NTNU – Trondheim
Norwegian University of
Science and Technology

Using machine learning for optimal SLA/SLO contract negotiation in 5G

Zhu Zhu

Submission date: April 2020
Supervisor: Bjarne Emil Helvik, IIK
Co-supervisor:

NTNU – Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Using machine learning for optimal SLA/SLO contract negotiation in 5G
Student:	Zhu Zhu

Problem description:

In 5G, services and slices will be provided in co-operation between many networks and service providers. The quality of service (QoS) provided to the end users and how the providers contribute to this QoS is agreed in Service Level Agreement (SLA). SLA is the contract that refers to the level of service guaranteed to a user by the network operator and to a number of quality of service parameters or metrics including dependability, latency, security, etc. To measure the quality of the offered service, the customer relies on a specific element of SLA, which is Service Level Objectives

(SLOs). Not meeting SLOs may have severe economic consequences. Fulfilling the SLOs may be expensive, while lack of control mechanisms may lead to over-dimension the provided resource. So the establishment and re-negotiations of contracts among the parties may be regarded as a game.
The problem to be solved in this project is how to optimize the strategy in the multi-provider domain regarding the system dependability SLO. More specifically how the "failure budget" should be shared among the sub-providers to achieve the lowest overall cost while meeting the dependability SLO. It is assumed that each sub-provider has two different service modes – cheap and expensive. And the optimal strategy is the decision about whether each sub-provider should shift from one mode to another or trade their downtime budget at certain points of time, with the objective to lower the overall cost. The project aims to explore one or more machine learning methods to solve the problem above.

Date approved: 2019-02-14
Supervisor: Bjarne Emil Helvik, IIK

Abstract

The objective of this master thesis is to optimize strategy in the multi-operator domain to achieve the lowest overall cost while meeting the dependability SLO by using machine learning.
To reach this objective, we identified to use reinforcement learning algorithms to interact with an discrete event simulated environment to optimize the decision making both for a single operator and for multiple-operators who are working simultaneously to provide service.
We designed a simulator which can be used to combine discrete event simulation and reinforcement learning algorithms. We created proper interfaces for our reinforcement learning agent to interact with one or multiple synchronized discrete event simulation processes. The reinforcement learning agent is able to complete its optimization learning process by getting real-time information from the simulator, deploying action orders to the simulator, and getting feedback from the simulator.
Our results showed that by using Q-learning algorithm with our simulator, the overall cost for a single operator can be reduced by 4.9% averagely.
This work has involved an analysis on how to use reinforcement learning in multi-operator scenario, what methods could be used and why these methods are considered to be appropriate for the problem. With a chapter devoted to the necessary background knowledge in reinforcement learning, the thesis should serve as an introduction to use reinforcement learning for optimal SLA/SLO contract negotiation in 5G for anyone who is interested in this field.

Preface

First and foremost, I want to thank my supervisor Bjarne Emil Helvik, whose guidance and feedback has been invaluable both to this thesis and to the specialization project preceding it. The number of feedback meetings he has held, the inspirations and guidance he has given, and the time he has spent reading drafts has exceeded my expectations.
This thesis was written across one year and three months, since I took ten months' maternity leave in the middle. I want to thank my supervisor and the faculty of IIK for the additional time they have spent on the arrangement due to my leave. I also want to thank my family, who gave me strong support of taking care of my kid when I was writing the thesis.

Contents

List of Figures

Chapter 1
Introduction

In 5G, services and slices will be provided in co-operation between many networks and service providers. The quality of service (QoS) provided to the end users and how the operators contribute to this QoS is agreed in Service Level Agreement (SLA). SLA is the contract between network service sub-operators and the service user. It usually states the sub-operators' obligations, which include the implemented dependability mechanism, the SLOs with the required *down-time threshold* and the *penalty* for *down-time overflow*, and how to share the penalty between sub-operators, etc.
The difficulties associated with sub-operators' behavior are how to react to possible failures according to SLOs and how to allocate appropriate resources.
To guarantee QoS in SLA in a cost effective way, the adequate framework based on dynamic resources allocation could thus be implemented; such a framework allows managing resources in order to minimize the risk of SLO violations while reducing the cost. Accordingly, the sub-operators' systems will be able to meet the SLO in SLAs in an economical way.

1.1 Background

The problem to be solved in this project is how to optimize the strategy in the multi-operator domain regarding the system dependability SLO. More specifically how the "failure budget" should be shared among the sub-providers to achieve the lowest overall cost while meeting the dependability SLO. It is assumed that each sub-provider has two different service modes – cheap and expensive. And the optimal strategy is the decision on whether each sub-provider should shift from one mode to another or trade their downtime budget at certain points of time, with the objective to lower the overall cost. The project aims to explore one or more machine learning methods to solve the problem above.
Existing work that address the related topic can be classified in the following two aspects:

i. What methods could be used to dynamically allocate resource along the service period and among the different service sub-suppliers?
There is a trade-off between fulfilling the SLA availability, and the amount of resources needed in order to do so. The paper "Guaranteeing Availability Requirements in SLAs using Hybrid Fault Tolerance"[AJGW15] introduces the potential hybrid fault tolerance approach. Assuming there are two different technologies A and B to be used. A provides relatively higher level of performance with relatively higher cost than B. Then two intuitive ways to combine the use of fault tolerance technologies A and B are studied: i) *Spend and Save*. ii) *Save and Spend*. *Spend and Save* is a hybrid fault tolerance approach that uses technology B at the beginning of the SLA. This initial phase is called *spend*. In order to have a tight control on the risk of violating the SLA availability parameter, the provider has the opportunity to switch to technology A at any time, starting the *save* phase. However, it is desired to delay the shift as much as possible, due to the cost saving consideration. The main challenge of this approach is to decide the appropriate point to switch to the *save* phase. *Save and Spend* is a hybrid fault tolerance approach that uses technology A at the beginning of the SLA period. This initial phase is called *save* phase, since there is a very high probability to satisfy or exceed the SLA availability parameter. However, the use of technology A implies more costs and resources. Therefore, it is desired to switch to technology B as early as possible. The mechanism has to verify that the the saving at the *save* phase is enough.
The Hybrid Fault Tolerance method is used to solve the individual service provider's problem - fulfilling the SLA availability while saving cost and resource as much as possible. However, future communication based services will be provided by a set of independent market actors who cooperate in providing the service. SLA will take the roll to guide and govern the related service activities among multiple operators and the user. The operators must distribute the obligations and benefits from this agreement[Zhu].
ii. What machine learning models could be chosen to satisfy the research objective. In the paper "SLA Violation Prediction In Cloud Computing: A Machine Learning Perspective" [RAH18], two machine learning models: Naive Bayes and Random Forest Classifiers have been explored to predict SLA violations. As SLA violations are events that rarely happen in the real world (0.2%), several re-sampling methods are used to overcome the challenge. And Random Forest with SMOTE-ENN re-sampling are found to have the best performance among other methods with the accuracy of 0.9988%. It is also mentioned that mem_requested is the most important feature of the random forest model in predicting violations. And random forest can be used real-time thanks to its relatively high speed.

In another paper "Failure prediction using machine learning and time series in optical network"[ZWL17], a performance monitoring and failure prediction method in optical networks based on machine learning is proposed. The primary algorithms

of this method are the Support Vector Machine (SVM) and Double Exponential Smoothing (DES). Though the method is not particularly used for network service failure prediction, the methodology used can still be of interest and consideration for this study[Zhu].
Many of the previous studies are devoted to SLA violation predictions. In this study, we will review and analyze the feasibility of existing methods, and try to find Machine Leaning methods for process control rather than violation prediction. We aim to apply Machine Learning tool for decision making optimization to fulfill the SLOs while minimizing the total cost for operators.

1.2 Objectives

The main objectives of this Master's project is to find the optimal policy for network service operators to minimize cost while fulfilling dependability SLO in SLA by using Machine Learning.

- Question 1: What kinds of machine learning methods are appropriate for policy optimization in this research problem and why?
- Question 2: How to achieve policy optimization for the research problem with the selected machine learning methods?

