, 2018*

Head of School of
Mathematics and Cybernetics _____ L. O. Sarybekova
(signature)

Project supervisor _____ N. S. Dairbekov
(signature)

Student _____ R. R. Mussabayev
(signature)

Project sheet

№	Format	Designation	Denomination	Quantity	Note
1	A4		Explanatory note	1	68 p.
2	pdf		Electronic copy of the diploma project	1	
3	pdf		Presentation files	1	
4	zip		Python and MATLAB source files	1	

ABSTRACT

The explanatory note contains 68 pages of typewritten text, 2 tables, 19 figures, the list of literature — 5 references and 1 appendix.

TRAFFIC SIGNAL CONTROLLER, REINFORCEMENT LEARNING, MACHINE LEARNING, CLUSTERING ALGORITHM, TEMPORAL-DIFFERENCE LEARNING, REWARD FORMULA.

The objective of the current research project is to build an adaptive traffic signal controller for an isolated intersection. The problem is formulated as a reinforcement learning task. The proposed solutions are based on an artificial neural network. In contrast with other studies in the field, we use queue length rather than the average delay as a measure of performance. A state is represented by the number of queuing vehicles before the intersection. Also, we propose to use a new reward formula consisting of the queue reduction and equilibrium terms. In this work we propose three different algorithms of training the neural network. We give a full theoretical explanation and conduct an experiment in the simulated environment for each of the proposed algorithms. Also, we propose the idea of a new clustering algorithm, and give intermediate results of the experiments on a test dataset.

АҢДАТПА

Дипломдық жоба машинамен басылған 68 беттен, 2 кестеден, 19 суреттен, 5 пайдаланылған әдебиеттен және 1 қосымшадан тұрады.

БАҒДАРШАМ БАСҚАРУШЫСЫ, НЫҒАЙТУМЕН ҮЙРЕНУ, МАШИНАЛЫҚ ОҚЫТУ, КЛАСТЕРЛЕУ АЛГОРИТМІ, УАҚЫТША АЙЫРМАШЫЛЫҚПЕН ОҚЫТУ, СЫЙАҚЫ ФОРМУЛАСЫ.

Осы зерттеу жұмысының мақсаты жекеленген қиылысқа адаптивті бағдаршам басқарушысын жасау болып табылады. Тапсырма нығайтумен үйрену тапсырмасы ретінде қойылған. Ұсынылған шешім әдістері нейрондық желіге негізделген. Осы саладағы басқа зерттеулерден ерекшелігі алгоритмнің орындалу көрсеткіші ретінде орташа күту уақытынан гөрі тұрақты машиналардың кезегінің ұзындығын қолданамыз. Қиылыстың жағдайы ретінде қиылыс алдында тұрған көліктердің саны алынады. Біз, сондай-ақ, екі бөліктен тұратын жаңа сыйақы формуласын ұсынамыз: тұрақты көліктердің жалпы санын қысқартуға жауапты қосылғыш және бағыттар арасындағы теңгерімді сақтауға жауапты қосылғыш. Осы жұмыста біз нейрондық желіні оқытуға арналған үш түрлі алгоритмді ұсынамыз. Біз ұсынылған алгоритмдердің әрқайсысына толық теориялық негіздемені келтіреміз және эксперименттер жүргіземіз. Біз, сондай-ақ, жаңа кластерлеу алгоритмін ұсынамыз және деректердің тәжірибе жиынтығы бойынша эксперименттердің аралық нәтижелерін келтіреміз.

АННОТАЦИЯ

Дипломная работа содержит 68 страниц машинописного текста, 2 таблицы, 19 рисунков, список использованных источников — 5 наименований и 1 приложение.

КОНТРОЛЛЕР СВЕТОФОРА, ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ, МАШИННОЕ ОБУЧЕНИЕ, АЛГОРИТМ КЛАСТЕРИЗАЦИИ, ОБУЧЕНИЕ ВРЕМЕННЫМИ РАЗНОСТЯМИ, ФОРМУЛА ВОЗНАГРАЖДЕНИЯ.

Целью данной исследовательской работы является создание адаптивного контроллера светофора для изолированного перекрестка. Задача поставлена как задача обучения с подкреплением. Предложенные методы решения основаны на нейронной сети. В отличие от других исследований в данной области, в качестве показателя производительности алгоритма мы используем длину очереди стоящих машин, а не среднее время ожидания. Состояние перекрестка представлено количеством машин, стоящих перед перекрестком. Также мы предлагаем использовать новую формулу вознаграждения, состоящую из двух частей: слагаемого, отвечающего за уменьшение общего количества стоящих машин, и слагаемого, отвечающего за сохранения баланса между направлениями. В данной работе мы предлагаем три разных алгоритма обучения нейронной сети. Мы приводим полное теоретическое обоснование и проводим эксперименты для каждого из предлагаемых алгоритмов. Также мы предлагаем новый алгоритм кластеризации и приводим промежуточные результаты экспериментов на тестовом наборе данных.

CONTENTS

INTRODUCTION 12

1 REINFORCEMENT LEARNING 13

1.1 The agent and the environment 13

1.2 Elements of reinforcement learning 14

1.3 Machine learning and reinforcement learning. Main machine learning paradigms 16

1.4 Exploration-exploitation dilemma 19

2 MARKOV DECISION PROCESSES 21

2.1 The agent-environment interface 21

2.2 Goal and reward 23

2.3 Returns and episodes 23

2.4 Policies and value functions 25

2.5 Optimal policies and optimal value functions 27

3 PROBLEM STATEMENT AND PROPOSED SOLUTIONS 30

3.1 Problem statement 30

3.2 State space 30

3.3 Action space 32

3.4 Reward formula 33

3.5 Temporal-difference learning 34

3.6 Shallow Q-Network (SQN) 37

3.7 Average Reward Network (ARN) 39
3.8 Direct method of training on history 40

4 EXPERIMENTS, IMPLEMENTATION NOTES AND IMPROVEMENT PROSPECTS 44
4.1 Shallow Q-Network (SQN). Implementation 44
4.2 Average Reward Network (ARN). Implementation 49
4.3 Direct method of training on history. Implementation 52

5 CONSTRUCTION OF A NEW CLUSTERING ALGORITHM . . . 56
5.1 Problem statement 56
5.2 Local density function 57
5.3 Experiments 58

CONCLUSION 59

REFERENCES 60

APPENDIX 61

INTRODUCTION

Nowadays traffic congestion is a big problem. A fixed signal plan of a traffic light neglects a changing demand in the intersection, location of the intersection, time, weather conditions, possibility of an emergency, etc. If the traffic light did take into account such useful information, the control would be much more efficient. However, it is not clear how exactly the traffic controller should process this information and adjust the current signal plan accordingly.

Obviously, the traffic controller cannot rely on the expertise of a human: the interplay between the above-mentioned factors is quite intricate, and the presence of a human expert cannot be guaranteed at all times. One solution is to let the controller *learn on its own*. The controller should be able to select the optimal signal plan based on the previous and incoming experience. This is why *reinforcement learning* with *artificial neural networks* has been used to solve the problem of optimal control. Reinforcement learning studies how to teach an agent to choose optimal actions from interaction with the environment. An artificial neural network is a powerful tool which is able to understand how the input factors are related to each other, provided that the neural network is taught properly.

In the subsequent chapters we will develop the necessary framework of reinforcement learning, provide a detailed problem statement, and describe the proposed algorithms along with their implementation in a simulated environment.

1 REINFORCEMENT LEARNING

In this chapter we will describe the basic concepts and definitions of reinforcement learning which are essential for understanding the proposed solutions. In doing so, we follow the book "Reinforcement Learning: An Introduction" by Richard S. Sutton and Andrew G. Barto [1].

1.1 The agent and the environment

The *trial-and-error approach* is at the core of reinforcement learning and fundamental to the concept of learning in general. A child learns about the world by the consequences that his actions entail. A person learns to build deep emotional connections with other people by trying to behave in a number of different ways, saying various things and carefully watching for how people react. In both cases, based on whether the performed action is followed by a negative or positive response, the actor adjusts his behavior accordingly with the goal of increasing the total positive response he will get over the future. This kind of learning happens through interaction of an *agent* with the *environment*. To put it simply, *reinforcement learning* is about how to teach the agent to interact efficiently with its environment with the goal of maximizing future positive responses from the environment. Abstracting away from specific cases, in what follows we are going to formalize the problem of reinforcement learning and develop a mathematically rigorous framework to solve it.

A learning agent is able to sense the state of the environment in part or in whole, and is able to take actions that affect the environment and, therefore, its subsequent states. The agent also must have a goal or goals relating to the state of the environment. For instance, a trash collecting mobile robot seeks to keep the room as clean as possible, while trying to reduce energy consumption to a

minimum. This robot observes the floor in a relatively small neighborhood and that information serves as a state. Also it can perceive when it successfully picks up a garbage (a positive response), and when it hits an obstacle or somebody yells at it (negative responses). Based on the incoming responses and despite the limited knowledge about the environment, the agent should increase its performance on the goals. The environment is often nondeterministic in the sense that the effects of actions cannot be fully computed ahead of time and the agent is obliged to monitor its environment frequently and react appropriately. For example, the above-mentioned garbage collector faces uncertainty about the result of moving in a particular direction, especially at early times. However, as the time goes by, with increasing experience the uncertainty decreases, and in a given state the agent starts to favor some actions over the others.

In each situation the agent ought to not only choose actions which are beneficial to immediately enhance the situation, but it should also be prudent to take into account how an immediate action choice will affect the options and opportunities available to the agent at later times. For instance, a chess player can make a move that gets an opponent's figure out yet ruin himself due to a bad strategy. Thus, the correct action choice must always consider consequences of actions and involve foresighted planning.

1.2 Elements of reinforcement learning

Let us introduce some other essential notions of reinforcement learning: a *policy*, a *reward*, a *value function* and a *model* of the environment.

A *policy* completely determines the agent's way of behaving, i.e. it tells what action is best to choose at a particular state. A policy can be represented by various means, e.g. a table or by a function approximator such as artificial neural network. To find a good policy is the goal of reinforcement learning algorithms.

A *reward* is a number which is sent by the environment to the agent at each time step and indicates what are the good and bad events for the agent. The reward signal is analogous to the sensation of pleasure or pain. Looking at the reward signal that follows the performed action, the agent changes its policy in order to increase performance on that situation later on. The sole objective of the reinforcement learning agent is to accumulate the maximum possible total reward over the long run.

In contrast to the immediate property of the reward signal, a *value function* indicates what is good in the long run. More specifically, the *value* of a state is the total accumulated reward that the agent can expect to receive over the future, and, hence, shows the long term desirability for the agent to be in that state. The value function of a state considers the states that are likely to follow and the rewards that will be available in those states. Therefore, the agent must strive to be in states of high value in order to achieve its goal of maximizing future reward. Maximizing only immediate reward at each step is clearly not sufficient — it is perfectly possible that the transition to a next state comes with a high reward, but henceforth is regularly followed by other states that yield low rewards. Or the reverse could be true. Hence, action choices should be made based on value judgments — the agent should prefer the action leading to a state of highest value, not highest immediate reward. Estimating value functions is crucial for the reinforcement learning algorithms and the sole purpose of estimating value functions is to achieve more reward in the future. Therefore, in that sense rewards are primary, whereas values, as predictions of rewards, are secondary.

A *model* of the environment is the information about it sufficient for inferences to be made about how the environment will behave. For instance, if the model is known and the current state and action are given, the model might predict the resultant next state and next reward. We can use model for plan-

ning, i.e. decide on a course of action by considering possible future situations. Methods for solving reinforcement learning problems that make use of a model of the environment are called *model-based* methods, as opposed to *model-free* methods that are trial-and-error learners which use no planning on choosing the action.

1.3 Machine learning and reinforcement learning. Main machine learning paradigms

Even amongst experienced machine learning pratitioners there is not a universal consensus on what is and what is not *machine learning*. However, let us consider a couple of ways people have tried to define it [2].

Here is the definition of what is machine learning due to Arthur Samuel. *Machine learning* is the field of study that gives computers the ability to learn without being explicitly programmed. This definition was given back in 1950s when Arthur Samuel wrote a checkers playing program which by playing tens of thousands of games against itself learned how to play checkers better than Arthur Samuel himself was able to. By watching what board positions tended to lead to wins, and what sort of board positions tended to lead to losses, the checkers playing program learned over time what are good and bad board positions and eventually outperformed Arthur Samuel himself. This was an outstanding result at the time. Arthur Samuel's definition is somewhat informal and an older one.

Here is a slightly more recent definition by Tom Mitchell. He defines *machine learning* by saying what is a *well-posed learning problem* first. A computer program is said to *learn* from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. For the checkers playing example the experience E will be the experience of having to play tens of thousands of games against

itself. The task T will be the task of playing checkers. And the performance measure P will be the probability that it wins the next game of checkers against some new opponent.

There are several different types of learning algorithms. The main three types are *supervised learning*, *unsupervised learning* and *reinforcement learning*.

Supervised learning is the most frequently studied kind of learning. In *supervised learning* we explicitly teach the computer on a set of labeled examples provided by an educated external supervisor, knowing that this data represents completely correct past behavior. The objective for this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. For instance, let us suppose we are given a house selling database in which each record is represented by a set of features of the house (e.g. size, number of bedrooms, etc.) and the label with a price the house has been sold for. Given this database as a training set, the supervised learning algorithm will be able to predict the most appropriate price for a new house with its features close, but not exactly equal to those contained in the training database. This is an example of *regression problem* in which the learning algorithm outputs a continuous value. There is also a *classification problem* in which the algorithm, given a set of features characterizing the object, is expected to tell to what class out of a finite number of classes the object belongs to. For example, let us say we have a dataset of medical records in which each entry is specified by the breast tumor size and is labeled to be either malignant or benign. Also, let us suppose we have a new patient who tragically has a breast tumor, and by looking at the size of the tumor we would like to determine whether she has a cancer or not. By learning on the dataset the algorithm will be able to classify this particular tumor to be either malignant or benign to a certain extent of accuracy.

As for *unsupervised learning*, roughly speaking, it is when we let the com-

puter learn by itself. In other words, unsupervised learning is concerned with finding structure hidden in collections of unlabeled data. Given a dataset of objects that have no labels at all, the unsupervised machine learning algorithm may break these data into separate *clusters* — groups of objects that are in some sense similar to one another. This is called a *clustering algorithm*. One example where clustering is used is in Google News where the search engine automatically groups into one cluster all the websites on the web that are related to a particular event. A user interested in reading about the event is free to choose any website suggested by the engine. Another examples might be processing astronomical data or identifying cohesive groups of friends based on their activity in a social network. For the example of supervised and unsupervised learning see Figure 1.

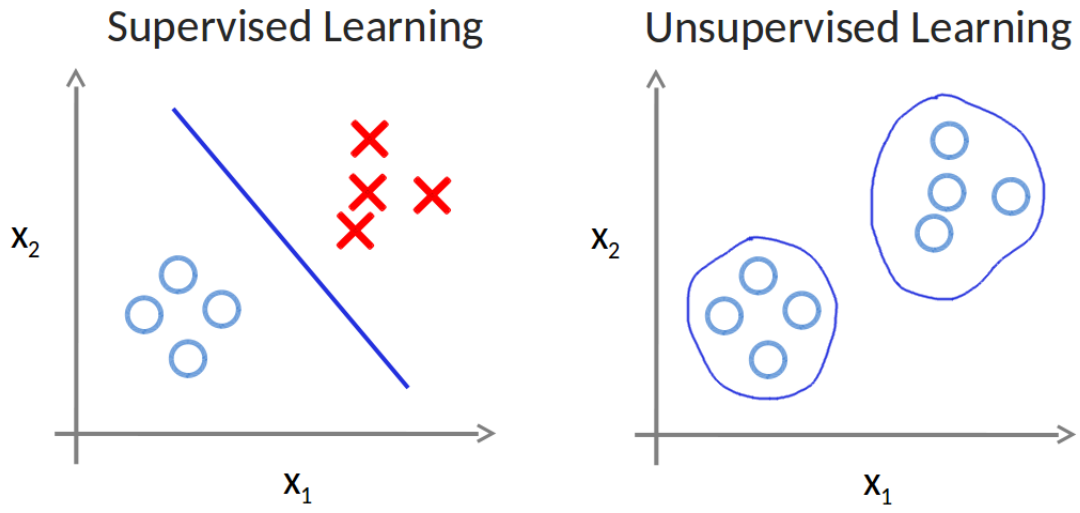


Figure 1: Example of a supervised (left) and unsupervised (right) problem settings. Each training example is characterized by a feature 2-vector $x = (x_1, x_2) \in \mathbb{R}^2$.

