Deep Reinforcement Learning, LSTMs and Pointers for Downlink Scheduling in LTE

by

Aisha Robinson, **B.Sc.**

A thesis submitted to the

Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

Downlink scheduling in the LTE system is an open problem for which several heuristic solutions exist. Recently, there has been an increase in interest in applying machine learning to networking problems, including downlink scheduling. Improvements in Physical Layer capabilities have generated new resource-intensive use cases and continuously modifying existing heuristic solutions could result in the development of systems too complex to maintain over time. We propose a LSTM/Pointer Network-based downlink scheduler which aims to improve upon the current models which utilize feed forward neural networks. Our scheduler flexibly handles changing numbers of UEs via the use of a recurrent neural network. We formulate the downlink scheduling problem as a Markov Decision Process that integrates the channel quality indicator and the buffer size of each UE as the observation and solve it using a Deep Reinforcement Learning algorithm. Our experiments demonstrate that our approach results in a scheduler which generalised across changing number of UEs and resource blocks and performed within the range of traditional schedulers.

Acknowledgments

I would like to express my gratitude to:

- My supervisor Professor Thomas Kunz for allowing me the opportunity to study with him and for his patient direction and advice throughout this process.
- My parents Vivene and Radcliffe Robinson whose legacy I hope to build on and especially to my mother for her unending support.
- And, God for Life, Grace and Ability.

Table of Contents

List of Tables

List of Figures

List of Acronyms

3GPP	Third Generation Partnership Project
AMC	Adaptive Modulation and Coding
API	Application Programming Interface
BID	Binding Identity
BS	Base Station
CBR	Constant Bit Rate
CDMA	Code Division Multiple Access
CPU	Central Processing Unit
CQI	Channel Quality Indicator
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DDPG	Deep Deterministic Policy Gradients
DL	DownLink
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
eNodeB	Evolved Node B
EPC	Evolved Packet Core
E-UTRAN	Evolved UTRAN
FDMA	Frequency Division Multiple Access
GBR	Guaranteed Bit Rate
GRU	Gated Recurrent Unit

HARQ	Hybrid Automatic Repeat Request
HSPA	High Speed Packet Access
IP	Internet Protocol
LSTM	Long Short-Term Memory
LTE	Long Term Evolution
MAC	Medium Access Control
MDP	Markov Decision Process
ML	Machine Learning
MME	Mobility Management Entity
NN	Neural Network
OFDM	Orthogonal Frequency Division Multiplexing
PDCCH	Physical Downlink Control Channel
PDCP	Packet Data Convergence Protocol
PDU	Protocol Data Unit
PGW	Packet data network GateWay
PHY	Physical (Layer)
PRNG	Pseudo Random Number Generator
QAM	Quadrature Amplitude Modulation
QCI	QoS Class Identifier
QoS	Quality of Service
QPSK	Quadrature Phase Shift Keying
RAN	Radio Access Network
RB	Resource Block
RBG	Resource Block Group
RL	Reinforcement Learning
RLC	Radio Link Control
RNN	Recurrent Neural Network
RRM	Radio Resource Management
RRC	Radio Resource Control
SGW	Serving GateWay
SINR	Signal to Interference and Noise Ratio

SISO	Single-Input and Single-Output
TDMA	Time Division Multiple Access
TSP	Travelling Salesman Problem
TTI	Transmission Time Interval
UDP	User Datagram Protocol
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
URLLC	Ultra-Reliable Low-Latency Communications
UTRAN	UMTS Terrestrial Radio Access Network
VoIP	Voice Over IP
WCDMA	Wideband Code Division Multiple Access

Chapter 1

Introduction

1.1 Motivation

The LTE (Long Term Evolution) standard was designed by the Third Generation Partnership Project (3GPP), the international organization that also developed the widely used UMTS WCDMA/HSPA or 3G standards, to be a major improvement on previous generations of cellular networks. The 3GPP Specification [11] required that LTE provide increased peak data rates, increased cell edge bit-rates, lower latency, and improved spectral efficiency over previous generations. With LTE, the 3GPP introduced an all-IP (Internet Protocol) architecture, a change from the combined circuit and packet switching which existed in UMTS. This simplified the structure of the network and made accessible more of the available bandwidth to pave the way for supporting high quality data services such as video streaming, Voice over IP (VoIP) and online gaming, which require higher data rates and low latency. According to [8], the desired improvements in performance, which drove the development of LTE, could only be achieved by implementing new procedures at the physical and MAC (Medium Access Control) Layers, which enable the

exploitation of the wireless link capacity up to the Shannon limit[1]. This idea is seen in much of the functionality designed into the E-UTRAN (Evolved-Universal Terrestrial Radio Access Network) which is the component that manages the air interface of the LTE network.