1.3 Structure

The rest of the thesis is structured as follows.
Chapter 2. Background Theory and Motivations gives an introduction to the background theory necessary to understand the chosen methods and the rest of the thesis, and the motivations for selecting methods and algorithms that will be used in this study.
Chapter 3. Problem Analysis and Methodology analyzes the problems relate to reinforcement learning and explain the methodology used for the simulator design in the upcoming chapter.
Chapter 4. Simulator Design describes the design of the simulator and how to build the interaction between the simulator and reinforcement learning agent.
Chapter 5. Results and Discussion presents the workflow of the reinforcement learning implementation with parameter optimization and final result.
Chapter 6. Summary summarizes the study findings and concludes with propositions for future work.

Chapter 2
Background Theory and Motivation

This chapter presents theoretic knowledge and terminology for the reader to understand the rest of the thesis. Section 2.1 briefly introduces concepts and fundamentals in machine learning and deep learning, section 2.2 explains the motivations for applying reinforcement learning and details on reinforcement learning, and section 2.3 elaborates the specific reinforcement learning algorithms appropriate for our problems.

2.1 Machine Learning

Machine learning addresses the question of how to build computers that improve automatically through experience[JM15]. It is about learning from data and making predictions and/or decisions[Li18].

2.1.1 Sub-branches of Machine Learning

There are many approaches that can be used when conducting machine learning. They are usually grouped into the areas of supervised learning, unsupervised learning, and reinforcement learning.

Supervised Learning

Supervised learning is the most widely used machine learning methods today. The applications of supervised learning includes email spam classification, fingerprints recognition, object recognition over images and so on.
Supervised learning is where the training data is a collection of (x, y) pairs and the goal is to predict y^* in respect to x^*. The predictions are generally formed via a learned mapping f(x), which produces an output y for each input x.[JM15] As each of the input data x has a corresponding output data y. We call the data as labeled data.

Supervised learning can be further group into two subcategories: Classification and regression.

- **Classification**: A classification problem is when the output variable is a category, such as “spam” and “not spam”, or “cat” or “dog”.
- **Regression**: A regression problem is when the output variable is a real value, such as “revenue” or “price”.

Unsupervised Learning

Unsupervised Learning deals with clustering and finding relations in unlabeled data[SZ19]. It generally involves the analysis of unlabeled data under assumptions that the data has structural properties(e.g. probabilistic, algebraic, or combinatorial)[JM15].

Reinforcement Learning

Reinforcement learning is known as a semi-supervised learning model in machine learning. Here the information available in the training data is intermediate between supervised learning and unsupervised learning. Instead of the form of the paired data (x, y) where y is the correct output for the given input x, the training data in reinforcement learning are assumed to provide only an indication in response to whether an action is correct or not; the problem of fining the correct action remains when an action is incorrect[JM15].

2.1.2 Deep Learning

In machine learning algorithms we have x and y, which we can also call them input layer and output layer. In deep learning, there are one or more hidden layers between input layer and output layer, and weights on links between units from layer to layer. At each layer except input layer, we compute the input to each unit, as the weighted sum of output of units from the previous layer, with non-linearity(or activation function), such as logistic, tanh, rectified linear unit (ReLU), and Softmax. After computations flow forward from input layer to output layer, we compute error derivatives backward at output layer and each hidden layer, and backpropagate gradients towards the input layer, so that weights can be updated to optimize some loss function.[Li18]

2.2 Reasons for Choosing Reinforcement Learning

Supervised Learning can solve a lot of problems, e.g. classifying images, translating text. However, the problem of SLO fulfillment optimization is a decision-making problem which is similar to the problems of playing games or teaching a robot to

take an action. Supervised learning is not thought to be appropriate to deal with this kind of problem. Why?
Suppose we had a data set containing all the history of the service provided by multiple operators. Then we could use the service state as input x and the optimal action that are taken for that state as output y. In theory it sounds to work but in practice a few issue arise.

i. Collecting such a data set might be quite expensive or unfeasible.
A typical supervised learning algorithm generally requires an immense amount of prior data (e.g., typically more than 100,000 items) for training the decision model. With this scale, the controller should have a strong storage capacity and powerful calculating ability. In a real network, determining whether a node is available is performed by many different kinds of equipment. If the controller monitors and predicts the state of each piece of equipment with that scale, it will suffer a seriously heavy burden[ZWL17].

ii. Data sets are incomplete for all domains we need.
Consider that we want to use machine learning algorithms to find out the best time point to take the action to shift mode or trade down-time. The problem is that it's impossible to have the data sets that involve all the time-steps that the system actually shifted its mode or traded down-time with another operator.

iii. This approach aims to imitate a human expert rather than actually learn the optimized strategy.
Suppose that we have the data sets that contains the output y, which is the lowest cost for operators. We don't know whether the cost could be reduced further. If we use the existing lowest cost as the target for supervised learning, it won't be possible for us to find optimal solution to achieve even lower cost.

Reinforcement learning comes to address the problems here. It attempts to learn optimal actions in an interactive environment by trial and error. In other words, it learns the optimal strategy by sampling actions and then observing which one leads to the desired outcome. Unlike supervised learning that learns the optimal action from a label, reinforcement learning learns it from a time-delayed label called a reward. The reward tells us whether the outcome of whatever we did was good or bad. Hence, the goal of reinforcement learning is to optimize actions for maximum reward.

2.3 More on Reinforcement Learning

In this section, we are going to present an overview of reinforcement learning. For reinforcement learning experts, as well as new comers, we hope this overview would be helpful as a reference and provide as much relevant information as possible for the readers to understand the rest of the thesis.

2.3.1 Terminologies

Actions

Actions are the methods for the *Agent* to interact with and change its *environment*, and thus transfer between *states*. Every *action* performed by the *Agent* yields a *reward* from the *environment*. The decision of which *action* to choose is made based on the *policy*.

Policy (π)

The *policy*, denoted as π (or $\pi(a|s)$), is a mapping from some *states* to the probabilities of selecting each possible *action* given that *state*. For example, a greedy *policy* outputs the *action* with the highest expected Q-Value for every *state*.

Agent

Agent is the reinforcement model that learns to maximize the *rewards* it is given by the *environment*.

State

State is every possible scenario the *agent* encounters in the *environment*. The *agent* transits between different *states* by performing *actions*. The terminal *state* usually marks the end of an *episode*. And it returns to the initial *state* when a new *episode* begins.

Environment

Environment is where the *agent* gets feedback on its *actions*, either directly or indirectly. The *environment* changes as the *agent* performs actions; every such change is considered a state-transition.

Reward

Reward is a numerical value received by the *agent* from the *environment* as a direct response to the *agent*'s *actions*. The *agent*'s goal is to maximize the *overall reward* it receives during an *episode*, so the *agent* is motivated by the *reward* in order to

take a desired *action*. *Reward* can be a positive value, which emphasizes a desired action, or a negative value, which discourage an undesired action.

Overall reward

Overall reward, sometimes referred to as "expected return", is the expected reward over an entire *episode*.

The Bellman Equation

Assume we already know what is the expected *reward* for each *action* on each step. Then we'll choose the sequence of *actions* that will eventually generate the highest *reward*. This cumulative *reward* is often referred to as *Q Value*, and *The Bellman Equation* is the mathematical formalization of this strategy.
[SB]

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

In the equation, the *Q Value* yielded from performing *action* a at *state* s, equals to the immediate *reward* r(s,a), plus the highest *Q Value* from *state* s' (which is the *state* it arrives in after performing *action* a from *state* s). The *agent* will receive the highest *Q Value* from s' by selecting the *action* that maximizes the Q Value.

Q Value

Q Value (Q Function), usually denoted as Q(s,a) (sometimes as $Q(s,a;\theta)$ in Deep RL), is the sum of the instant reward and the discounted future reward (of the resulting state). "Q" is the abbreviation of the word "Quality".

Q-table

Q-table is a matrix where we store the Q-values for each state and action.

α - learning rate

α (alpha) is the learning rate ($0 < \alpha \leqslant 1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.

γ - discount factor

γ controls the importance of long term *reward* versus the immediate *reward*.

Exploitation Exploration

Reinforcement learning has no pre-generated data sets which they can learn from. Instead, it creates its own experience and learn by trial and error. Exploration is something about the *agent* explores the *environment* — the *agent* tries many different *actions* in many different *states* in order to learn all available possibilities and find the path which will maximize its overall reward. If all the *agent* will do is explore, it will never find the best path to maximize its overall reward — it must also use the information it learned to do so. Exploitation is something about the *agent* exploits its knowledge to maximize the rewards it receives[SB]. The trade-off between the two is one of the biggest challenges of reinforcement learning.