Reinforcement learning falls into neither of the two above-mentioned categories of learning. It is not a supervised learning, since we don't have a prede-

fined exemplary dataset of correct behavior of an agent within the environment. However, in the current work we will consider a solution method which uses a history of the past behavior of the agent in order to train the policy function, and this approach might be viewed as supervised learning to some extent. Nevertheless, since in general reinforcement learning problems are interactive in nature, it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. Reinforcement learning is also different from unsupervised learning, for the goal of reinforcement learning is not to find a hidden structure in the agent's behavior, but rather to maximize the reward signal. Revealing structure in an agent's experience might certainly be useful, yet by itself it does not address the issue of maximizing a reward signal. Thus, reinforcement learning is often considered to be a third machine learning paradigm.

1.4 Exploration-exploitation dilemma

One of the distinctive challenges of reinforcement learning is the trade-off between *exploration* and *exploitation*. In order to maximize a reward the agent should prefer the actions it has tried in the past and found to be effective in producing a high reward in a particular state. In other words, the agent has to *exploit* the knowledge it has collected about the states it has seen and the good actions available in those states which it has performed. On the other hand, to discover such actions, the agent has to endeavor trying actions it has never selected before, i.e. *explore* new and, possibly better, action selections to ensure higher reward. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task [1, Sec 1.1]. The agent must try a variety of actions and progressively favor those that appear to be best. The exploration-exploitation dilemma has been studied my mathematicians for a long time, yet still remains unresolved. Maintaining the balance between

exploration and exploitation is an essential part of every reinforcement learning algorithm.

2 MARKOV DECISION PROCESSES

In this chapter we will consider a special mathematical framework called *Markov decision process (MDP)* that is well suited for describing the reinforcement learning problems. Throughout the rest of the work we will be using the machinery of MDP to formulate and solve the main problem of the current project.

2.1 The agent-environment interface

The learner and decision maker is called the *agent* [1, Sec 3.1]. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions. See Figure 2.

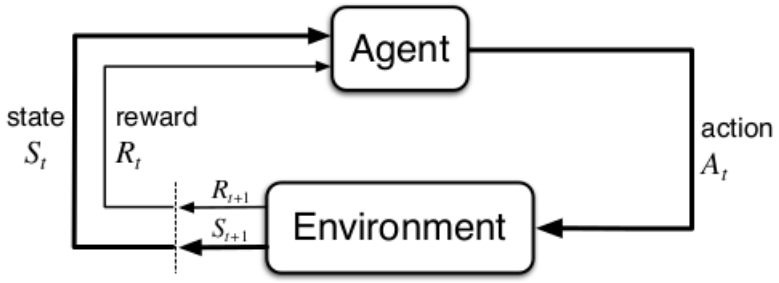


Figure 2: The agent-environment interaction in a Markov decision process

The time is divided into discrete time steps: $t = 0, 1, 2, 3, \ldots$. At each time step t the agent receives a representation of the environment's *state* $S_t \in \mathcal{S}$, and thereafter selects an action $A_t \in \mathcal{A}$ according to some action selection rule.

One time step later, after the selected action has influenced the environment, the agent is given a numerical reward $R_{t+1} \in \mathcal{R}$ and finds itself in a new state S_{t+1}:

$$S_t \underset{A_t}{\longrightarrow} (R_{t+1}, S_{t+1}) \tag{2.1}$$

The continual interaction in form of (2.1) between the agent and the environment gives rise to the following sequence or *trajectory*:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.2}$$

MDP is *finite* if the sets of states, actions and rewards ($\mathcal{S}$,$\mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements. In the problem statement of the current project the set of states $\mathcal{S}$ will be assumed to be unbounded, but, since actually there can be only a finite number of vehicles waiting in line before the intersection, the set of states $\mathcal{S}$ will also be finite, hence the set of rewards $\mathcal{R}$ will be finite as well. In the case of a finite MDP, the random variables $\mathcal{R}_t$ and $\mathcal{S}_t$ have well defined discrete probability distributions dependent only on the preceding state and action. That is, for a particular choice of values $s' \in \mathcal{S}$ and $r \in \mathcal{R}$ there exists a conditional probability of those values occurring at time t, given particular values of the preceding state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.3}$$

Obviously the following equality holds for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1 \tag{2.4}$$

The function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is an ordinary deterministic function of four arguments. The bundle of probabilities given by the function p completely characterize the dynamics of a finite MDP and, therefore, constitute a *model* of the environment.

2.2 Goal and reward

At each time step, the environment passes to the agent a special reward signal — a simple number $R_t \in \mathcal{R} \subset \mathbb{R}$. The purpose of the agent is to maximize the total amount of reward it receives. In order to do this, the agent needs to not only maximize immediate reward, but cumulative reward in the long run. Let us clearly state this idea: the purpose or goal of the agent is the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

2.3 Returns and episodes

As yet we have stated that the objective of learning is maximization of the total cumulative reward the agent obtains in the long run. In order to formalize this idea mathematically, let us introduce the notion of a *return*. The return for an *episodic* task is simply the sum of the observed rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T, \tag{2.5}$$

where T is a final time step. By an *episodic* task we mean a task that naturally can be broken into a series of episodes, such as plays of a game, trips through a maze, or any sort of repeated interaction. Each episode consists of a certain number of steps and terminates on a special state called the *terminal state*. All episodes are independent from one another in the sense that the result of an episode has no effect on the beginning of the subsequent one.

On the other hand, in many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually without limit [1, Sec 3.3]. We call these *continuing tasks*. The return formulation in form of (2.5) will not be suitable for continuing tasks, since there is no terminal state and $T = \infty$, so the return, which is what we are trying to maximize, could itself easily be infinite. For example, suppose the agent receives a reward of +1

at each time step.

The additional concept that will be useful in addressing the issue of an infinite return for a continuing task is that of *discounting*. According to it, in a continuing task it would be sensible to try to maximize the expected value of the *discounted return* which is defined as follows:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.6}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

The discount rate gives weight to the future rewards: a reward observed k time steps later in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If the discount rate $\gamma < 1$, the infinite series in (2.6) has a finite sum so long as the reward sequence $\{R_k\}$ is bounded. Indeed, if $\exists M \in \mathbb{R},\ \forall k \in \mathbb{N},\ |R_k| \leq M$, then:

$$\begin{aligned} |G_t| \quad &\doteq |R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots| = \\ &\leq |R_{t+1}| + \gamma |R_{t+2}| + \gamma^2 |R_{t+3}| + \ldots = \\ &\leq M\left(1 + \gamma + \gamma^2 + \ldots\right) = \\ &= \frac{M}{1-\gamma} \end{aligned} \tag{2.7}$$

From (2.7) it follows that the return G_t is finite. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose A_t so as to maximize only R_{t+1}. If each of the agents actions happened to influence only the immediate reward, not future rewards as well, then a myopic agent could maximize (2.6) by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Returns at successive time steps are related to each other in a way that

is of quintessential importance for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.8}$$

2.4 Policies and value functions

Almost all reinforcement learning algorithms involve estimating value functions — functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state) [1, Sec 3.5]. The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

A *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is merely the probability that $A_t = a$ if $S_t = s$. That is,

$$\pi(a|s) \doteq \Pr\{A_t = a \mid S_t = s\} \tag{2.9}$$

The *value* of a state s under a policy π, denoted $v_\pi(s)$ is the expected return when starting in s and following π thereafter. For MDP's, we can define v_π formally by

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \end{aligned} \tag{2.10}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. We call the function v_π the *state-value function for policy* π. In expanding the formula (2.10), we have used the property of linearity for mathematical expectations. The expression at the end of the formula (2.10) was obtained by writing the expectation by means of transition and policy probabilities. From (2.10) it is seen that state-value functions satisfy a fundamental recursive property similar to the one we have established for the return. Equation (2.10) is the *Bellman equation* for v_π. It expresses a relationship between the value of a state and the values of its successor states. The value function v_π is the unique solution to its Bellman equation.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.11}$$

We call q_π the *action-value function* for policy π.

Note that the formula (2.11) gives a way to compute the action-value function in terms of the state-value functions of all states. Also, if the probability distribution of the given policy π is known, then we can express the state-value function of a particular state in terms of action-value functions as follows:

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \\ &= \sum_a \pi(a|s) q_\pi(s, a) \end{aligned} \tag{2.12}$$

The value functions v_π and q_π can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average by the law of large numbers will converge to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values $q_\pi(s, a)$. We call estimation methods of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns. Of course, if there are very many states, just as in the problem considered in the current work, it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain v_π and q_π as parametrized functions (with fewer parameters than states) and adjust the parameters to better match the observed returns. This can also produce accurate estimates, although much depends on the nature of the parameterized function approximator. In the current project we will be using an *artificial neural network* as the parametrized function approximator.

2.5 Optimal policies and optimal value functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the following way [1, Sec 3.6]. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called

the *optimal state-value function*, denoted v_*, and defined as

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.13}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_*, and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \tag{2.14}$$

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

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{2.15}$$

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (2.10). Because it is the optimal value function, however, v_*'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for v_*, or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) \quad &= \max_{a \in \mathcal{A}} q_{\pi_*}(s,a) = \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] = \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] = \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] = \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.16}$$

The last two equations in (2.16) are two forms of the Bellman optimality equation for v_* and they are independent of the policy. The Bellman optimality equation for q_* is

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] = \\ &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] = \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{2.17}$$

For finite MDPs, the Bellman optimality equation for v_* (2.16) has a unique solution independent of the policy. Indeed, the Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then (2.16) defines n equations in n unknowns. If the dynamics p of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of nonlinear equations. One can solve a related set of equations for q_*.

Once we have the optimal v_* for all states, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation for v_* (2.16). Any policy that assigns nonzero probability only to these actions is an optimal policy. You can think of this as a one-step search. If you have the optimal value function, v_*, then the actions that appear best after a one-step search will be optimal actions.

Having the optimal q_* makes choosing optimal actions even easier. With q_* the agent does not even have to do a one-step-ahead search: for any state s, it can simply choose any action that maximizes $q_*(s, a)$. That is to say, the deterministic *greedy policy* is the optimal one:

$$\pi_*(s) \doteq \operatorname*{argmax}_a q_*(s, a) \tag{2.18}$$

3 PROBLEM STATEMENT AND PROPOSED SOLUTIONS

In this chapter we will state the problem of the current project using the framework of reinforcement learning developed in the preceding chapters. We will also describe three different approaches that have been tried to solve the problem, namely, SQN (Shallow Q-Network), ARN (Average Reward Network) and the direct method of training the neural network on the history of occurred states and selected actions.

3.1 Problem statement

The goal is to build an adaptive traffic signal controller for an isolated intersection. In reinforcement learning model the *agent* is the traffic signal controller. The *environment* is the road junction. The task of the controller is to reduce the total number of vehicles waiting in line before the intersection, keeping the balance between the conflicting directions. The controller should be adaptive, i.e. it should be constantly reacting to the changing nonstationary environment. One of the proposed methods, SQN, is suitable to be extended to a transport network of arbitrary size, and this is one of the major generalizations that we anticipate to make in the future.

3.2 State space

The state signal represents the aggregated number of queuing vehicles in non-conflicting directions [3]. The links D10 and D30 are considered to have non-conflicting directions, so do D20 and D40. Consequently, at each time step the state signal is represented by a 2-vector of values as follows:

$$NS \doteq D10 + D30 \tag{3.1}$$

$$WE \doteq D40 + D20 \tag{3.2}$$

$$S \doteq (NS, WE) \tag{3.3}$$

See Figure 3 for a schematic view of the road junction.

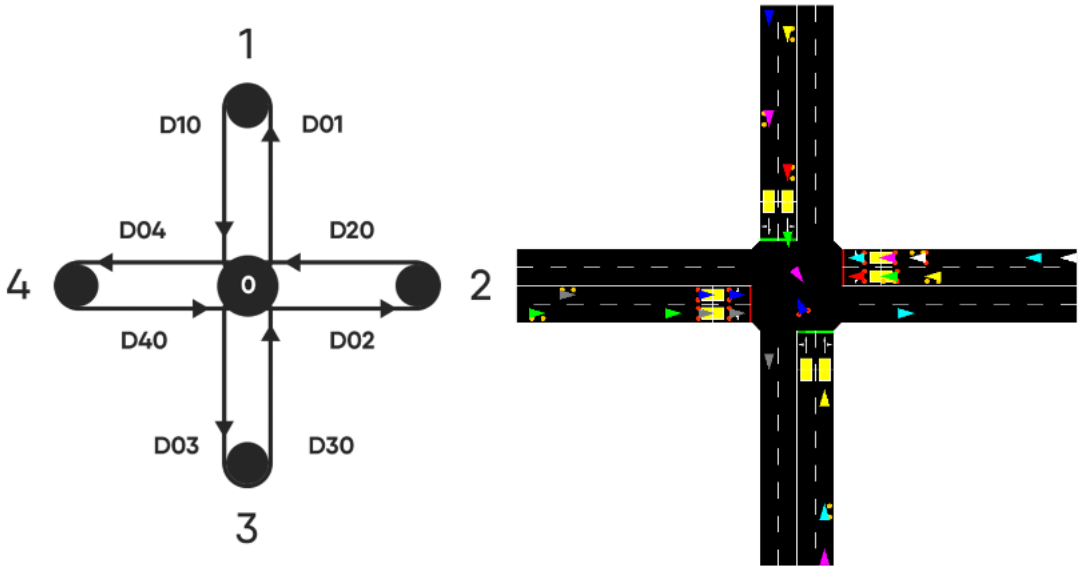


Figure 3: The test transport network

Thus, the state space $\mathcal{S}$ in this setting is *unbounded*:

$$\mathcal{S} \doteq \{(s_0, s_1) \mid s_0, s_1 \in \mathbb{Z}^+\} \tag{3.4}$$

Besides the number of halting vehicles, a variety of other factor influencing decision making might be included into the state vector, such as the current phases duration, time of the day, weather conditions, etc. This might be a major improvement to the accuracy of decision making and will be considered in the future work.

3.3 Action space

At the beginning of each cycle the agent first receives a state signal, then it performs an action, and, finally, it gets a reward for the action. The agent-environment interaction is an infinite horizon process (which does not naturally break into episodic tasks and continues without limits) [3]. The intersection has the green, red and amber phases. The signal plan of each cycle comprises the following components:

1. Green (NS) - Red (WE)
2. Amber
3. Green (WE) - Red (NS)
4. Amber

We modify only the green and red phases, while the amber remains constant. After each cycle the agent can choose one of the three actions:

1. $a_0 \doteq (0, 0)$ – do nothing
2. $a_1 \doteq (+dt, -dt)$ – extend the NS green phase and shorten the WE green phase
3. $a_2 \doteq (-dt, +dt)$ – shorten the NS greed phase and extend the WE green phase

These actions change the signal plan of the next cycle. Here dt is a constant extension parameter. Hence, the action space is *finite*:

$$\mathcal{A} \doteq \{a_0, a_1, a_2\} \tag{3.5}$$

In prospect, we are going to make the action space continuous: the action will be a precise value of the NS green phase duration, then, if the cycle length is fixed, the duration of the WE green phase can be immediately evaluated. Since

we are using the neural network for producing actions, and it is a continuous function approximator, this generalization will be natural.