Greedy Policy, ϵ-Greedy Policy

A Agent constantly performs the action that is believed to yield the highest expected reward is called greedy policy. Obviously, such a policy will not allow the Agent to explore at all. An ϵ-greedy policy is often used instead to allow some exploration: we decide whether to pick a random action or to exploit the already computed Q-values. This is done simply by using the ϵ parameter in the range of [0,1] and comparing it to a random number in the range of [0, 1].

Markov Decision Process (MDP)

MDPs are a classical formalization of sequential decision making, where actions not just yield immediate *rewards*, but also influence subsequent *states* and through those future *rewards*. Reinforcement learning problems can be mathematically formulated into MDPs, where we define the interaction between a learning *agent* and its *environment* in terms of *states*, *actions*, and *rewards*.[SB]

Episode

Episode can be broken down into sequences in which the *agent* interacts with its *environment* until it reaches a certain terminal *state* that resets the *environment* to its initial *state*.

2.3.2 Methods

Q-learning

Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best *action* to take given the current *state*, where a *policy* is derived by choosing the *action* with the highest Q-value in the current *state*.

Deep Q Network(DQN)

Deep Q Network is the combination of Q learning and deep learning (specifically, a deep convolutional Artificial Neural Network (ANN))[SB]. It comes into play when the number of states and actions are very large. The idea is to replace the Q-table with a neural network that works to approximate Q-Values. It is referred to as the approximating function, and denoted as $Q(s, a; \theta)$, where θ represents the neural network's weights.

Policy Gradient

Policy Gradient is a kind of reinforcement learning methods that learn a parameterized *policy* that can choose *actions* without consulting a value function. The parameter vector is denoted as θ. The methods learn to optimize θ (which are usually the neural-network's weights) based on the gradient of some performance measure J(θ) with respect to θ.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

[SB]

where α is a learning rate.

Chapter 3
Problem Analysis and Methodology

This chapter will analyze our study problems relate to reinforcement learning and explain the methodology used for the simulator design in the upcoming chapter. In section 3.1, we are going to setup and analyze the reinforcement learning problems. In section 3.2, we will discuss what reinforcement learning methods can be used and why. In section 3.3, we will elaborate the possible simulation tools for reinforcement learning implementation and our choice.

3.1 Reinforcement Learning Problem Setup

In this study, we are trying to deal with two consecutive problems. The first problem is to minimize cost for a single operator who provides service. The second is to minimize the overall cost for multiple operators who provide service simultaneously.

3.1.1 Single operator scenario

The *environment* is the simulation of one operator provides service either in cheap mode (denoted as mode A) or in expensive mode (denoted as mode B). The *agent* is the reinforcement learning model who will make sequential decisions at certain time-steps on whether to provide the service in mode A or mode B. The actions(either 1 or 2) is the mode selecting decisions that the *agent* makes at every *state* it arrives. In simple terms, the *agent* will first observe and construct its own representation of the environment (state). The *state* is defined as two-dimensional: The first domain is time; the second domain is accumulated down-time in respect to time. The *agent* will take certain *actions* like shift the mode at different *state* and observe how would the service availability response (next *state*). Assume the *agent* is in cheap mode. If the accumulative down-time is longer than expected (receiving a negative *reward*), the agent will switch to expensive mode (updating the *policy*); if the accumulative down-time is shorter than expected (receiving a positive *reward*), the *agent* will stick to the cheap mode. The *agent* will repeat the process until it finds a *policy* (what to

do under different circumstances) that minimizing the total cost (maximizing the total *rewards*).

3.1.2 Multiple operators scenario

To be simple, here we assume there are only two operators, denoted as operator 1 and operator 2. The *environment* is the simulation of the two operators provide service either in mode A or in mode B simultaneously. The *agent* is the reinforcement learning model who will make a bunch of sequential decisions at each time-step. The *actions* is the decisions at each time-step: 1) whether to provide the service in mode A or mode B for operator 1 and operator 2 respectively; 2) whether to trade down-time budget or not between operator 1 and operator 2; 3) if trade down-time budget, whether operator 1 sells its down-time budget to operator 2 or the other way around; 4) how much down-time budget do they sell or buy respectively. Since we need to observe the two simultaneous process, the *state* should be defined as three-dimensional: the first domain is time; the second domain is accumulated down-time in respect to time of operator 1 ; and the third domain is accumulated down-time in respect to time of operator 2.

3.2 Chosen Reinforcement Learning Method

3.2.1 Method for Single Operator Scenario

We are going to apply Q-learning to the single operator scenario. The choice is made based on the following reasons: 1) Q-learning is one of the fundamental reinforcement learning methods, it directly parameterize and update value functions or policies without explicitly modeling the environment. So it is typically simpler, more flexible to use. 2) Q-Learning tries to have complete and unbiased knowledge of all possible actions. Its learning procedure is to figure out the quality of each possible action, and select the best one. Since we only have two possible actions in the single operator scenario, Q-learning is capable of performing the action selection.

3.2.2 Method for Multiple Operator Scenario

As we discussed in Section 3.1.2, the number of actions in the multiple operator scenario becomes much larger compared with that in the single operator scenario. Exploring all possible actions using an ϵ-greedy strategy such as in Q-learning might take too long. DQN is an extension of Q-learning that combines deep learning. It could handle the scenario of Q-learning when the number of states and actions becomes very large. Intuitively it could be applied to deal with our multiple operator problem. Besides, policy gradient is the kind of method that learns in a more robust way, by not trying to evaluate the value of each action — but simply evaluating which

action should it prefer. When the number of actions is large in the environment, policy gradient algorithms guarantees stronger convergence than action-value methods. In policy gradient, with continuous policy parameterization, the action probabilities change smoothly as a function of the learned parameter, whereas in ϵ-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change leads to a different action with the maximal value[SB].

3.3 A discrete event simulation environment

A large number of iterations are needed before a RL algorithm works. Therefore, a simulated environment that can reflect the real world problem is needed. Furthermore, to model the system dependability, we usually apply discrete event simulation method that can model the operation of a system as a discrete sequence of events in time. We know there are some great open source solutions for reinforcement learning such as OpenAI Gym, and some tools for discrete event simulation such as the Python module Simpy. Let's evaluate the two tools.

OpenAI Gym

OpenAI Gym is a toolkit that was created for developing and comparing reinforcement learning algorithms. It is also the most widely used toolkit that supports teaching agents for the variety of applications ranging from playing video games like Atari to problems in robotics. The OpenAI Gym library is a collection of simulated environments. These environments are virtual, and they have a shared interface, which allows you to write general reinforcement learning algorithms. It also allows source building for environment modification and adding new environments[w2].

Simpy

The SimPy package offers a straightforward way to build process-based discrete-event simulation environment. It allows us to create event-driven simulators which step the environment from event to event[w3].

Although OpenAI Gym is a quite convenient tool for performing reinforcement learning, we choose Simpy over OpenAI Gym due to the following reasons: 1) OpenAI Gym is not designed for discrete event simulation, it may miss some necessary functions we need when we simulate the network service environment; 2)Time is important for our decision making, so our reinforcement learning agent needs to be time sensitive. Recall Section 3.1 where we defined the state with two domains: one is time, the other is the accumulated down-time at the time. Therefore the agent is necessarily to be synchronized with the environment so as to get the on-time state;

3)In our problem, we don't necessarily to interact with a virtual environment such as the environments in OpenAI Gym.
As we choose to apply Simpy to create our discrete event environmemt, we need to create the interfaces for reinforcement learning from draft.

Chapter 4
Simulator design

4.1 Introduction

This chapter will introduce a simulator that combines the discrete event simulation environment with reinforcement learning algorithms to achieve the objective of this study. First, we are going to illustrate the simulator in the single operator scenario in section 4.2, then we are going to explore how the simulator could be expanded into the multiple operator scenario in section 4.3.

4.2 Simulator for Single Operator Scenario

Since we want our reinforcement agent to be synchronized with the discrete event simulation environment, we need to create two synchronized processes in the single operator scenario: one is the work process which has a sequence of discrete events of "fail" and "recover", the other is the step process, which keeps track on the work process and get the state from the work process at certain time-steps.

4.2.1 The State Transition and the Sequence of Processes

The state transition diagram and the sequence diagram of the simulator for single operator scenario are shown as below.
The work process is a discrete event process. It includes two subordinate discrete event processes - the states become up and down in each mode. The work process serves as an reinforcement learning environment and it provides interfaces for reinforcement learning agent. The step process is a synchronized process with the work process. It allows the reinforcement agent to apply those interfaces and to implement reinforcement learning during the simulation process.

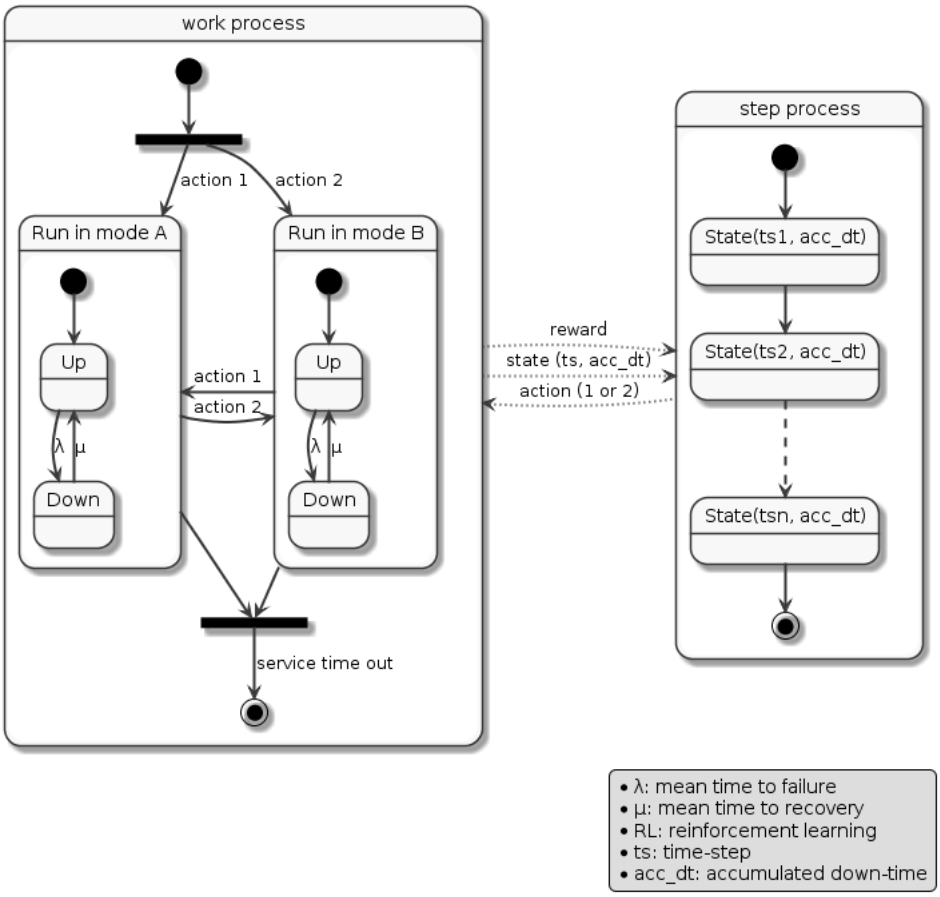

Figure 4.1: State transition diagram of single operator scenario

4.2.2 Work Process

The work process of the simulator is to simulate the system that run by the service operator. The process runs in a certain period, which is called service period.
The operator provides service either in mode A(cheap mode) or in mode B(expensive mode) in a service time period. The system fails according to a Poisson process of intensity λ_A and λ_B respectively in mode A and mode B. And it recovers following normal distribution with mean time μ_A and μ_B and standard deviation σ_A and σ_B respectively in mode A and mode B. The major difference between mode A and mode B is the recovery time. The system is able to recover from failure within a shorter time in mode B than in mode A.

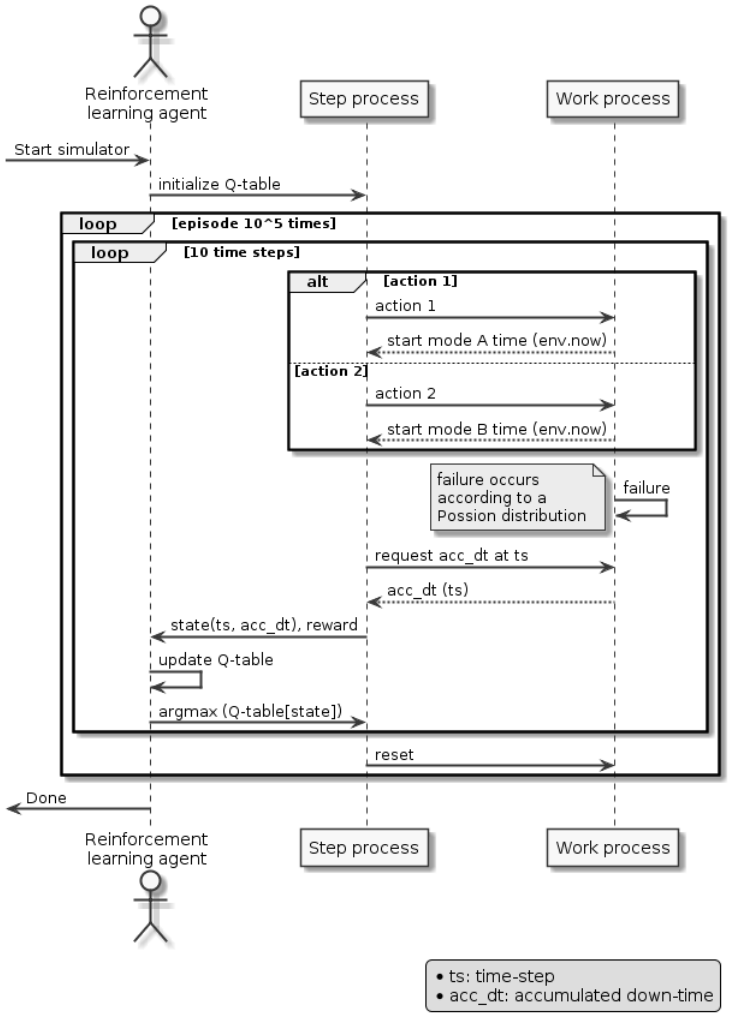

Figure 4.2: Sequence diagram of single operator scenario

Total Cost, Operational Cost and Penalty

The total cost C_T of running the system includes operational cost C_O and penalty C_P:

$$C_T = C_O + C_P$$

The operational cost C_O is the sum of the operational cost in mode A denoted as C_{OA} and the operational cost in mode B denoted as C_{OB}:

$$C_O = C_{OA} + C_{OB} = C_A \times T_A + C_B \times T_B$$

where C_A, C_B are cost per minute respectively in mode A and mode B ($C_A < C_B$), and T_A, T_B are running time respectively in mode A and mode B.
The operator will receive a penalty if the accumulate down time when the service period ends is longer than the down time budget pre-determined in SLOs. The penalty function is called at each time-step to get the penalty value which will be parsed to the step process as a negative reward for reinforcement learning agent. The penalty function is as followed:

$$C_P = \begin{cases} 0 & t_{acc\_dt} <= threshold \\ C_{ppm} \times (acc\_dt - threshold) & acc\_dt > threshold \end{cases}$$

where C_{ppm} denotes penalty per minute, acc_dt denotes accumulate down-time, and *threshold* denotes down-time threshold for the operator in the service period.
The graph of the penalty function is as below.