3.4 Reward formula

Insofar as the task of the agent is to reduce the number of halting vehicles as much as possible, while keeping the balance between the conflicting directions, the reward formula ought to clearly reflect these goals. For this task we propose to use the following *continuous* reward formula. Upon transition between states

$$S_t = (NS, WE) \rightarrow S_{t+1} = (NS', WE'), \tag{3.6}$$

the reward signal is defined as follows:

$$\begin{aligned} R(S_t, S_{t+1}) \quad &\doteq \mu \overbrace{(|NS - WE| - |NS' - WE'|)}^{\text{equilibrium term}} + \\ &+(1-\mu) \underbrace{(NS + WE - (NS' + WE'))}_{\text{queue reduction term}}, \end{aligned} \tag{3.7}$$

where $\mu \in [0, 1]$ is the trade-off between the queue reduction and equilibrium terms. Throughout all the proposed algorithms we have used $\mu = 0.5$, i.e. the queue reduction and equilibrium goals have been equally important for learning.

The reward formula (3.7) captures the following intuition: if the difference between the conflicting directions has decreased, then the first term will be positive; if the total number of standing vehicles has decreased, then the second term will be positive, and the reward value will also be positive in proportion to the magnitude the mentioned values have decreased. In case of the increase in the values of both goals, the overall reward will be negative in the magnitude of the increases. Finally, when there is an increase in the value of one goal and decrease in the other, the two terms of the formula will eat each other and the resultant value will indicate whose impact has been stronger — this accounts for the presence of the plus sign in the formula.

3.5 Temporal-difference learning

There are two problems that are central to reinforcement learning. The *policy evaluation* or *prediction* is estimating the value functions v_π or q_π for a given policy π. Based on these new estimates, we can ameliorate the current policy to obtain a better one which gives higher returns at each state by making the policy greedy with respect to the current value function. This is called *policy improvement*. The overall problem of finding an optimal policy is called the *control* problem. Alternating the estimation of the value function and the improvement of the policy is called the *generalized policy iteration*. In the current work we will mostly concern ourselves with the control problem, since it is important for the agent to learn how to make good actions.

In reinforcement learning there are several fundamental approaches to solving the prediction and control problems. The *dynamic programming (DP)* method relies heavily on the knowledge of the model of the environment, i.e. the set of all state transition probabilities is assumed to be known. The dynamic programming uses the iterative method of successive approximations to solve the system of linear equations defined for the value function by the set of equations (2.10). To improve the policy, for a given state the DP methods use the model of the environment and make one step of lookahead into the next state to see which action would lead to the highest value in the next state and making the new policy to select this action henceforth. For more technical information about the policy evaluation and policy improvement in DP see [1, Sec 4.1] and [1, Sec 4.2].

Another method, called *Monte Carlo*, is used to make the agent learn entirely from the actual *experience* — sample sequences of states, actions, and rewards from direct interaction with the environment. That is, this sort of learning requires no prior knowledge of the environment's dynamics (model),

yet it can still attain the optimal behavior, and this is astonishing. Monte Carlo is usually considered for episodic tasks, where well-defined returns are available after each completed episode. To put it simply, *Monte Carlo* methods solve the prediction problem by averaging sample returns. An actual return value from every successive episode is a sample. Because we have no environmental model at hand, in order to make policy improvement we need to estimate the action-value function q_π instead of the state value function v_π. In policy improvement Monte Carlo methods just make the current policy greedy with respect to the newly estimated action-value function. For more details on the Monte Carlo methods see [1, Sec 5.1] and [1, Sec 5.3].

The method we are going to be using in defining our algorithm in the next section is called the *temporal-difference (TD) learning*. Essentially, this method is a combination of both the dynamic programming and Monte Carlo methods. Like DP, TD methods update estimates of the value functions obtained in turn from other learned estimates. However, in contrast to DP, TD assumes no predefined model of the environment whatsoever. Like Monte Carlo, TD methods can learn directly from raw experience in the *online* fashion, meaning that the TD approach does not wait for a final return of each episode to update the estimates (they *bootstrap*).

Let us denote by $V(S_t)$ the value of the current estimate of the state-value function for state $S_t \in \mathcal{S}$. Upon transition to a new state S_{t+1} and receiving R_{t+1}, the TD method makes the following update:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{3.8}$$

Another perspective on the above formula is:

$$V(S_t) \leftarrow V(S_t) + \alpha\delta_t \tag{3.9}$$

$$\delta_t \doteq \mathit{Target} - \mathit{Estimate} \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{3.10}$$

So, what the update does is it shifts the current estimate in the direction of the error between the target at time t and the current estimate of the value function. This error δ_t is called the *TD error*. Notice that the TD error at each time is the error in the estimate made at that time. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, δ_t is the error in $V(S_t)$ available at time $t+1$.

The update formula (3.8) follows the following general rule that occurs frequently throughout reinforcement learning:

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} + \textit{StepSize}\,[\textit{Target} - \textit{OldEstimate}] \tag{3.11}$$

This update rule can be treated as the averaging of the old estimate and the target with the constant parameter $\textit{StepSize} = \alpha$ that defines the extent to which the new target affects the average. With this interpretation in mind, the update rule for the Monte Carlo method appears as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha\,[G_t - V(S_t)]\,, \tag{3.12}$$

where G_t is the actual full return following the end of an episode at time t. Formula (3.12) clearly reflects how the Monte Carlo method works: it averages sample returns following the episodes starting from state S_t.

Recall that according to the definition of the state-value function:

$$\begin{aligned} v_\pi(s) \quad &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] && (3.13)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (3.14) \end{aligned}$$

Since the true target value v_π is not known, Monte Carlo methods use an estimate of (3.13) as a target, whereas DP methods use an estimate of (3.14) as a target. The Monte Carlo target is an estimate because the expected value in (3.13) is not known; a sample episodic return is used in place of the real expected

return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because the true value of $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (3.14) *and* it uses the current estimate V instead of the true v_π. Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP.

3.6 Shallow Q-Network (SQN)

In the current work we propose to use a reinforcement learning algorithm based on the temporal-difference learning to solve the problem of controlling the traffic signal. We name it the *Shallow Q-Network (SQN)*. The word "shallow" comes from the fact that in the algorithm the approximate action-value function is represented not as a table, but as a parametrized continuous function — the *neural network* with a shallow architecture. That is,

$$q_*(s, a) \approx \hat{q}(s, a, w), \tag{3.15}$$

where $\hat{q}(s, a, w)$ represents the value of the neuron corresponding to the action a in the output of the neural network evaluated at state s. The neural network is parametrized by the set of weight matrices $w \doteq \{w^{(1)}, w^{(2)}, \ldots, w^{(L-1)}\}$, where L is the total number of layers in the neural network. The matrix $w^{(k)}$ for $k = \overline{1, L-1}$ defines the transition from the k^{th} to the $(k+1)^{th}$ layer.

In order to be able to train the neural network, we need to define the cost function. The algorithm should adjust the weight matrices w so as to reduce the cost function as much as possible. The natural way to define a cost function which is consistent with our goal of approximating the optimal action-value function q_* is:

$$J(w) \doteq \frac{1}{2} \sum_{s \in \mathcal{S}, a \in \mathcal{A}} \left(q_*(s, a) - \hat{q}(s, a, w)\right)^2 \tag{3.16}$$

However, we don't know the true values of $q_*(s,a)$ in the sum above, so this cost function is inappropriate. To remedy this state of affairs, the temporal-difference learning comes in.

The SQN algorithm proceeds in the *online* manner: at each time step t it makes a TD estimate $U_t(S_t, A_t)$ to the target value $q_*(S_t, A_t)$ as follows. Given the current transition from MDP:

$$S_t \underset{A_t}{\longrightarrow} (S_{t+1}, R_{t+1}) \tag{3.17}$$

$$U_t(S_t, A_t) \doteq R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{t+1}, a', w_t) \tag{3.18}$$

Afterwards, we define the cost function as

$$J_t(w_t) \doteq \frac{1}{2} \left(U_t(S_t, A_t) - \hat{q}(S_t, A_t, w_t) \right)^2, \tag{3.19}$$

which involves the difference between the new estimate of the target and the current estimate of the action-value function produced by the neural network with the current weights.

Gradient descent is used to update the weights of the neural network by a small amount along the anti gradient of the cost function (3.19). Note that the cost function (3.19) does not involve summation over all the states and actions, but rather it uses only state S_t and action A_t from the current time step. These were provided stochastically by the environment. That is the reason why this is the *stochastic gradient descent*: it uses the cost function involving a training example which is chosen stochastically, and seeks to reduce error just on it.

Let us compute the gradient of the cost function (3.19):

$$\nabla J_t(w_t) = \left(U_t(S_t, A_t) - \hat{q}(S_t, A_t, w_t) \right) \nabla_{w_t} \hat{q}(S_t, A_t, w_t) \tag{3.20}$$

Then the SQN algorithm updates the weights w_t of the neural network as follows:

$$w_{t+1} \doteq w_t - \alpha \nabla J_t(w_t), \tag{3.21}$$

where α is the learning rate.

Note also that (3.20) is not the true gradient of (3.19), since, while computing it, we did not take into consideration the fact that the value $U_t(S_t, A_t)$ is dependent on the network weights w_t. After $U_t(S_t, A_t)$ is evaluated, it is treated to be a constant. That is why (3.21) is called the *semi-gradient descent*.

In general, the gradient $\nabla_{w_t}\hat{q}(S_t, A_t, w_t)$, on the right of (3.20), is not easy to compute for the neural networks having one or more hidden layers, and, instead, the whole $\nabla J_t(w_t)$ is computed by means of the *backpropagation algorithm*. The error δ_t is backpropagated at each time step t which is defined as follows:

$$\delta_t \doteq U_t(S_t, A_t) - \hat{q}(S_t, A_t, w_t) \tag{3.22}$$

For more detailed information on neural networks and how the backpropagation algorithm works see [2].

3.7 Average Reward Network (ARN)

In this section we propose another approach of training the neural network by learning a numerical *preference* for each action a to be selected in state s, which we denote $H(a, s, w_t)$, where w_t is again a set of weight matrices of the neural network. You can think of $H(\cdot, s, w_t)$ as a vector of size $k \doteq |\mathcal{A}|$ produced at the output layer of the neural network when the activation function has not yet been applied to it. The larger the preference of an action, the more favorable it is to be taken; however, the preference has no interpretation in terms of reward. Only the relative preference of one action over another is important; if we add 1000 to all the preferences, there is no effect on the action probabilities, which are determined according to the *softmax* (or Boltzmann) *distribution* as follows:

$$\pi(a, s, w_t) \doteq \frac{e^{H(a,s,w_t)}}{\sum_{b=1}^{k} e^{H(b,s,w_t)}}, \tag{3.23}$$

The parametrized probability distribution $\pi(\cdot, s, w_t)$ constitutes the current *stochastic* action selection policy.

Next, we need to maintain the average reward $R_t^*(s)$ at time t for each state $s \in \mathcal{S}$ updated by the following recurrent rule:

$$R_t^*(s) \doteq R_{t-1}^*(s) + \alpha \left(R_t(s) - R_{t-1}^*(s)\right) \tag{3.24}$$

$$R_0^*(s) \doteq 0 \tag{3.25}$$

Note that the above update follows the general rule (3.11), and here we assume that the state space $\mathcal{S}$ is *finite*.

Upon selecting action A_t in state S_t and receiving reward R_t, the average reward $R_t^*(S_t)$ is updated, and the error $\delta_t \in \mathbb{R}^k$ is computed according to (3.26). The term $R_t^*(S_t)$ serves as a baseline with which the current immediate reward is compared. If the reward is higher than the baseline, then the probability of taking A_t in the future is increased; and if the reward is below baseline, then the probability is decreased. The non-selected actions move in the opposite direction.

$$\begin{aligned} \delta_t(A_t) &\doteq -\left(R_t - R_t^*(S_t)\right)\left(1 - \pi(A_t, S_t, w_t)\right), \\ \delta_t(a) &\doteq \left(R_t - R_t^*(S_t)\right)\pi(a, S_t, w_t) \text{ for all } a \neq A_t \end{aligned} \tag{3.26}$$

Afterwards, the error δ_t is backpropagated through the network using the back-propagation algorithm, and the weight matrices w_t are updated. We name the algorithm described in this section as the *Average Reward Network (ARN)*.

3.8 Direct method of training on history

In this section we propose a learning algorithm based solely on immediate rewards. In order to describe it, we need to redefine the notion of action-value function:

$$q_*(s, a) \doteq \mathbb{E}\left[R_t \mid S_t = s, A_t = a\right] \tag{3.27}$$

The value (3.27) is nothing but the expected immediate reward which follows from taking action A_t in state S_t. That is, for a fixed state the true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:

$$Q_t(s,a) \doteq \frac{\text{sum of rewards when } a \text{ taken in } s \text{ prior to } t}{\text{number of times } a \text{ taken in } s \text{ prior to } t}, \tag{3.28}$$

where $Q_t(s,a)$ denotes an estimate of the true action-value $q_*(s,a)$ at time t.

In order to maintain the value $Q_t(s,a)$ for each state-action pair conveniently in a computer memory, the update of $Q_t(s,a)$ is carried out according to the following recurrent rule. Let us fix a state-action pair (s,a) and write simply Q_t instead of $Q_t(s,a)$. Then:

$$\begin{aligned} Q_{t+1} &\doteq \frac{1}{t}\sum_{i=1}^{t} R_i = \\ &= \frac{1}{t}\left(R_t + \sum_{i=1}^{t-1} R_i\right) = \\ &= \frac{1}{t}\left(R_t + (t-1)\frac{1}{t-1}\sum_{i=1}^{t-1} R_i\right) = \\ &= \frac{1}{t}\left(R_t + (t-1)Q_t\right) = \\ &= \frac{1}{t}\left(R_t + tQ_t - Q_t\right) = \\ &= Q_t + \frac{1}{t}\left(R_t - Q_t\right) \end{aligned} \tag{3.29}$$

The obtained formula (3.29) is much reminiscent of the general update rule (3.11), and this justifies our interpretation of (3.11) as averaging. The only difference is that the learning rate α is replaced with the factor $1/t$ — in this case (3.11) turns exactly into the arithmetic mean.

The simplest action selection rule is to select one of the actions with the highest estimated value, that is, the action selection policy is *greedy*. If there is a tie between a number of actions having the same estimated value, then the

tie is broken in some arbitrary way, perhaps randomly. We write this *greedy* action selection rule as follows:

$$A_t \doteq \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q_t(S_t, a) \tag{3.30}$$

Greedy action selection always exploits current knowledge to maximize immediate reward [1, Sec 2.2]; it spends no time at all sampling apparently inferior actions to see if they might really be better — we come across the exploration-exploitation dilemma. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ε, instead select randomly from among all the actions with equal probability, independently of the action-value estimates. We call methods using this near-greedy action selection rule *ε-greedy* methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Q_t(s, a)$ converge to $q_*(s, a)$.