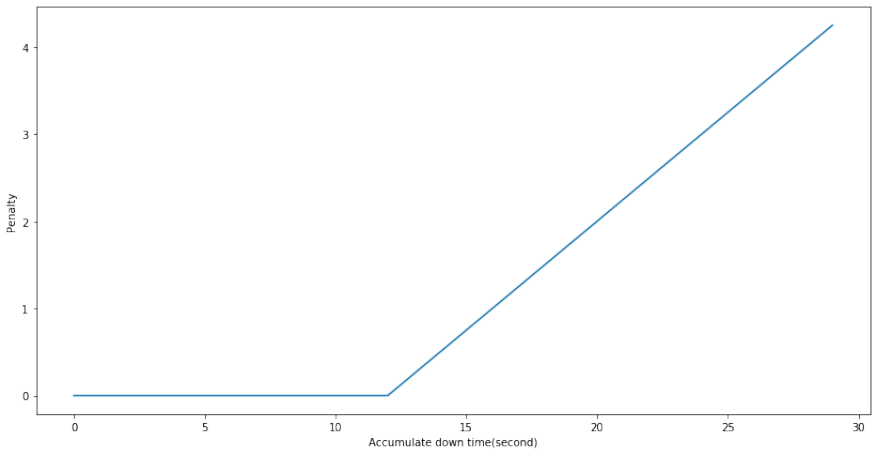

Figure 4.3: Penalty function: penalty relates to accumulated down-time

4.2.3 Step Process

The step process is implemented for interaction between the environment (work process) and the reinforcement agent. It has two functions: 1)for the agent to keep track of the state and get reward from the system at certain time-steps; 2)to execute the action determined by the agent at each of the time-steps.

4.2.4 Apply Reinforcement Learning

We know that reinforcement learning is intended for the agent to determine appropriate actions at each state to maximize the long-term reward. Now we are going to define the key elements *agent*, *state*, *actions* and *reward* in our simulator.

Agent

To be simple, the agent is our reinforcement learning model.

State

The agent observes the state at each time step and decide the action to take next. We define the state as two dimensional: The first domain is time-step, which slices the simulation time of one *episode* into a sequence of equal-length intervals; the second domain is accumulated down-time slots, where we slice the possible highest accumulated down-time at each time-step into a sequence of equal-length slots, and any possible value of the accumulated down-time will fall into one of these slots.

Action

The *agent* selects between two actions: 1) run in mode A; 2) run in mode B. The *agent* chooses its action based on the applied reinforcement learning algorithm.

Rewards

Our objective is to achieve lowest total cost. So in our Q-value function, the *rewards* will be negative rewards which relates to our cost. We set the immediate reward at each time-step as the sum of the negative penalty and the negative operational cost during one time-step.

$$reward_{time\_step} = -[C_P + (C_A * t_A + C_B * t_B)]$$

where t_A is the time in mode A and t_B is the time in mode B during that time-step.

Q-Learning

In Q-learning, Q-values are usually initialized to an arbitrary value, and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated using the equation:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action)+\alpha(reward+\gamma maxQ(next\ state, all\ actions))$$

Q-values are stored in a Q-table. Q-table is a matrix where we have a row for each state and a column for each action. Usually the states are first initialized to 0, and the values are updated along training. In our problem, we created the Q-table as below:

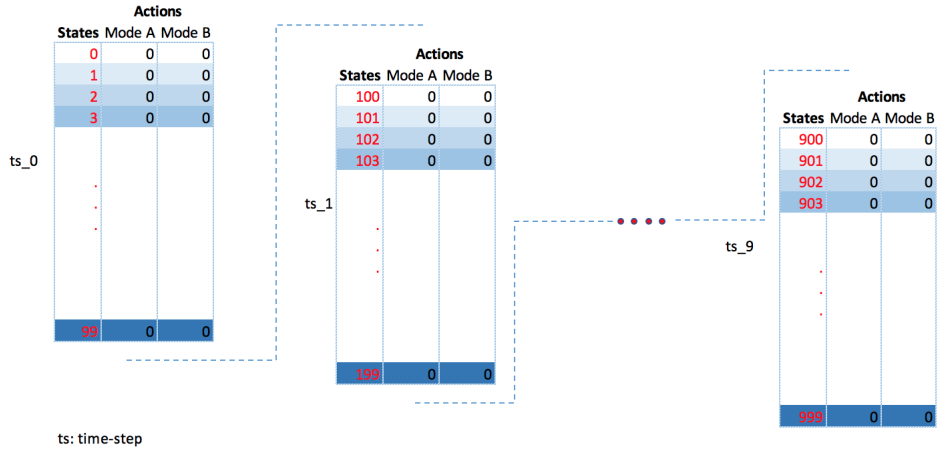

Figure 4.4: Q-Table

4.3 Simulator for Multiple (Two) Operator Scenario

Recall Section 3.1.2 on the multiple operator scenario, we need one more work process for each additional operator. To be simple, we design the simulator for two operators. We have three synchronized processes in our simulator: one step process and two work processes. The step process is responsible to keep track of the two work processes at each time step, getting the accumulated down-time and reward from the two work processes and sending the action decisions to them respectively . The two work processes are treated independently when the agent makes decisions on whether to shift mode, whether to trade down-time, who is the buyer and who is the seller, and how much down-time budget to trade.

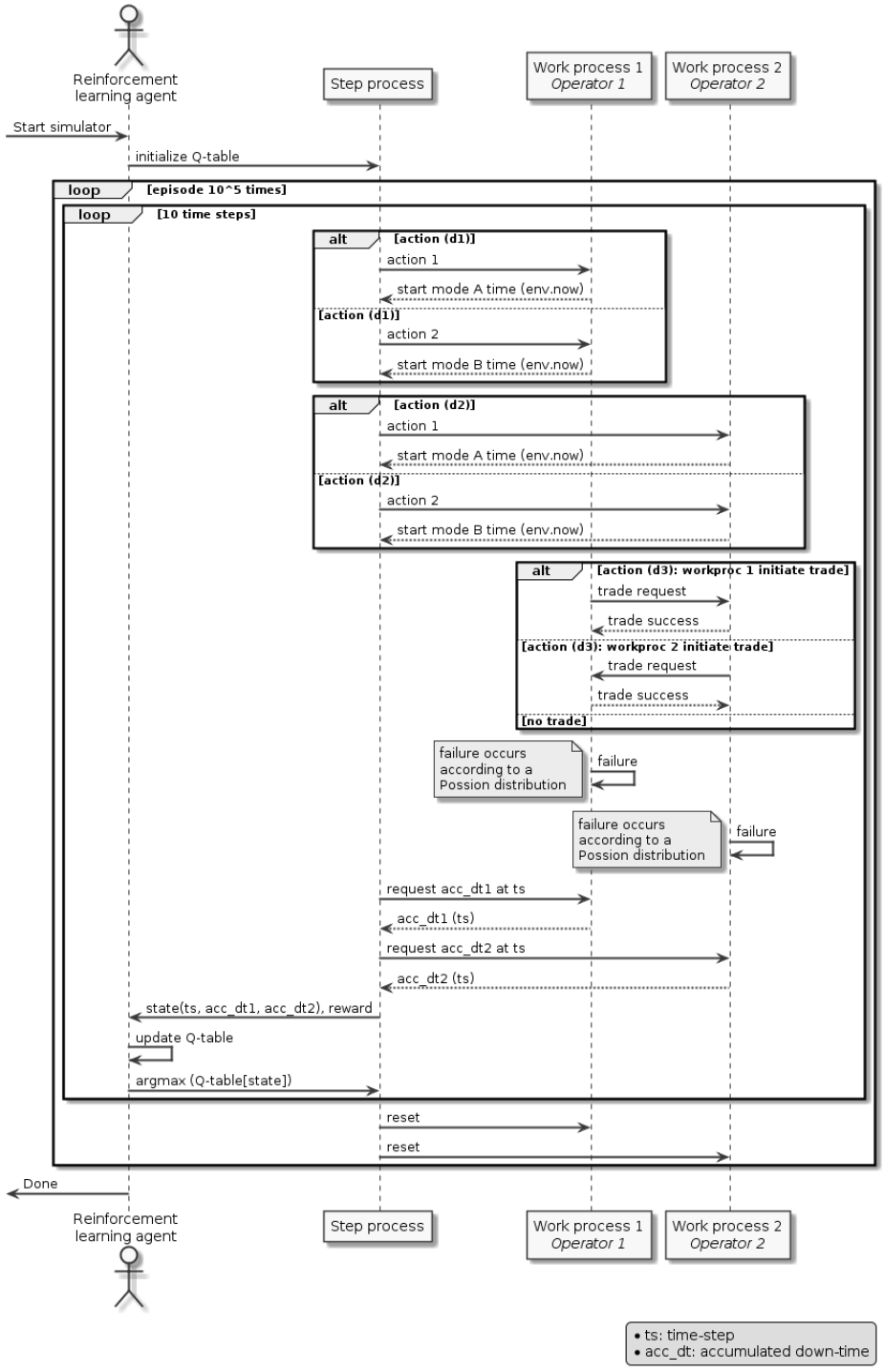

Figure 4.5: Sequence diagram of multiple operators scenario

Chapter 5
Results and Discussions

5.1 Optimising parameters

5.1.1 Parameters of Q-learning

When we vary parameters in reinforcement learning, we observe the results over a number of episodes where we hope to show stability and convergence. This is a two dimensional comparison (x = episode and y = output). And when we want to observe the results if we vary a parameter this becomes three dimensional (x = episode, y = output and z = parameter). So we use multiple plots for each parameter choice. And we use average total cost for each 10000 iteration as the output y.