Another way to ensure enough exploration for addressing the exploration-exploitation issue, and which has been actually used in our implementation of the proposed algorithm, is to run a simulation with a completely random action selection policy for a large number of cycles N. This will produce the following *history dataset*

$$\mathcal{H} \doteq \{< s, a, s', r >\}_{i=1}^{N}, \tag{3.31}$$

each record of which represents a transition between the successive states s and s' with the action taken a and the observed reward r. If the number of cycles N is large enough, then by using the history we can get reliable estimates of the action values. Therefore, this history will be a supervised exploratory training dataset for the algorithm which will follow completely greedy policy afterwards.

In the current algorithm a neural network is used to produce the action A_t to be taken when given a state vector S_t. The *sigmoid* activation function is

used on the output layer of the network. We define the error δ_t as follows:

$$\delta_t \doteq A_t - \operatorname*{argmax}_{a \in \mathcal{A}} Q_t(S_t, a), \tag{3.32}$$

where $\mathcal{A} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$ is the space of all possible actions.

Eventually, by backpropagating the error δ_t we update the weights of the neural network by the gradient descent algorithm.

4 EXPERIMENTS, IMPLEMENTATION NOTES AND IMPROVEMENT PROSPECTS

In this chapter we will provide the results of the experiments for all the proposed algorithms: SQN, ARN, and the direct method. We will also discuss some subtleties of the implementation for each of them, as well as give prospects for further improvement.

4.1 Shallow Q-Network (SQN). Implementation

SQN is a model-free, online, off-policy method. *Off-policy* property refers to the fact that the current action selection policy is not involved in the learning formula (3.18). The distinctive features of the SQN algorithm over other analogous algorithms are:

- *Shallow* architecture of the neural network
- *Unbounded* state space $\mathcal{S}$: $S_t = (NS, WE),\ NS \in \mathbb{Z}^+,\ WE \in \mathbb{Z}^+$
- *Exclusive* continuous reward formula

All the algorithms have been implemented in Python programming language, and tested in one of the open source microscopic traffic simulation software [4], called *SUMO*. The neural network, including feedforward and backpropagation steps, has been implemented without any library whatsoever (except for the *NumPy* library intended for optimized matrix operations). The neural network was of 2-25-3 architecture, that is, 2 neurons in the input layer, 25 neurons in the hidden layer, and 3 neurons in the output layer. *Rectified linear unit (ReLU)* activation function $f(x) \doteq \max\{0, x\}$ was used in the hidden layer, and no activation in the output layer. The action selection policy was ε-greedy with $\varepsilon = 0.1$. The discounting factor was $\gamma = 0.6$. Gradient descent was

used with the learning rate $\alpha = 0.001$, 5 iterations with the regularization parameter $\lambda = 0.03$ (for information about regularization see [2]). Feature scaling and mean normalization [2] have been done to the input state vector to ensure the stability of learning, that is, each component of the input vector was put in the range $[-1, 1]$. There have been 1000 cycles of SUMO simulation, in which vehicles are spawned in both directions with uniformly random distributions. The result of the simulation is presented in Figures 4-7 below.

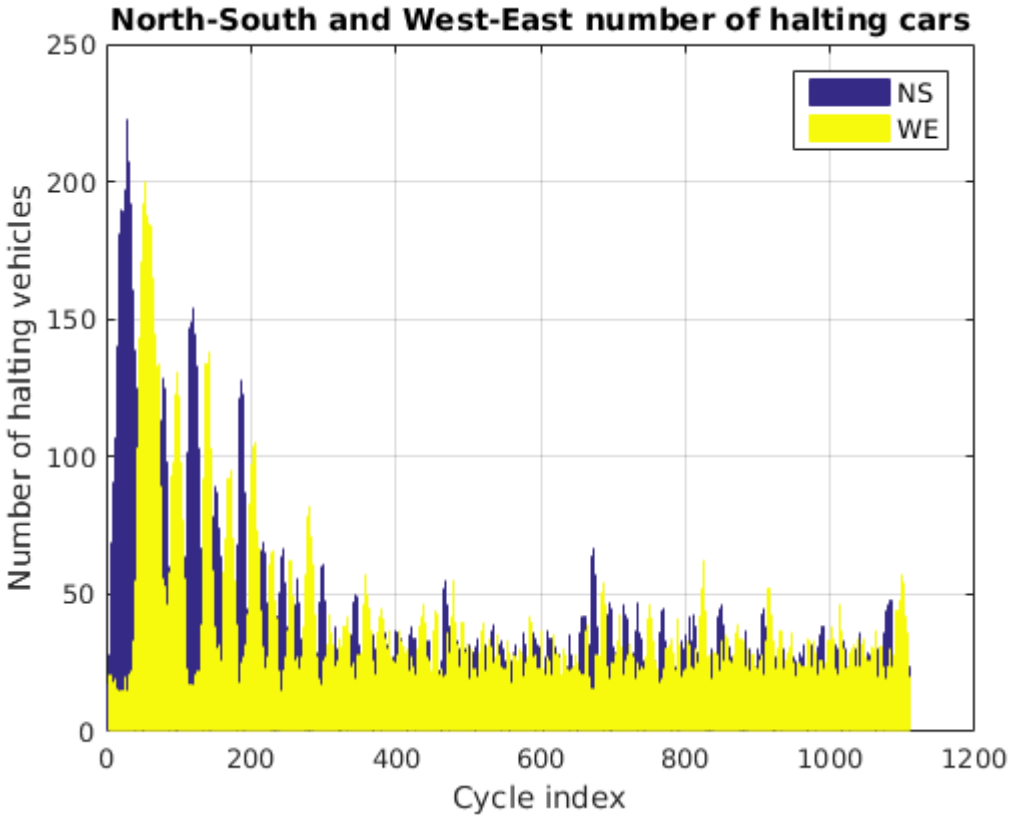


Figure 4: SQN. Number of halting vehicles per cycle

As you can see on Figure 4, the SQN algorithm successfully converges to an optimal solution, and the number of halting cars is kept low in both directions starting approximately from the 300^{th} cycle.

On Figure 5 you can see the evident convergence of the reward signal, which

is kept approximately in the range $[-20, 20]$ starting from the 300^{th} cycle. This is also indicative of convergence, since the better the agent is performing, the lesser the environment should reward or penalize it.

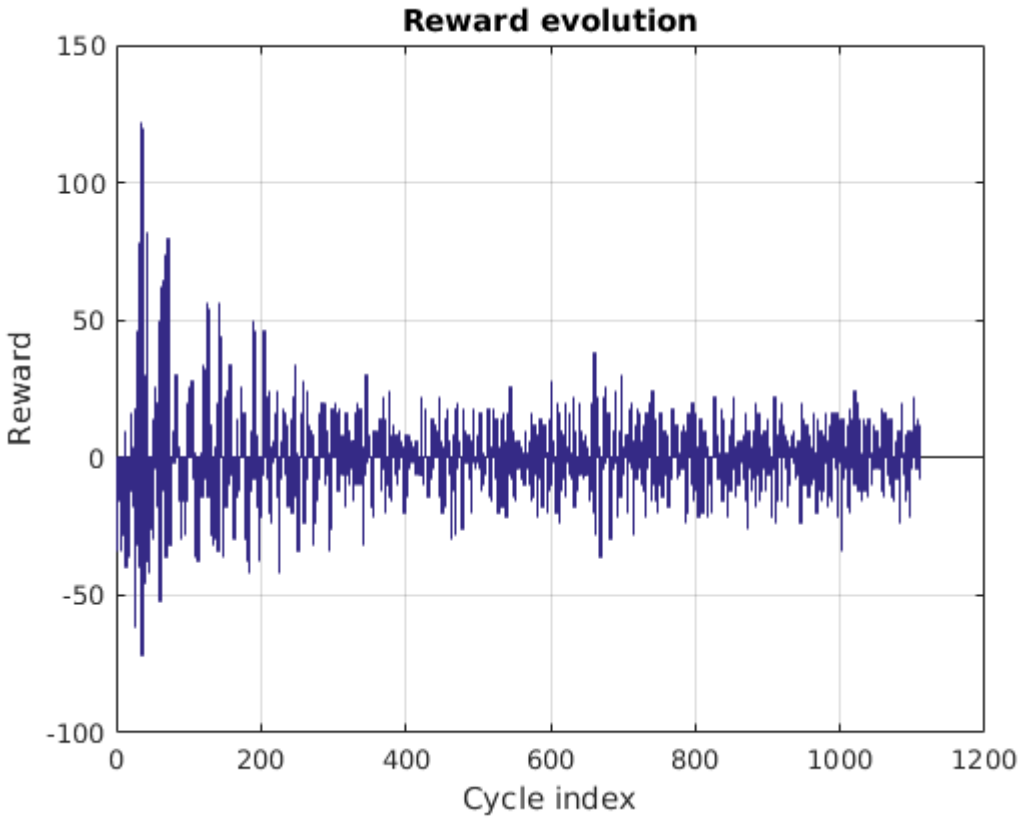


Figure 5: SQN. Reward per cycle

As we said earlier, the state space $\mathcal{S}$ is unbounded. However, in order to better visualize the state of the environment at each time step, for each of the directions — NS and WE — we discretize the number of cars as shown in the Table 1. As a result, the discretized state space consists of total $5 \times 5 = 25$ states.

On the top of Figure 6 the state statistics is shown, i.e. the number of times each state has occurred during the simulation. On the bottom of Figure 6 for each cycle the corresponding state of the environment at that time is shown.

As you can see, the states number 2, 6, and 7 have been dominating during the simulation, and they correspond to the states ELL, LEL, and LL respectively. This clearly suggests that throughout the simulation the number of vehicles standing before the intersection was kept low in both directions, and, therefore, the algorithm works well on the goal.

States	Shorthand	Description
Extra low	EL	$0 \leq n \leq 16$
Low	L	$17 \leq n \leq 32$
Medium	M	$33 \leq n \leq 48$
High	H	$49 \leq n \leq 64$
Extra high	EH	$65 \leq n \leq 240$

Table 1: Discretization of the state space

There are a number of ways to improve the proposed algorithm:

1. Extend the state space to include the phases duration, time, etc.
2. Add the polynomial features $s = [1, s_0, s_1, s_0^2, s_0 s_1, s_1^2, \ldots]$ into the input vector to account for a possible interplay between quantities
3. Double Q-learning to prevent positive bias [5]
4. Deep CNN architecture [5]
5. Train the NN on mini-batches from the history rather than immediate responses [5]
6. Adjust the parameters
7. Use SARSA estimate of the target [1, Sec 6.4]

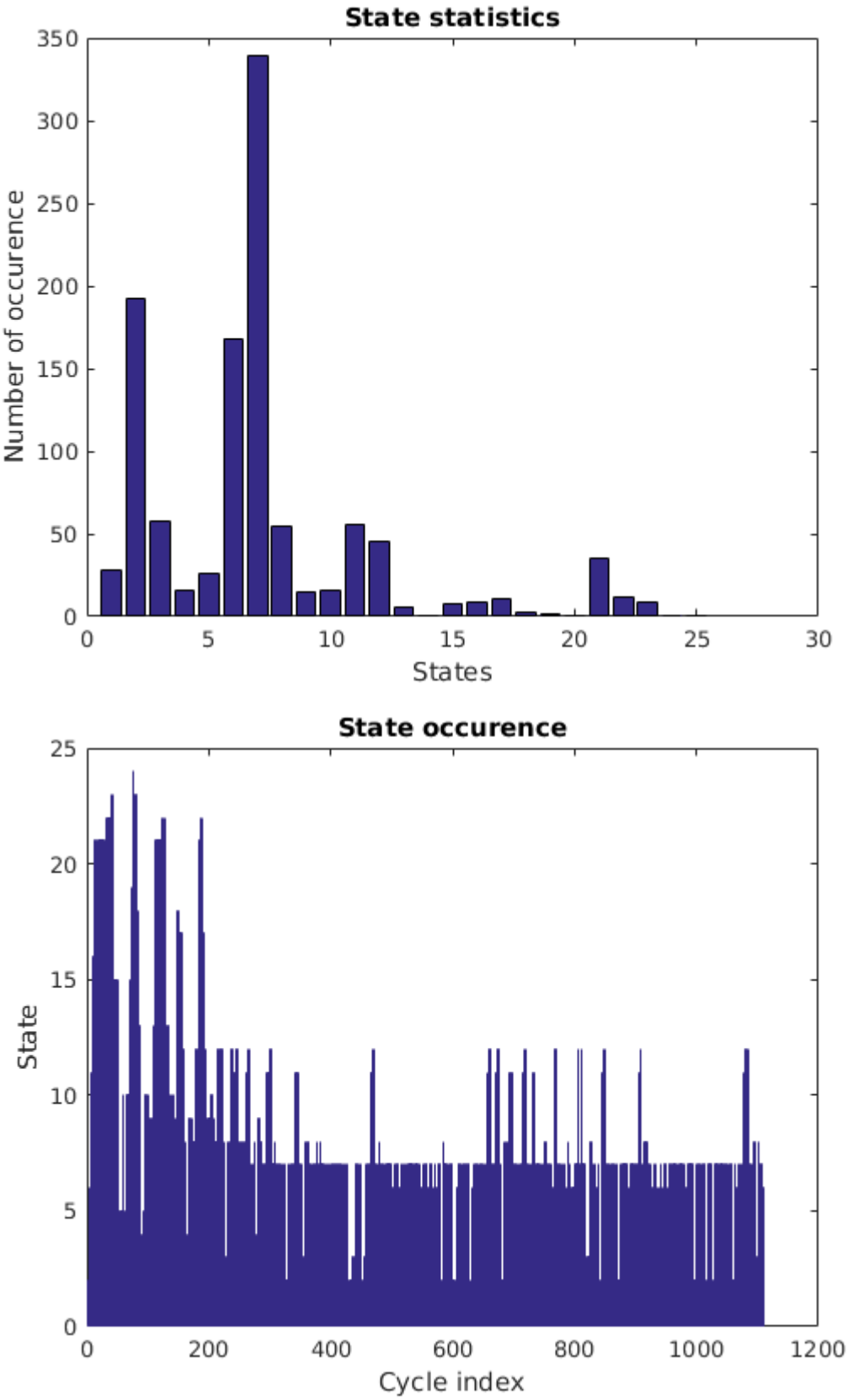


Figure 6: SQN. State statistics and state occurrence

4.2 Average Reward Network (ARN). Implementation

Average Reward Network (ARN) algorithm is a model-free, online, on-policy method. The term *on-policy* refers to the fact that the current action selection policy is involved in the learning formula (3.26). The implementation of the ARN algorithm in most respects coincides with that of the SQN, except that the discounting factor γ is absent, and there is no need to adjust it.

However, one distinguishing feature of the ARN implementation was that the neural network was penalized with a fixed negative reward (-10) each time the agent attempted to take an unfeasible action. This gimmick was helpful in speeding up learning, since with it the agent gets to know faster what are bad actions to take.

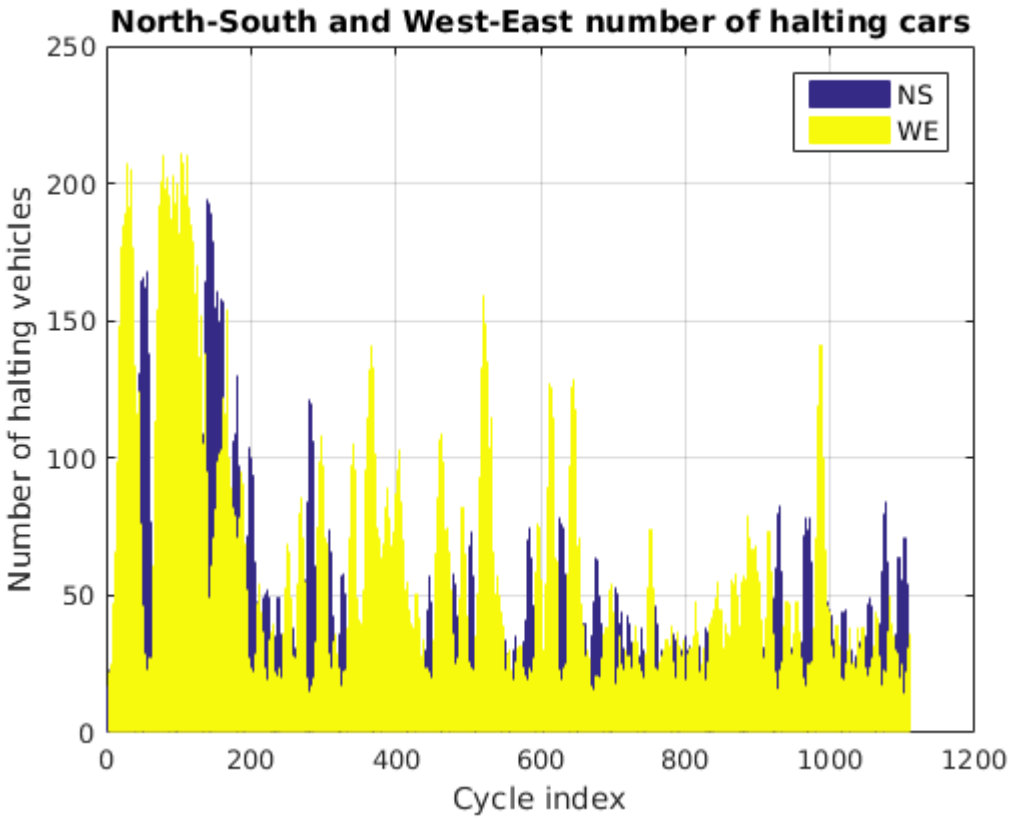


Figure 7: ARN. Number of halting vehicles per cycle

As it is seen from Figure 7, during the SUMO simulation with 1000 cycles, the agent was performing fairly well, though occasionally there were some bumps and spikes even after the algorithm has apparently shown convergence after approximately the 200^{th} cycle. This might be due to the fact that, unlike SQN, the ARN algorithm by construction considers only immediate rewards, and seeking to maximize them, can get into bad states after the one that follows immediately.

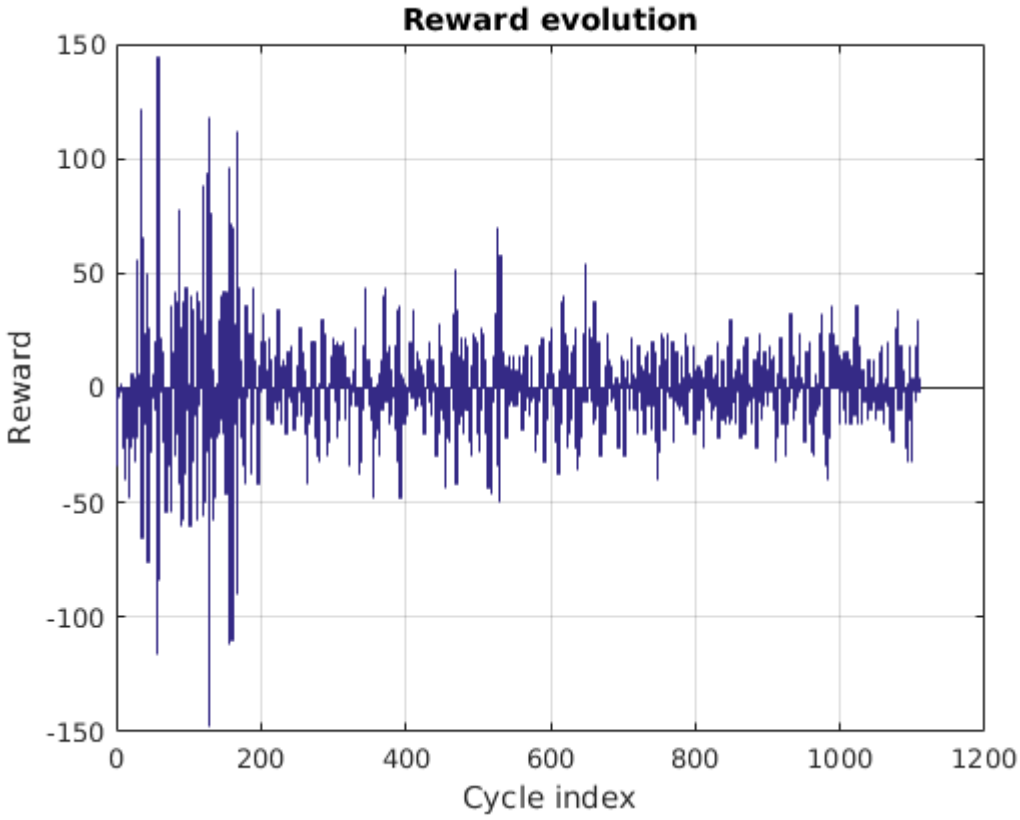


Figure 8: ARN. Reward evolution

Obviously, Figure 8 shows the convergence of the reward signal after the 200^{th} cycle. So, in part because it is easier for the ARN to learn only by immediate rewards, we can conclude that the ARN converges faster that the SQN, albeit afterwards it took actions that led to congestion.

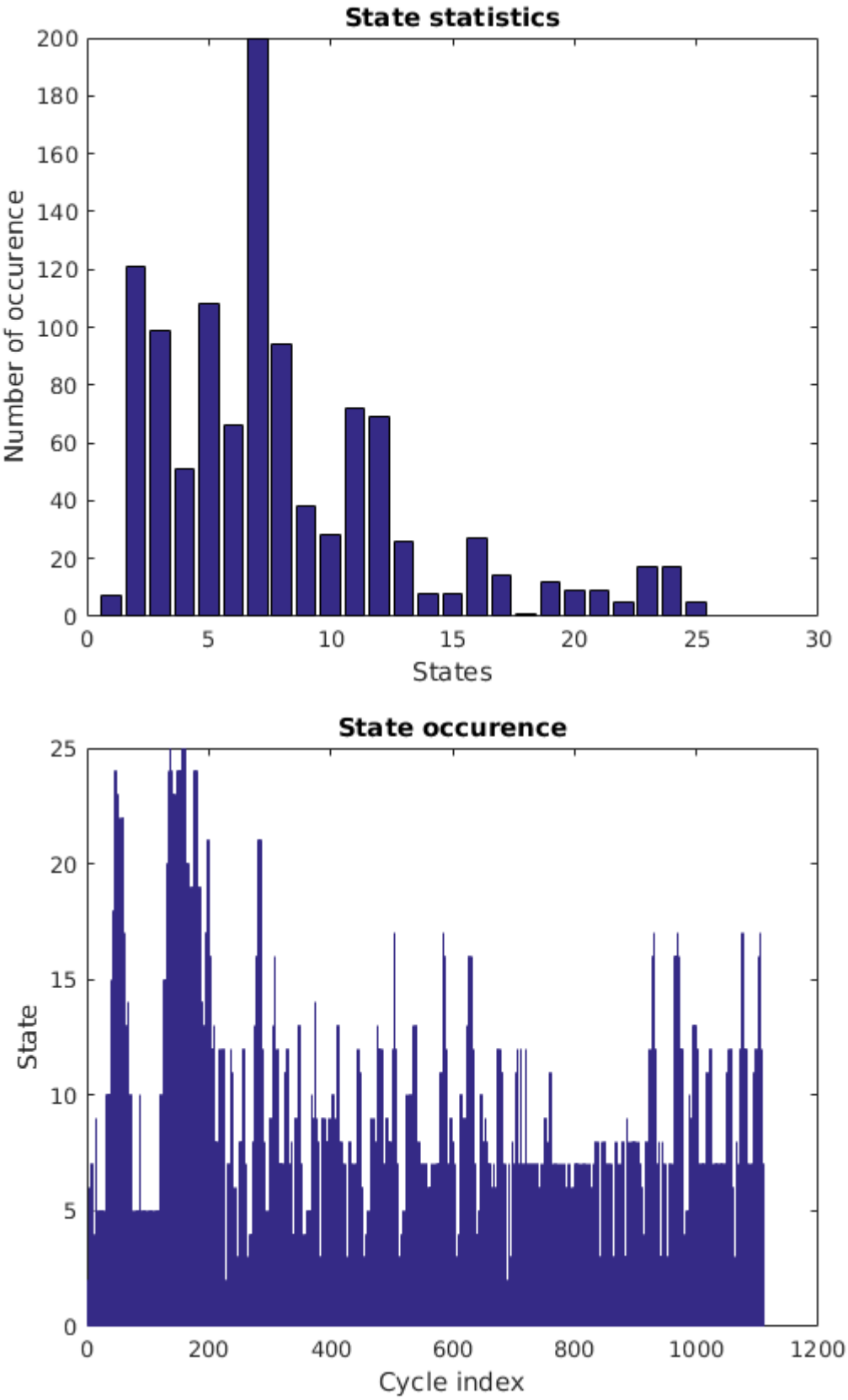


Figure 9: ARN. State statistics and state occurrence

The poorer performance of the ARN in comparison with SQN is reasonable, for by neglecting future possibilities, the ARN algorithm can take actions that are beneficial immediately, but are likely to fail over the long run. Another major difference between the two algorithms is that the ARN should maintain in memory a finite array to hold the action-value estimates, and we used the states from the discretized state space, as shown in Table 1, to represent values in the array. As you can notice from Figure 9, the states with low number of vehicles have been dominating. However, the state number 5, which is ELH, got a fairly large number of occurrences.

4.3 Direct method of training on history. Implementation

The direct method is model-free, online, and off-policy. In our implementation, for the sake of speeding up convergence, we collected the history dataset, running the simulation with uniformly random action selection policy for several thousand cycles. An alternate fully online way to run the algorithm without history is to use the ε-greedy policy. The history dataset provided the algorithm with reliable action value estimates. Subsequently, the algorithm proceeds by collecting batches consisting of 50 successive cycles, and trains the neural network on them. The neural network has the same topology as earlier, except that the *sigmoid* activation function $f(x) = 1/(1+e^{-x})$ was used in the hidden and output layers. The gradient descent was used to update the weights with the learning rate $\alpha = 0.003$ and 30 iterations on the current training batch.

On the top of Figure 10 it is seen that the number of cars rapidly decreases after the 400^{th} cycle and it kept low from that time on. On the bottom of Figure 10, you can observe a good convergence of the reward signal. On Figure 11 you can see the state statistics in which another discretization of the number of cars was used (see Table 2).

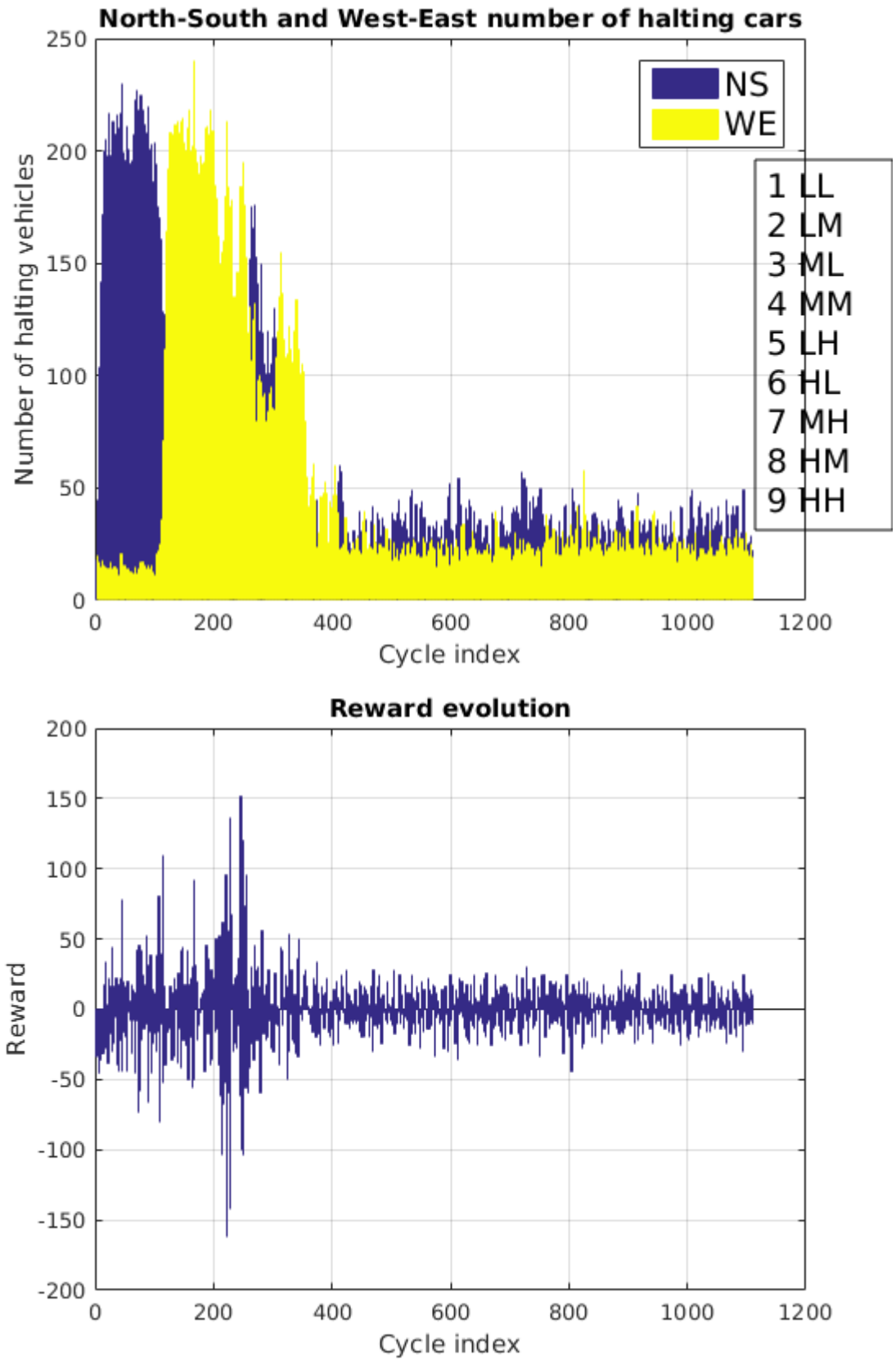


Figure 10: Direct method. Number of halting vehicles per cycle and the reward signal