α - learning rate

α (alpha) is the learning rate ($0 < \alpha \leqslant 1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.

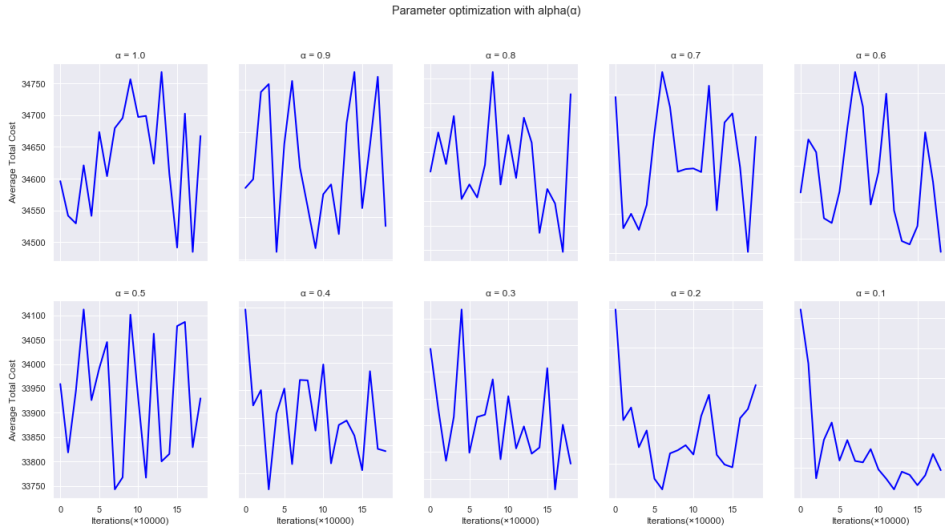

Figure 5.1: Tuning α for optimization

As we see from the above figure, in our problem, the smaller the learning rate, the better convergence.

γ - discount factor

γ (gamma) is the discount factor ($0 \leqslant \gamma \leqslant 1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to 1) captures the long-term effective award, whereas, a discount factor of 0 makes our agent consider only immediate reward, hence making it greedy.
As we prefer future reward to immediate reward, we need to set γ as the highest value. After tuning the γ parameter, we see the result is what as we expected.

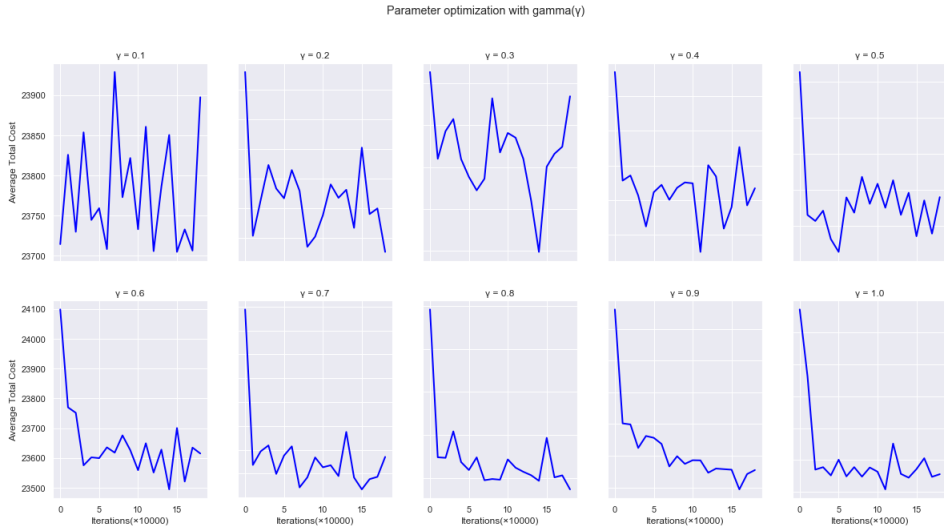

Figure 5.2: Tuning γ for optimization

We select a big gamma value $\gamma = 0.9$, as it learns quickly and converges steadily.

ϵ - epsilon

ϵ(epsilon) is the parameter used for trad-off between exploration and exploitation. Instead of just selecting the best learned Q-value action, we'll sometimes favor exploring the action further.

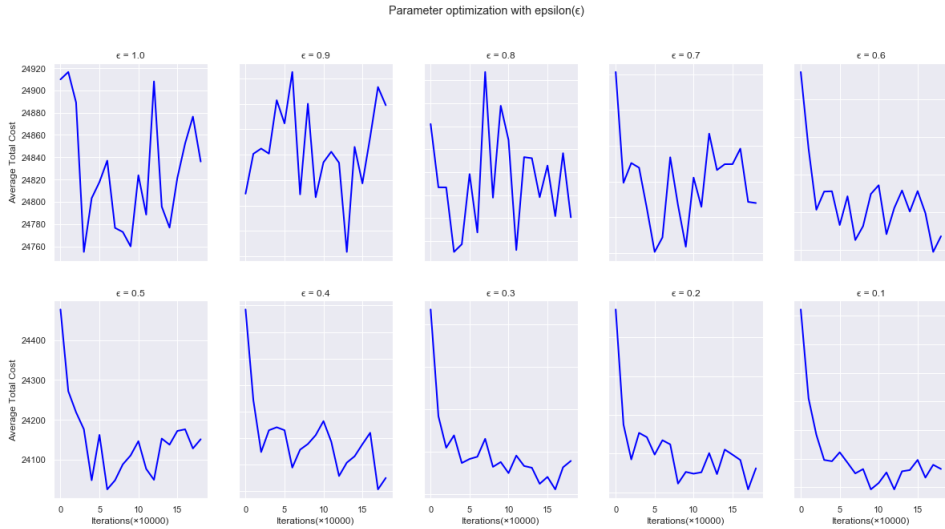

Figure 5.3: Tuning ϵ for optimization

In our case, from the result we see that the agent favors exploitation to exploration, therefore we select a smaller epsilon value $\epsilon = 0.1$.

5.2 Workflow and final result

We have created a visual that shows and summarises the entire workflow to produce the final output:

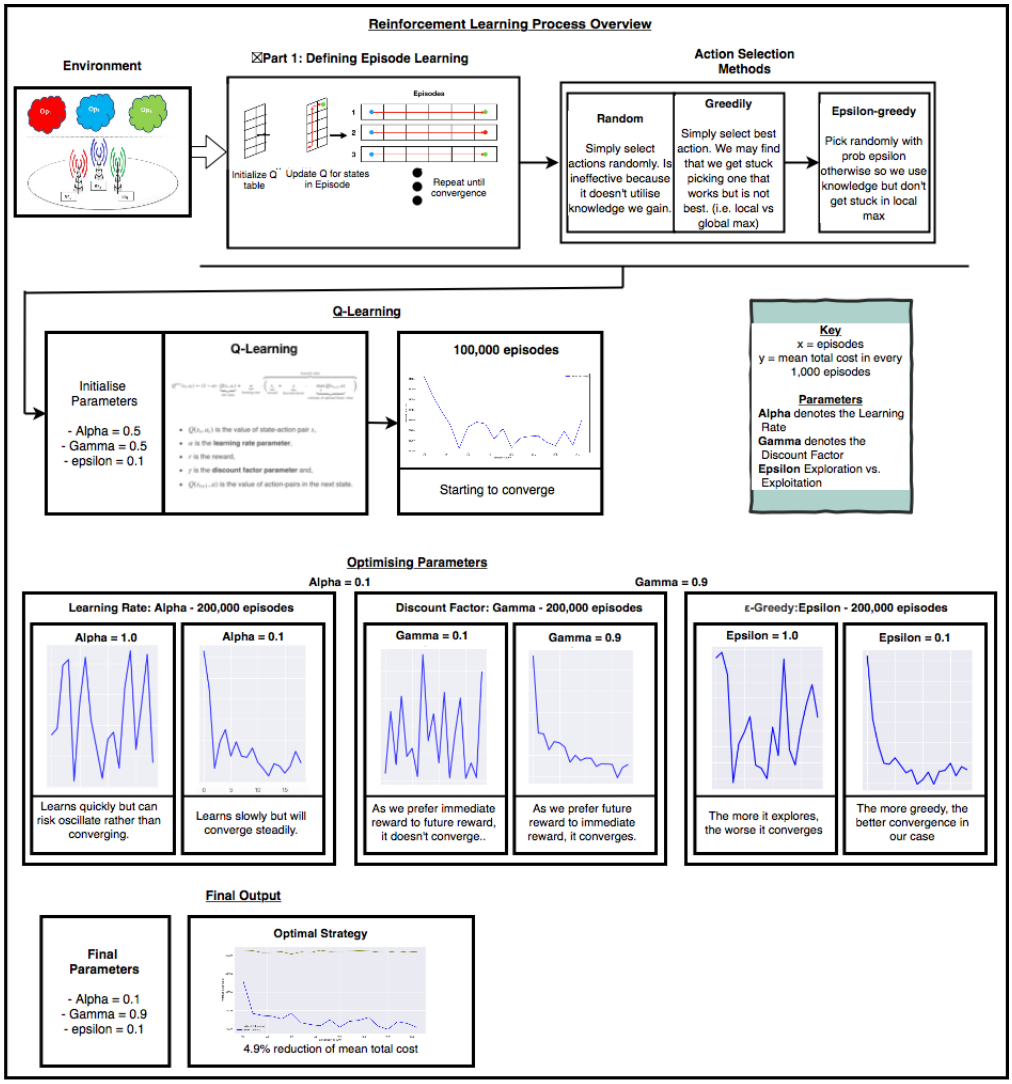

Figure 5.4: Reinforcement Learning Process Overview

We use the average total cost for each 10,000 iteration for result evaluation. And we see the average total cost declined around 4.9% after 200,000 iterations.

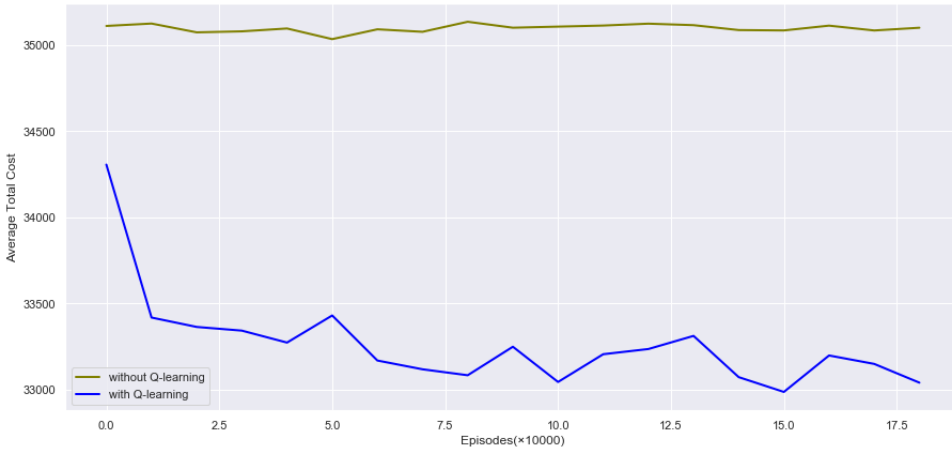

Figure 5.5: Average total cost at each 10000 iteration

Chapter 6
Summary and Recommendations for Further Work

This thesis's two research questions were: 1) What kinds of machine learning methods are appropriate for policy optimization in this research problem and why? 2) How to achieve policy optimization for the research problem with the selected machine learning methods?

To answer the first question, we analyzed different sub-branches of machine learning and explained why reinforcement leaning is believed to be appropriate for our specific research problems. To answer the second question, we evaluated and compared different reinforcement algorithms in respect to our research problems, designed a simulator with necessary interfaces for interacting with reinforcement learning agent, and implemented Q-learning in our simulator. From the Q-learning result, we see that the overall cost for a single operator has been reduced by 4.9% after 100,000 episodes. This gives us a much better decision making strategy compared with random decision making. Furthermore, this simulator could be used for both single operator scenario and multiple operator scenarios.

However, we could have implemented more reinforcement learning algorithms to find an optimal strategy for both single operator scenario and multiple operator scenario. But due to limited time, we haven't finished implementation of the other reinforcement algorithms such as DQN or policy gradient.

For future research, an introduction of the reinforcement learning fundamentals and a discrete event simulator with interfaces for reinforcement agents will probably be the most useful contribution of this study.

The future work could be a comparison for possible reinforcement learning algorithms for the single operator scenario problem, a thorough investigation of the multiple operator scenario and implementations of the possible reinforcement learning solutions for the multiple operator scenario.

References

[AJGW15] Prakriti Tiwari† Denis M. Becker‡ Andres J. Gonzalez, Bjarne E. Helvik† and Otto J. Wittner. Gearshift: Guaranteeing availability requirements in slas using hybrid fault tolerance. *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[JM15] M. I. Jordan and T. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science, 349(6245):255–260*, 2015.

[Li18] Yuxi Li. Deep reinforcement learning: An overview. 2018.

[RAH18] Universite de Montre Reyhane Askari Hemmat, Abdelhakim Hafid. Sla violation prediction in cloud computing: A machine learning perspective. *book:Computational Science and Its Applications*, 2018.

[SB] Richard S. Sutton and Andrew G. Barto.

[SZ19] Algorithm Engineer at Taboola Shaked Zychlinski. The complete reinforcement learning dictionary. *Medium*, 2019.

[w2] Openai gym.

[w3] Simpy.

[Zhu] Technical report.

[ZWL17] Danshi Wang Chuang Song Min Liu Jin Li Liqi Lou Zhilong Wang, Min Zhang and Zhuo Liu. Failure prediction using machine learning and time series in optical network. 25(16):18553–18565, Aug 2017.