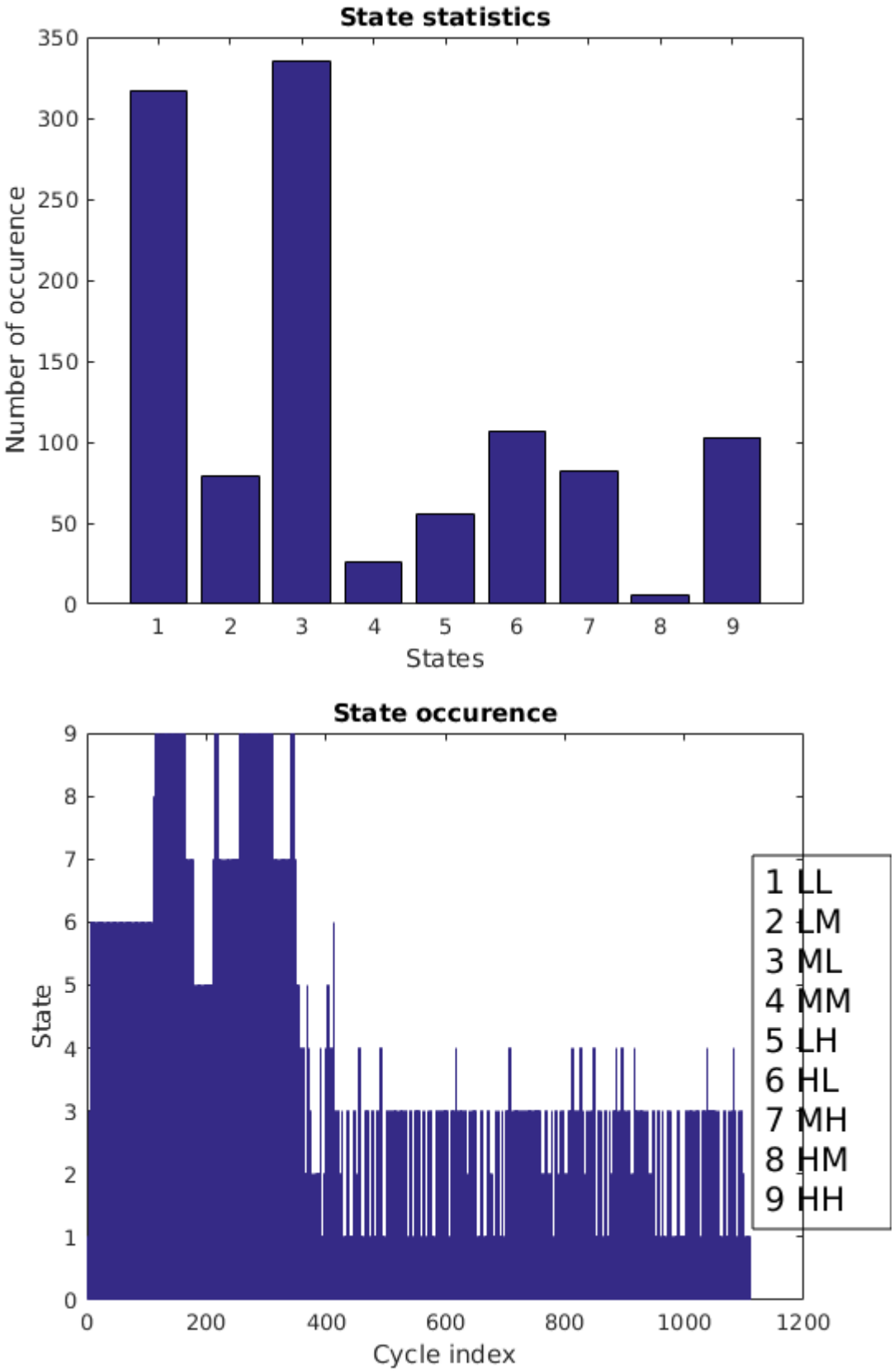


Figure 11: Direct method. State statistics and state occurrence

States	Shorthand	Description
Low	L	$0 \leq n \leq 29$
Medium	M	$30 \leq n \leq 59$
High	H	$60 \leq n \leq 240$

Table 2: Discretization of the state space used in the direct method

There is a possibility to improve the algorithm if we choose to not use a preliminary collection of history. We could use the Upper-Confidence-Bound technique [1, Sec 2.7] instead of the ε-greedy action selection policy to ensure enough exploration.

5 CONSTRUCTION OF A NEW CLUSTERING ALGORITHM

The discretization of the state space (see Table 1 and Table 2) used in the proposed algorithms may be rough: the boundaries set too strictly delimit the states, and the states near the boundaries, that are in effect similar, are placed into different elements in the discretized space, within which they are considered unrelated to one another. This is a major drawback, since when training a neural network, which is a continious function, it will not be able to perceive proximity between all the states properly.

One way to handle this issue is to apply a *clustering algorithm* (for information about clustering and unsupervised machine learning see Sec. 1.3) to the history of occurred states in order to let the computer program itself find the most appropriate discretized space on a case-by-case basis, namely, by designating the detected clusters as the new elements of the discretized space. This is a motivation for developing a new clustering algorithm which would possess necessary properties to be used in the context of our problem. In this chapter we will give the idea of a new clustering algorithm, as well as some intermediate experimental results, although currently the idea is incomplete and we are planning to develop it further in the future work.

5.1 Problem statement

Let a 2D dataset of N points $D \doteq \{(x_i, y_i)\}_{i=1}^{N}$ be given. The objective is to identify groups of points that are in some sense similar to each other. Also, a new clustering algorithm should:

1. be fast enough to be able to work in the online way
2. address the problem of clusters of various densities

3. be able to identify clusters of complex shapes

We are going to use *density* of points as the measure for cluster identification.

5.2 Local density function

Let us introduce the *local density function* $f : \mathbb{R}^2 \to \mathbb{R}$ as follows:

$$f(x, y) \doteq \sum_{(x_i, y_i) \in D} e^{-\lambda((x-x_i)^2+(y-y_i)^2)} \tag{5.1}$$

Intuitively, the value of this function at a particular point gives a local density of points disposed in a neighborhood of the point. The parameter λ controls the radius of this neighborhood.

Our conjecture is that by shifting the points along the gradient of the mean curvature we will obtain a *skeleton* of each cluster; and in order to classify a new point, we will simply determine to which cluster skeleton the point is nearest.

In order to obtain the formula for the mean curvature function H, firstly we need to find the first and second fundamental forms for the surface $(x, y, f(x, y))$ determined by the local density function.

$$I = \begin{pmatrix} E & F \\ F & G \end{pmatrix} = \begin{pmatrix} 1 + f_x^2 & f_x f_y \\ f_x f_y & 1 + f_y^2 \end{pmatrix} \tag{5.2}$$

$$II = \begin{pmatrix} L & M \\ M & N \end{pmatrix} = \begin{pmatrix} -f_{xx} & -f_{xy} \\ -f_{xy} & -f_{yy} \end{pmatrix} \tag{5.3}$$

All the partial derivatives of the function f involved in the expressions above can be easily calculated by hand.

Finally, the mean curvature is calculated as follows:

$$H = \frac{EN + GL - 2FM}{2(EG - F^2)} \tag{5.4}$$

The gradient ∇H of the mean curvature function is calculated numerically.

5.3 Experiments

The proposed idea was tested on an experimental dataset (see Figure 12 in appendix) in the MATLAB environment. On Figures 13, 14 the corresponding local density surface and mean curvature surface are shown. On Figures 15-19 you can see 10 iterations of the algorithm, and on each iteration the algorithm shifted all the points along the mean curvature gradient vectors by a small amount which is also a parameter.

On the last iteration (Figure 19) we can clearly distinguish various skeletons of the clusters. But yet we have not devised the way to distinguish among clusters.

CONCLUSION

In this work we have proposed, described, and tested 3 different algorithms of training the neural network of a traffic signal controller. The SQN algorithm was able to adapt to the environment and converge to an optimal action selection rule. The performance of the ARN algorithm was satisfactory in the sense that a state with a low number of vehicles had been dominating over the simulation; however, occasionally the algorithm allowed the number of vehicles to rise by a lot. The direct method has shown convergence, nevertheless we have not yet tested it using truly online learning.

We can conclude that the SQN algorithm is the most promising one: it has shown a good performance without assuming that the state space is discretized. This gives the opportunity to try to extend the state space so as it includes a variety of different factors influencing the road traffic. Also, in the future work we are planning to make the action space continuous as well: the neural network will directly produce a precise value of the phase duration.

In the future work we are going to test the proposed skeletonization algorithm on another datasets, as well as devise a method of distinguishing amongst clusters. We are planning to apply this clustering algorithm to the history of occurred states to arrive at an optimal discretization. Next, we will try a *model-based* approach, that is, we will collect a statistics with respect to the optimal discretization of the state space and estimate the transition probabilities. We will use the obtained model to apply the methods of dynamic programming.

6 REFERENCES

[1] Richard S. Sutton and Andrew G. Barto, 2017. *Reinforcement Learning: An Introduction*. Second edition, complete draft. The MIT Press.

[2] Andrew Ng, 2011. *Machine learning*. Course at coursera.org, Stanford University.

[3] D. Kurmankhojayev, N. Suleymenov, and G. Tolebi, 2017. *Online model-free adaptive traffic signal controller for an isolated intersection*.

[4] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. *Recent development and applications of SUMO - Simulation of Urban MObility*. International Journal On Advances in Systems and Measurements, 5(3&4):128138, December 2012.

[5] Xiaoyuan Liang, Xusheng Du, Guiling Wang, and Zhu Han. *Deep Reinforcement Learning for Traffic Light Control in Vehicular Networks*, 2018. arXiv:1803.11115

7 APPENDIX

The results of the experiments with the clustering algorithm in MATLAB

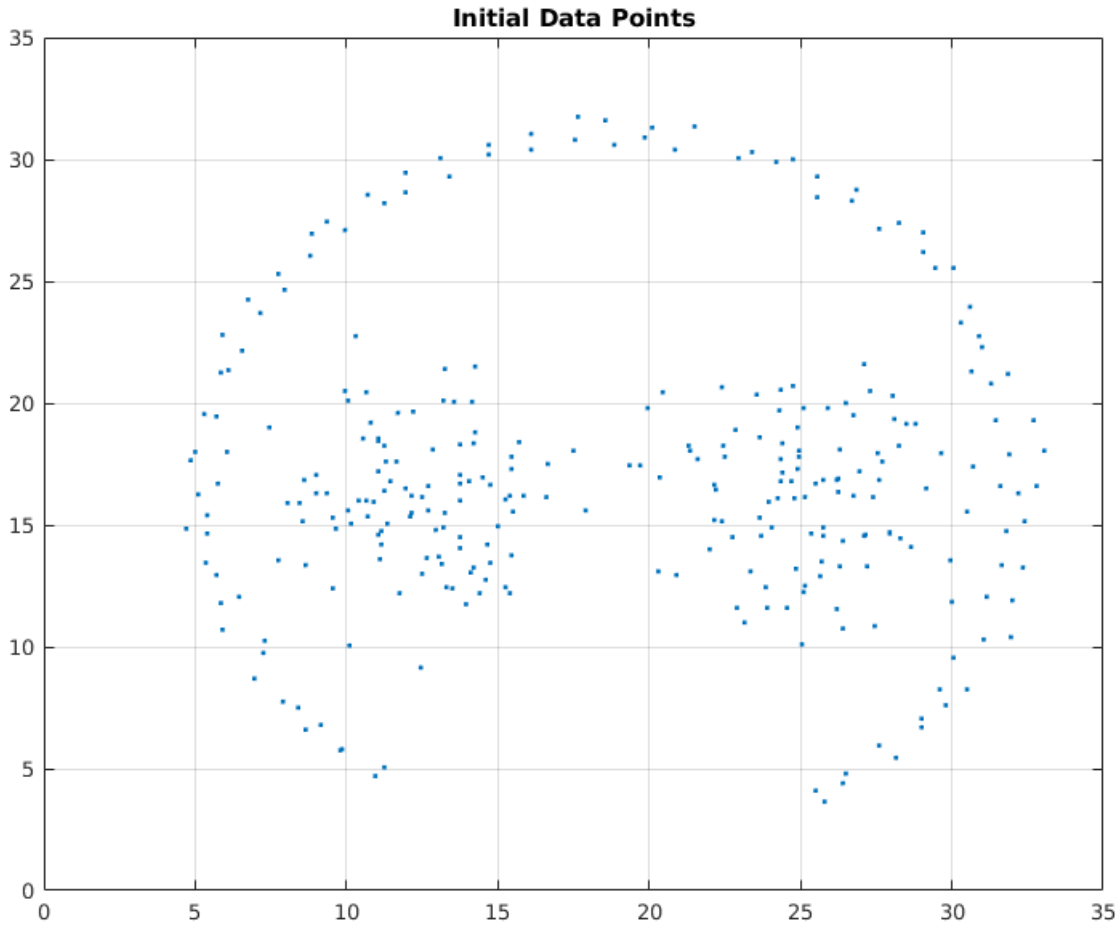


Figure 12: The experimental dataset

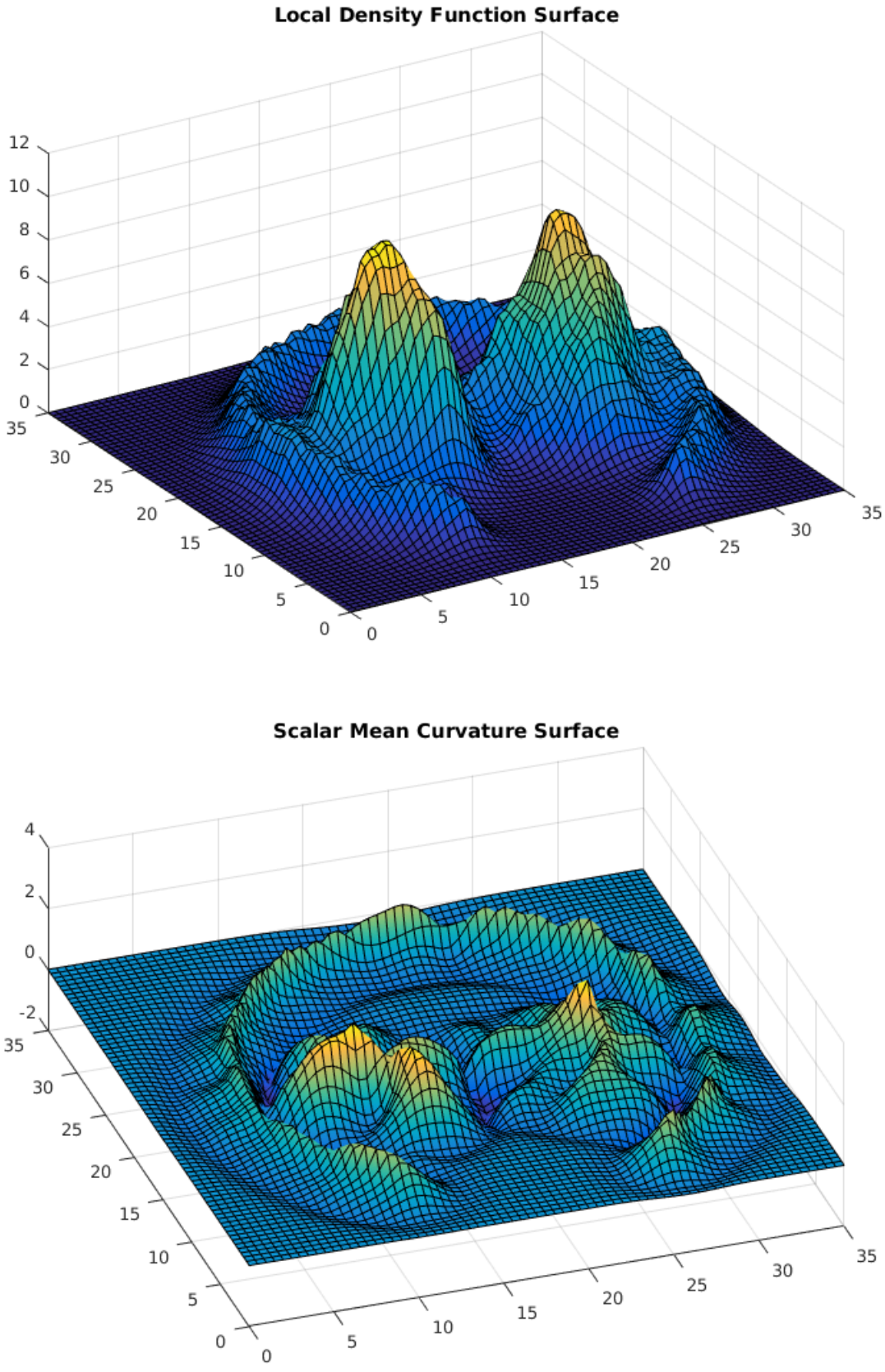
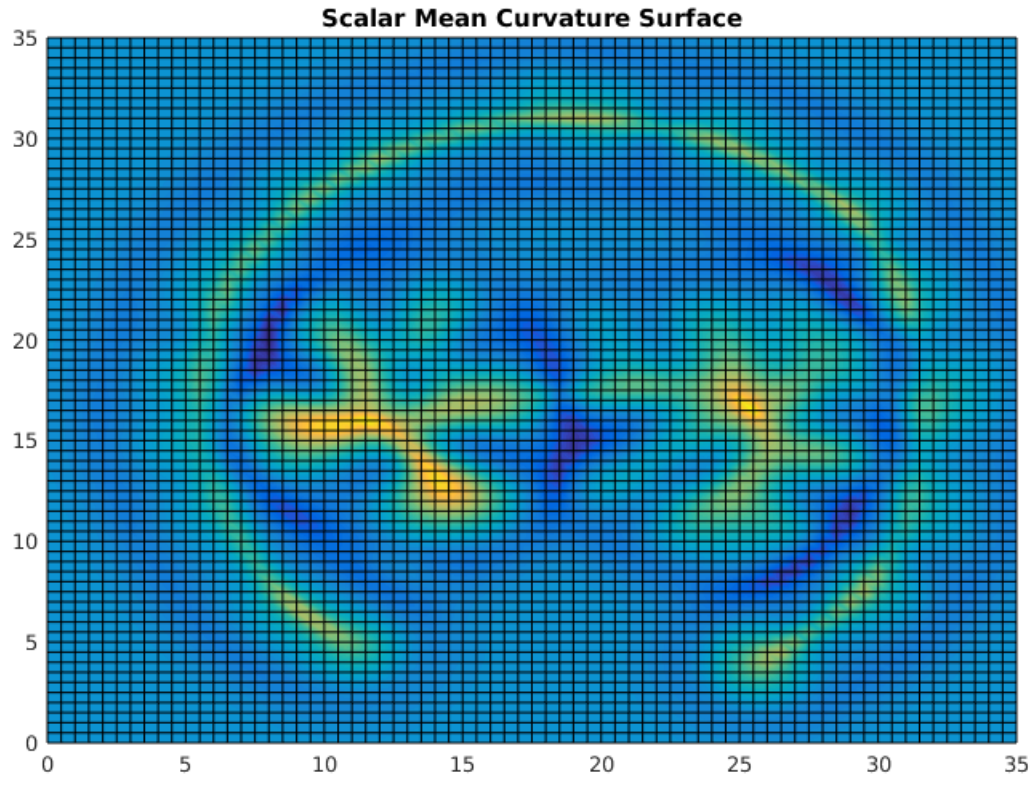


Figure 13: The local density surface, and the mean curvature surface



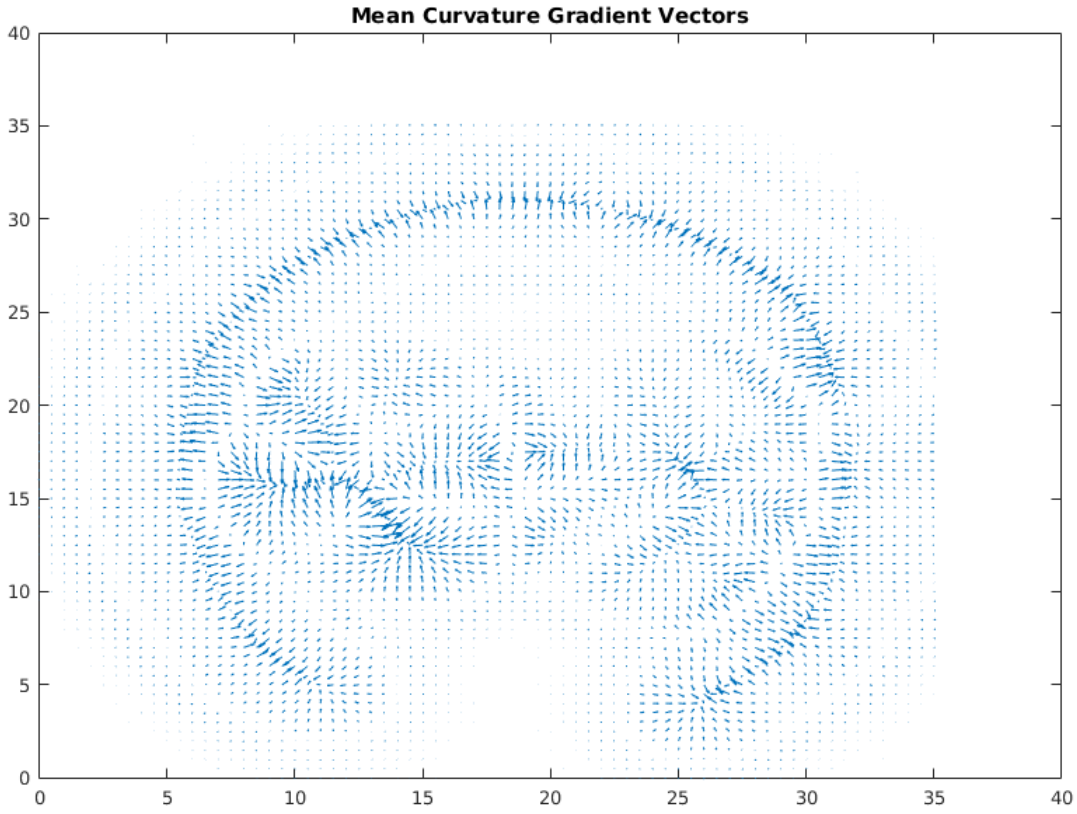
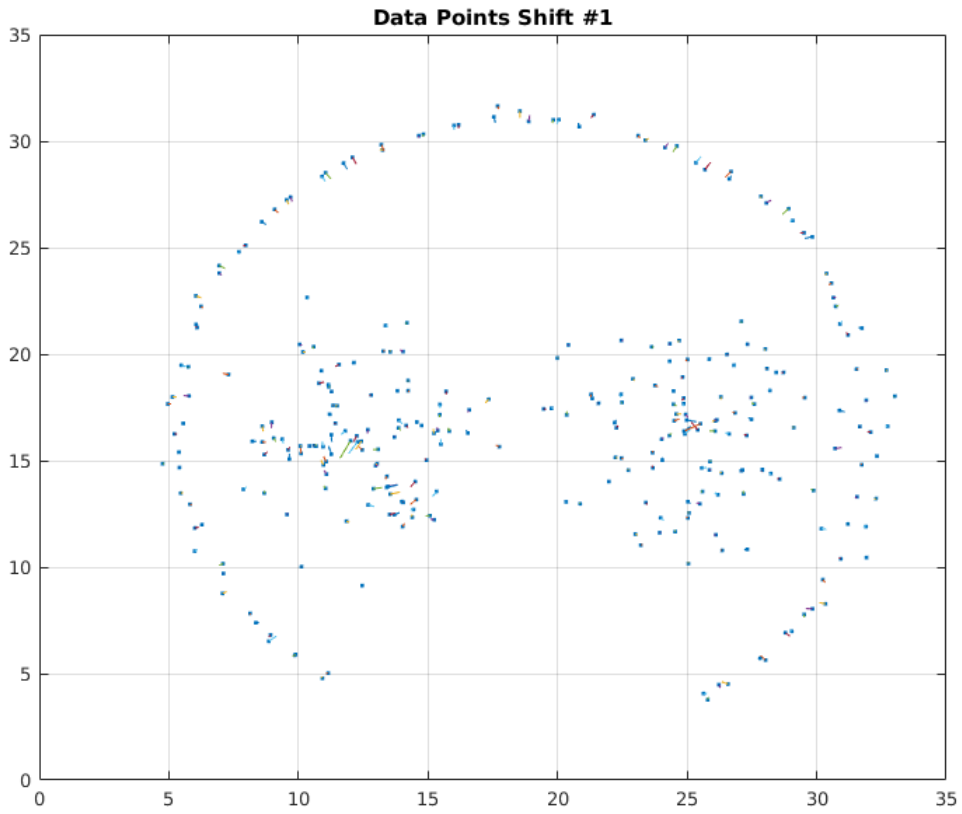


Figure 14: The mean curvature surface (view from the top), and the mean curvature gradient vector field



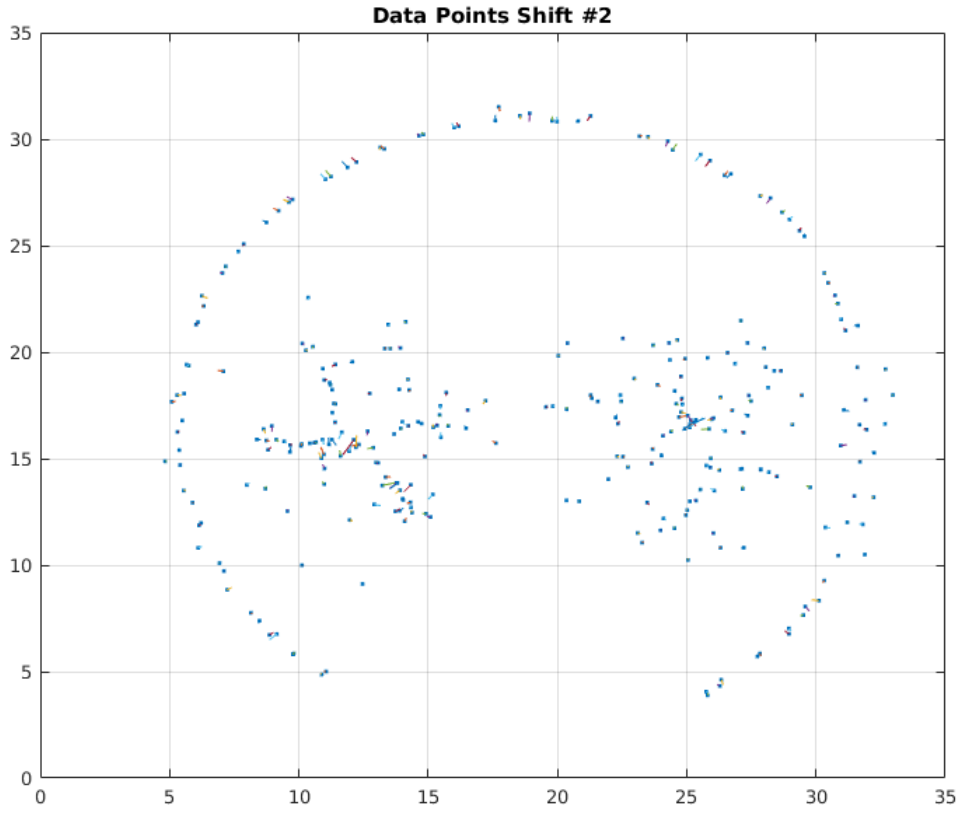
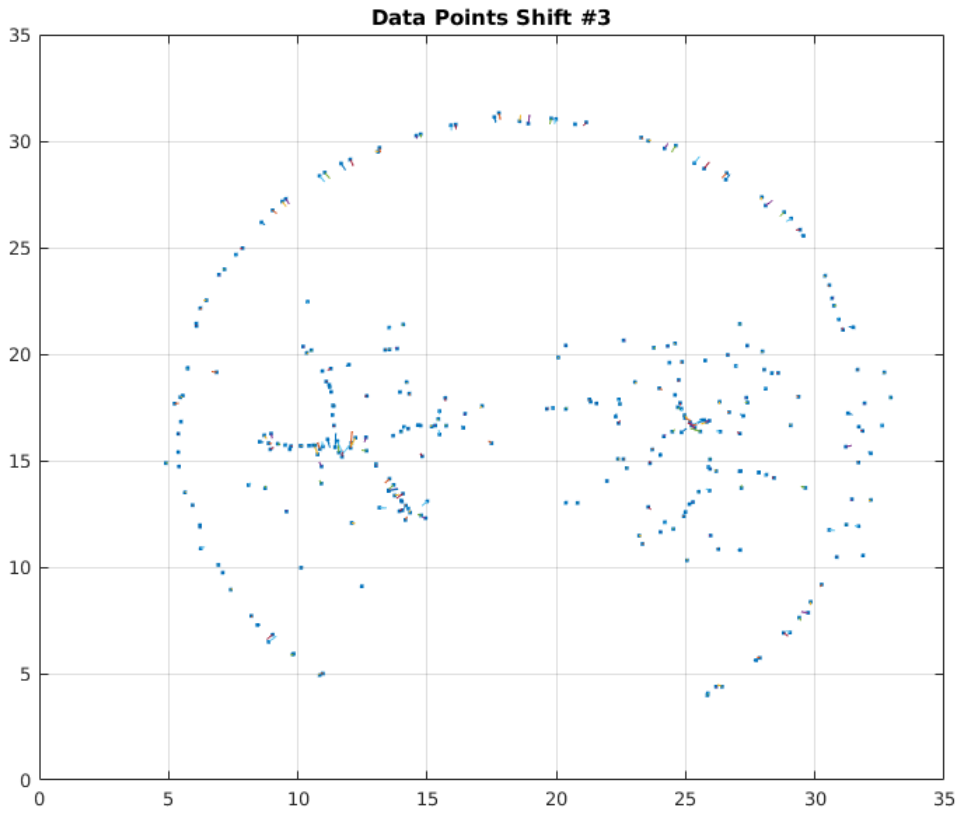