Appendix A

Python code of the Simulator

```
"""Using␣machine␣learning␣for␣optimal␣SLA/SLO␣contract␣negotiation␣in␣5G"""

import random
import simpy
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import seaborn as sns

"""Initiate␣parameters"""
"""--------------------"""
"""Parameters␣for␣the␣simulator"""
NUM_RUNS = 200000
MTTF = 500000.0                    # Mean time to failure in minutes
MTTR_1 = 1                        #Mean time to recovry in minutes
MTTR_2 = 0.1
RT_MEAN = 0.02                    # Avg. recovering time in minutes
RT_SIGMA = 0.01                    # Sigma of recovering time
RECOVERY_TIME_2 = 100.0
OPER_COST_PER_MIN_A = 0.01         #Operation cost(thousand) per minute
OPER_COST_PER_MIN_B = 0.012
NUM_SYSTEMS = 1                   # Number of subsuppliers included in the SLA contract
WEEKS = 300                          # Simulation time in weeks
SIM_TIME = WEEKS * 7 * 24 * 60     # Simulation time in minutes
DOWNTIME_BUDGET = 0.2           # The allowed maximum accumulated down time in minutes
THRESHOLD = 30
PENALTY_PARAM = 25             #Penalty(thousand) per minute
TIMESTEP = 10
ACC_DT_SLOT = 100
ACC_DT_PER_SLOT = 0.1
MODE = 2

"""Hyperparameters␣for␣Q-learning"""
alpha = 0.1 # the learning rate
gamma = 1.0 # [0,1], a higher value(close to 1) captures the long-term effective award
epsilon = 0.1   # [0,1], whether to pick a random action or to exploit the already computed Q-va
                # the lower the value, the more it will exploit the Q-values

"""Simulator␣design"""
"""-----------------"""
def time_to_failure(mttf):
    """Return␣time␣until␣next␣failure."""
```

```
    return random.expovariate(1.0/mttf)

def time_to_recover(rt_mean):
    """Return time until recover"""
    return random.normalvariate(rt_mean, RT_SIGMA)

def get_time(minutes):
    """Change the time into a readable form"""
    week = int(minutes/(7*24*60))
    day = int((minutes-week*7*24*60)/(24*60))
    hour = int((minutes-week*7*24*60-day*24*60)/60)
    minute = minutes-week*7*24*60-day*24*60-hour*60
    return "%d:%d:%d:%d" % (week, day, hour, minute)

def get_penalty(acc_dt):
    """Return penalty for down-time overflow"""
    penalty = 0
    if acc_dt > DOWNTIME_BUDGET:
        overflow = acc_dt - DOWNTIME_BUDGET
        penalty = PENALTY_PARAM * overflow
    return penalty


def fail(env, mttr, num):
    """Fail and recover process"""
    #print('----------------------------------------')
    while True:
        time_to_fail = time_to_failure(MTTF)
        while time_to_fail < (SIM_TIME - env.now):
            fail = env.timeout(time_to_fail)
            yield fail
            fail_time = env.now
            #print('Failed in mode %d at' % num, get_time(fail_time))
            time_to_recv = time_to_recover(mttr)
            if time_to_recv < (SIM_TIME - env.now):
                recover = env.timeout(time_to_recv)
                yield recover
                #print('Recovered in mode %d at' % num, get_time(recover_time))
                return time_to_recv
            else:
                return env.now - fail_time
        else:
            return 0

def encode(timestep, acc_dt_slot):
        """ Create states of 10 * 100 for Q-table"""
        i = timestep
        i *= ACC_DT_SLOT
        i += acc_dt_slot
        return i

def decode(i):
        out = []
        out.append(i % ACC_DT_SLOT)
        i = i // ACC_DT_SLOT
        out.append(i)
        assert 0 <= i < TIMESTEP
        return reversed(out)


def update_Q_value(alpha, old_value, reward, gamma, next_max):
    """update Q-value according to bellman equation"""
```

```
    new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
    return new_value

def Q_state(time, acc_dt):
    state_2d = (int(time//(SIM_TIME/TIMESTEP)), int(acc_dt//ACC_DT_PER_SLOT))
    state = encode(state_2d[0], state_2d[1])
    return state


"""The␣system␣includes␣two␣simultaneous␣processes:␣Work␣Process␣and␣Step␣Process.
Q-learning␣algorithm␣is␣implements␣in␣the␣Step␣Process"""
"""--------------------------------------------------------------------------------"""
class System:
    def __init__(self, env, step_pipe, reward_pipe, q_table, total_cost):
        self.env = env
        self.q_table = q_table
        self.total_cost = total_cost
        self.work_proc = env.process(self.work(env, step_pipe, reward_pipe))
        self.step_proc = env.process(self.step(env, step_pipe, reward_pipe))
        #print("================================")
        #print('Start␣working␣at', get_time(env.now))

    "===Step␣Process==="
    def step(self, env, out_pipe, in_pipe):
        state = 0
        "Q-learning␣-␣epsilon␣greedy"
        while True:
            if random.uniform(0,1) < epsilon:
                action = np.random.randint(2)
                act = (env.now, action, state) # Explore action space
                #print("Random␣Action:␣", action)
            else:
                action = np.argmax(self.q_table[state])
                act = (env.now, action, state)
                #print("Q␣value␣Action:␣", action)
            out_pipe.put(act)

            yield env.timeout(SIM_TIME/TIMESTEP)

            info = yield in_pipe.get() ## 0-time, 1-acc_DT, 2-mode, 3-old state
            state = Q_state(info[0], info[1])
            old_state = info[3]
            penalty = get_penalty(info[1])
            cost = info[4]
            reward = - penalty - cost

            old_value = self.q_table[old_state, action]
            next_max = np.max(self.q_table[state])
            "Update␣Q␣value"
            new_value = update_Q_value(alpha, old_value, reward, gamma, next_max)
            self.q_table[old_state, action] = new_value

    """===Work␣Process==="""
    def work(self, env, in_pipe, out_pipe):
        info = [0, 0, 0, 0, 0] # info that will be transmitted to step process: 0-current time,
        down_time = 0
        cost = 0

        while True:
            # Get event for action pipe
            action = yield in_pipe.get() # 0-time, 1-action, 2-state
            old_state = action[2]
```

```
            if(action[1] == 0):
                """Work␣in␣mode␣1"""
                start_mode_A = env.now

                #print('Start␣mode␣A␣at␣%s' % get_time(start_mode_A))
                fail_in_mode_A = self.env.process(fail(env, MTTR_1, 1))
                down_time = yield fail_in_mode_A
                end_mode_A = env.now
                if (end_mode_A - start_mode_A > SIM_TIME/TIMESTEP):
                    time_in_mode_A = end_mode_A - start_mode_A
                else:
                    time_in_mode_A = SIM_TIME/TIMESTEP
                cost += OPER_COST_PER_MIN_A * time_in_mode_A
                info = (end_mode_A, info[1] + down_time, 1, old_state, cost)
                #print('Accumulated␣down␣time␣at␣time␣%s:␣%d' % (get_time(end_mode_A), info[1]))
                #print('----------------------------------------')


            if(action[1] == 1):
                """Work␣in␣mode␣2"""
                start_mode_B = env.now
                #print('Start␣mode␣B␣at␣%s' % get_time(start_mode_B))
                fail_in_mode_B = self.env.process(fail(env, MTTR_2, 2))
                down_time = yield fail_in_mode_B
                end_mode_B = env.now
                if (end_mode_B - start_mode_B > SIM_TIME/TIMESTEP):
                    time_in_mode_B = end_mode_B - start_mode_B
                else:
                    time_in_mode_B = SIM_TIME/TIMESTEP
                cost += OPER_COST_PER_MIN_B * time_in_mode_B
                info = (end_mode_B, info[1] + down_time, 2, old_state, cost)
                #print('Accumulated␣down␣time␣at␣time␣%s:␣%d' % (get_time(end_mode_B), info[1]))
                #print('----------------------------------------')

            out_pipe.put(info)

            self.total_cost = cost + get_penalty(info[1])


"""Running␣the␣system␣for␣one␣episode"""
"""---------------------------------"""
def simSystem(end_sim, q_table, total_cost):
    env = simpy.Environment()
    step_pipe = simpy.Store(env)
    reward_pipe = simpy.Store(env)

    system = System(env, step_pipe, reward_pipe, q_table, total_cost)
    env.run(until = end_sim)
    return system.q_table, system.total_cost


"""Initialize␣Q-table"""
"""------------------"""
state_num = TIMESTEP * ACC_DT_SLOT
q_table = np.zeros([state_num, 2])


"""Letting␣the␣reinforcement␣agent␣play␣and␣learn"""
"""----------------------------------------------"""
total_cost = 0
cost_sum = 0
```

```
cost_list = []  # store the total cost of each iteration so we can plot it later
aver_cost_list = []    # store the average of total cost of every 10000 iteration so we can plot

#Set the episodes the agent will run
for i in range(0,NUM_RUNS):
    q_table, total_cost = simSystem(SIM_TIME, q_table, total_cost)
    cost_sum += total_cost
    cost_list.append(total_cost)

    if i % (NUM_RUNS/20) == 0:
        #print("Episode␣%d" % i)
        print("-----------------------------------------")
        if i == 0:
            print("Cost␣at␣iteration␣%d:" % i, cost_sum)

        else:
            aver_cost = cost_sum/(NUM_RUNS/20)
            aver_cost_list.append(aver_cost)
            print("Average␣cost␣from␣iteration␣%d␣to␣%d:" % (i-(NUM_RUNS/20)+1, i), aver_cost)

        cost_sum = 0
        #print("=========================================")


"""Plot␣the␣mean␣cost␣for␣each␣10,000␣episodes␣after␣the␣agent␣running␣for␣200,000␣episodes"""
"""-------------------------------------------------------------------------------------------"""
plt.figure(figsize=(15,7))
plt.plot(range(len(aver_cost_list)),aver_cost_list, color='blue',
         linewidth=2, label='with␣Q-learning')
plt.xlabel('Episodes( 1000 )')
plt.ylabel('Average␣Total␣Cost')
plt.legend()
plt.show()
```