Figure 15: Iterations # 1, 2



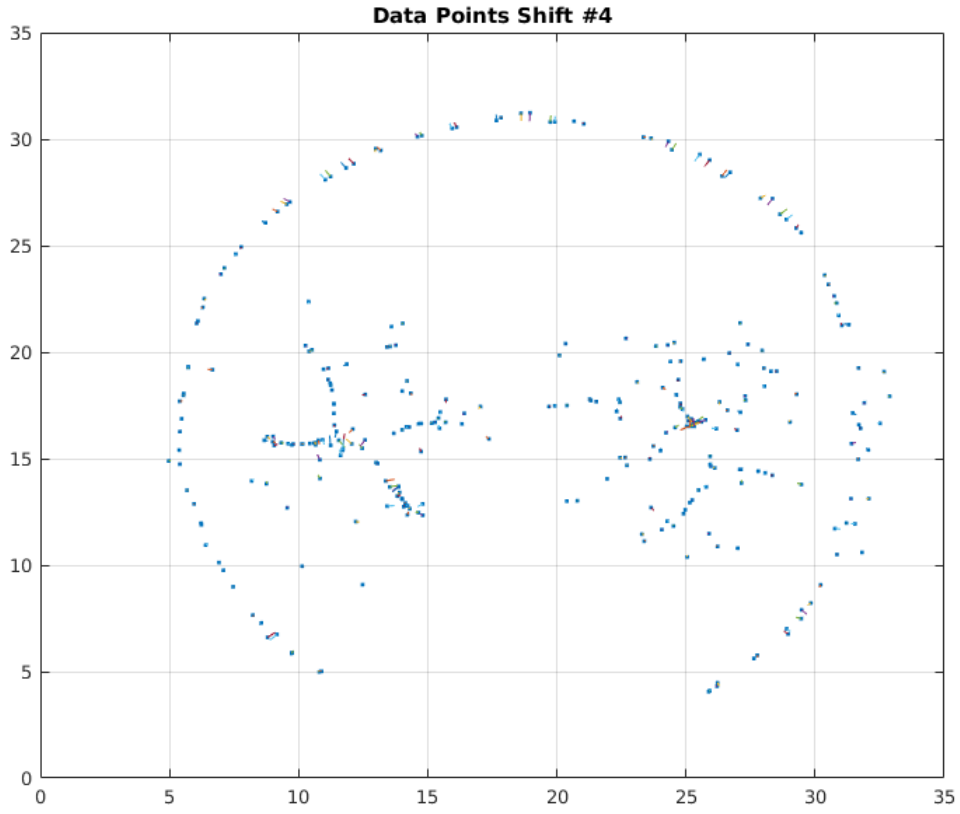
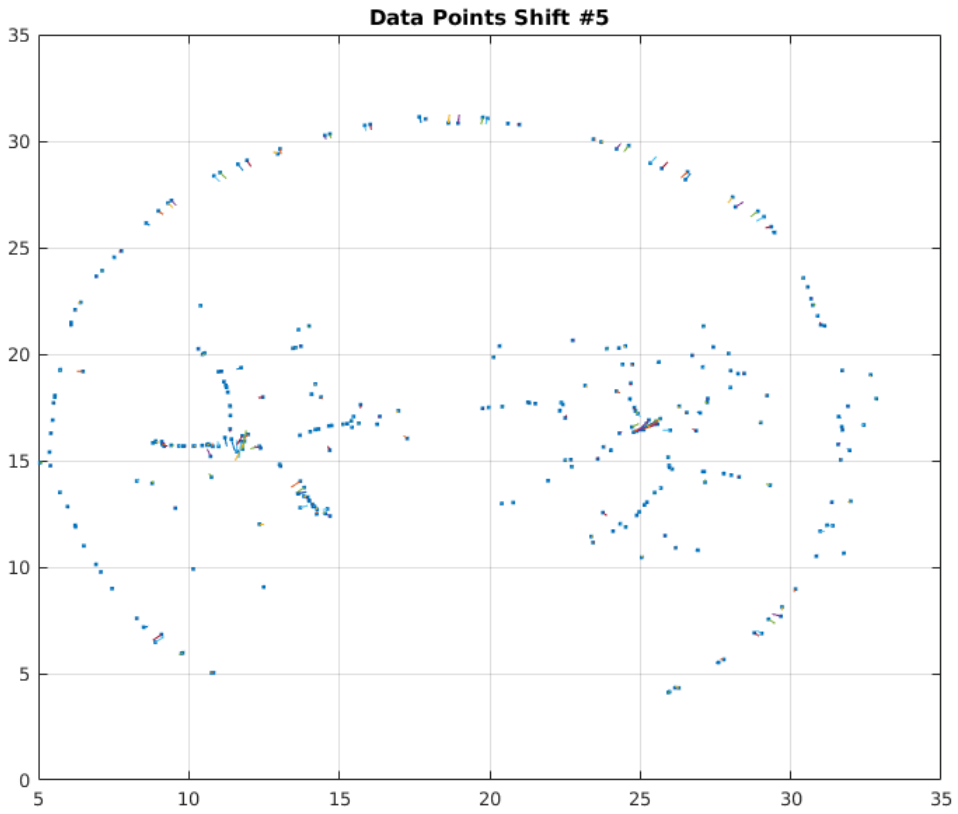


Figure 16: Iterations # 3, 4



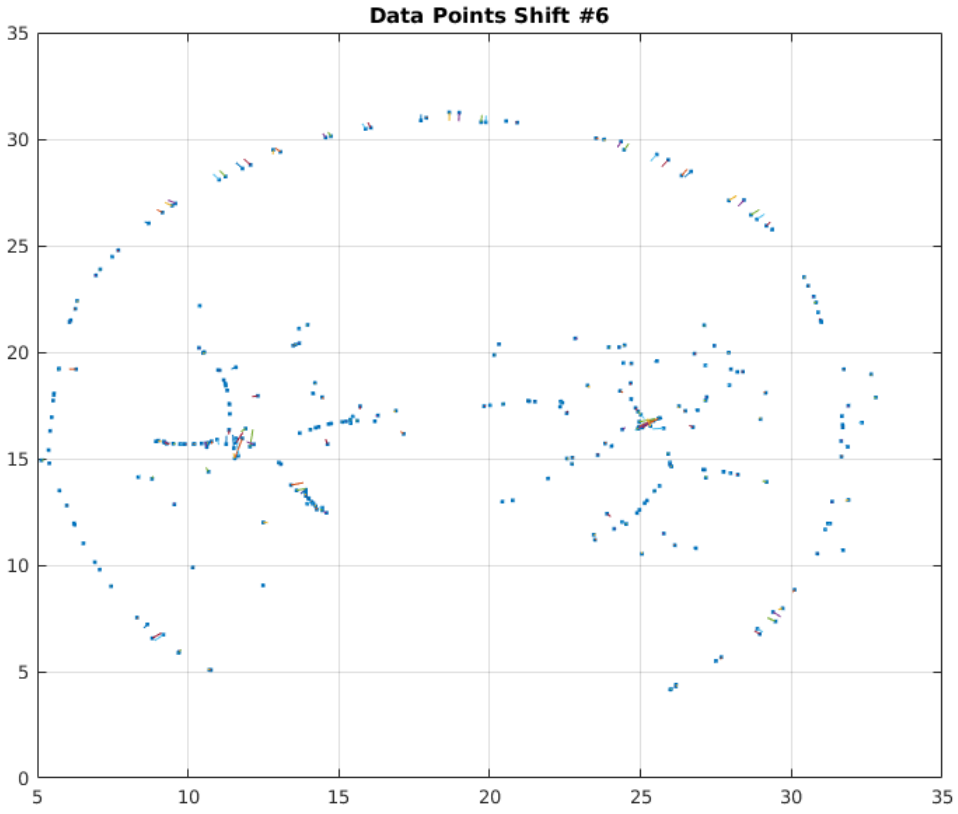
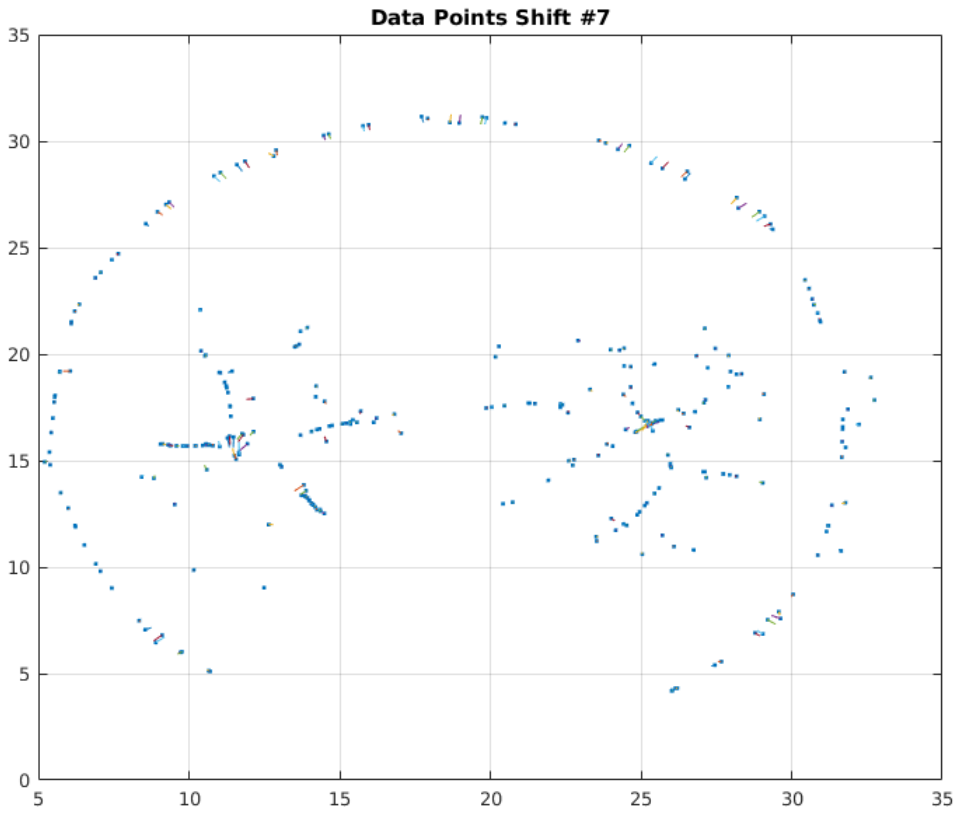


Figure 17: Iterations # 5, 6



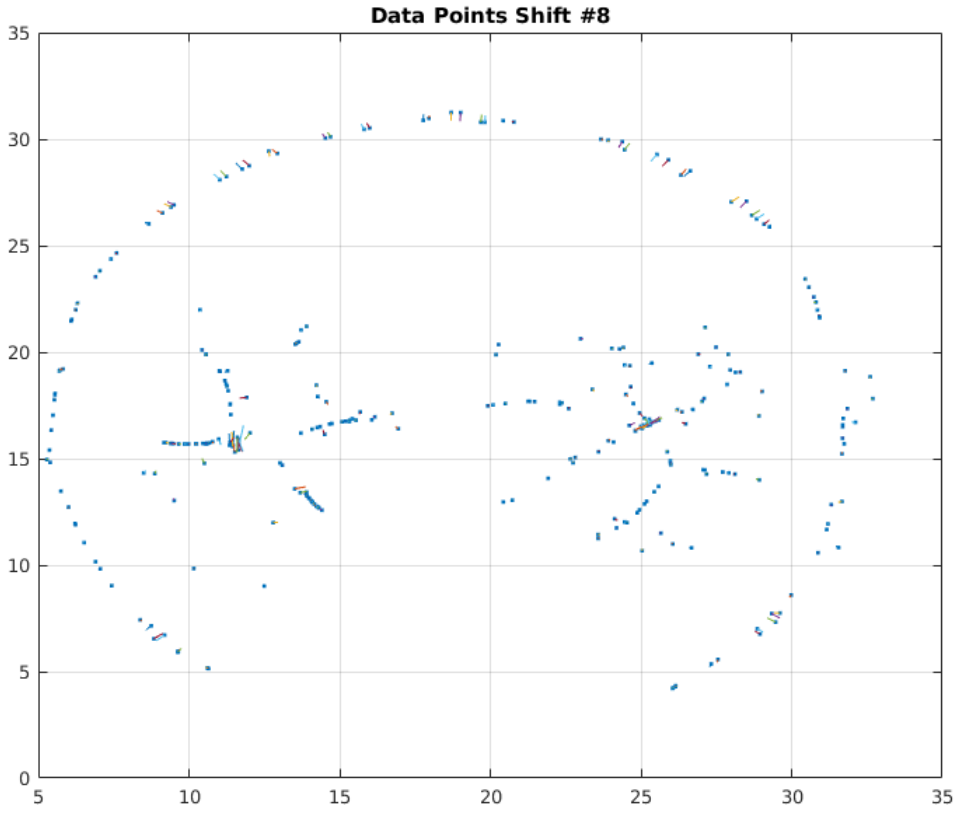
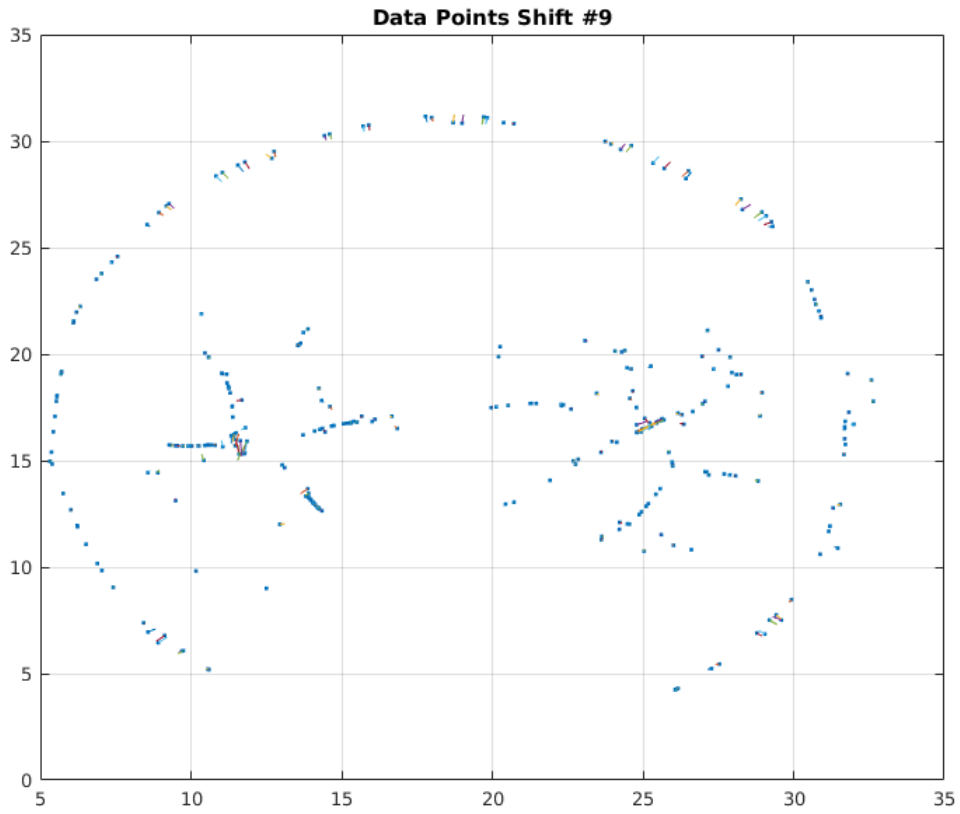


Figure 18: Iterations # 7, 8



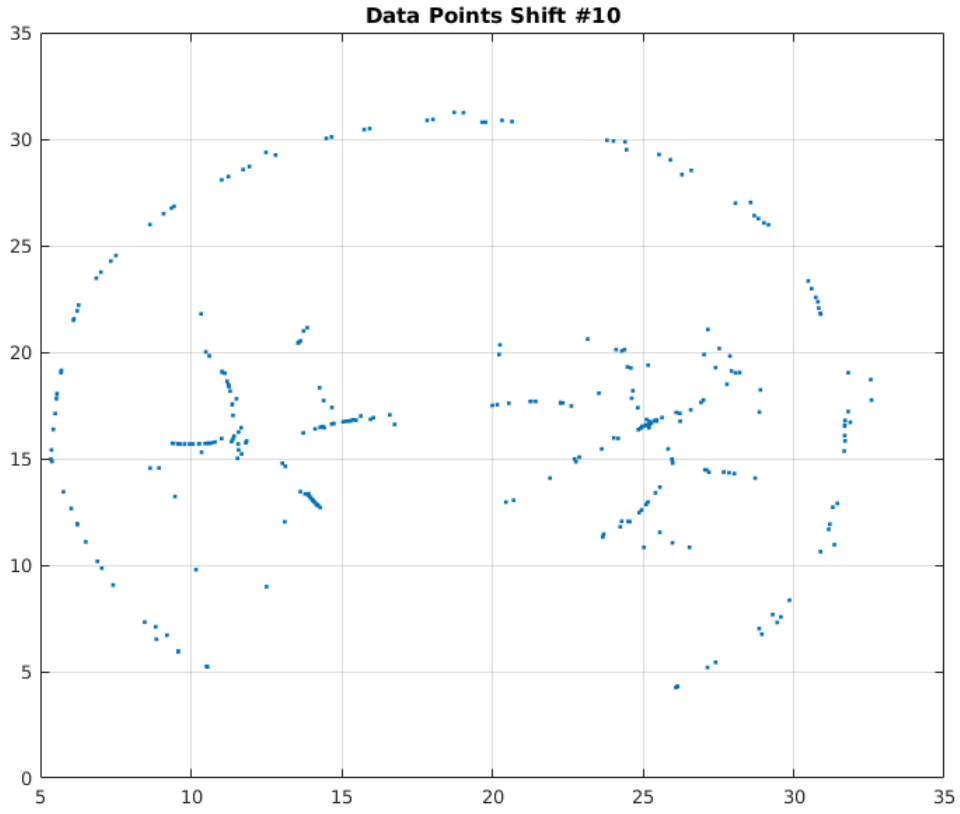


Figure 19: Iterations # 9, 10