Within E-UTRAN, the Radio Resource Management (RRM) block of the LTE architecture exploits a mix of advanced MAC and Physical Layer functions, like resource sharing, Channel Quality Indicator (CQI) reporting, link adaptation through Adaptive Modulation and Coding (AMC), and Hybrid Automatic Retransmission Requests (HARQ) [8]. Additionally, in the downlink direction, the LTE standard introduced Orthogonal Frequency Division Multiplexing (OFDM) in order to transmit more data at a lower per-bit cost. OFDM is a multi-carrier transmission scheme where the available bandwidth is divided into multiple narrow sub-carriers and data is transmitted on these sub-carriers in parallel. One thing the LTE standard did not specify however, is how resource allocation should be carried out. Even with all the improvements mentioned, there still exists the need for the design of effective and efficient resource allocation strategies to share the available (and limited) radio resources among the growing numbers of users in order to meet the ever changing performance requirements of advanced applications.

Many researchers have proposed novel algorithms for uplink/downlink scheduling in LTE. These algorithms are typically heuristics, and, due to the nature of the scheduling problem (a combinatorial problem), involve trade-offs in some metrics such as cell throughput, edge throughput, block error rate, fairness and delay to achieve others. In many cases, designing an optimal solution for resource sharing is computationally difficult due to the size of the state-space [2] and the fact that the states are only partially observable. Nonetheless, there are some algorithms which

[1]The Shannon limit of a communication channel refers to the max rate of error-free data that can theoretically be transferred over the channel given a particular noise level.

have been developed and are commonly used in the operation of LTE networks. These include Proportional Fair (PF), Round Robin (RR) and MaxCQI schedulers. These are very useful methods but present limitations and may be difficult to apply out of the box in an environment with more complex requirements. For this reason and in keeping with general trends in the network management space, there is quite a bit of exploration happening around the use of machine learning to automate network management tasks such as routing and resource allocation.

There are 3 main branches of machine learning: supervised, unsupervised and reinforcement learning. The need to adapt to changing network conditions makes reinforcement learning one of the most promising paths for applying machine learning to the resource allocation problem. As discussed in [5], supervised learning methods (for combinatorial problems) are essentially learning by imitation, whereas reinforcement learning (learning through experience) allows for the discovery of new policies and optimizations. Unsupervised learning is typically more suited to classification tasks and involves methods such as clustering, density estimation, and dimensionality reduction. Consequently, [5] in their review of learning methods for combinatorial optimization, chose not to discuss unsupervised learning on the basis that "it has received little attention in conjunction with combinatorial optimization and its immediate use seems difficult".

We will focus on reinforcement learning, which is a paradigm where an agent learns to take actions within an environment with the goal to maximise some cumulative reward. This approach has been used to learn to play games and in other strategy-based problems. It is a fit for the resource allocation problem in LTE for a few reasons. First, there is a centralised location within the LTE architecture (the eNodeB - Evolved Node B) from which the agent can collect information about the system and execute actions. Secondly, agents can continue

to learn and improve while the network is online. Thirdly, there are approaches which can be used in reinforcement learning that remove the need to have an accurate mathematical model of the network dynamics. Finally, because this is a current area of research interest, tools are being developed and released which allow greater exploration by creating interfaces between popular reinforcement learning tools and popular network research tools.

1.2 Contributions

This thesis explores the use of reinforcement learning to handle downlink scheduling in LTE networks. We design and train, using reinforcement learning, a downlink scheduler with a LSTM/Pointer Network structure. The scheduler is integrated into the ns-3 simulation environment which is used to evaluate its usefulness.

The proposed solution treats the UEs to be scheduled as an input sequence and leverages the memory capacity and flexibility of LSTM networks along with an Encoder-Decoder structure to produce a corresponding sequence of UEs as the scheduling decision. Further, the solution employs an attention mechanism to ensure only UEs in the input can be selected as output. These two design decision are made to allow the scheduler to handle changing numbers of UEs without the need to retrain the model. This is an improvement on existing machine learning-based schedulers which employ feed forward neural networks with fixed dimensions.

The simulation results confirm that our proposed solution is flexible. It manages changing numbers of UEs without further training and maintains performance in terms of throughput and fairness within in the range of traditional schedulers. It also provides results competitive to the more traditional ML-based schedulers that are trained for a specific number of UEs.

1.3 Organization

The thesis is organized as follows: In Chapter 2 we give an overview of the relevant factors for this study including: the functioning of an LTE network and the mechanics of scheduling, scheduling metrics, existing scheduling algorithms, and machine learning. In Chapter 3 we explore existing attempts to design schedulers using reinforcement learning and the open areas for development. In Chapters 4 and 5 we introduce our proposed solution and describe the design and implementation. And finally, in Chapter 6 we share the experiments we conducted, present the performance results, and the insights drawn from them.

Chapter 2

Background

2.1 LTE Structure

The LTE system is comprised of two parts, the core network, known as the Evolved Packet Core (EPC), and a Radio Access Network (RAN) known as the Evolved-Universal Terrestrial Radio Access Network (E-UTRAN). The EPC has three main components, the Mobility Management Entity (MME), the Serving Gateway (SGW) and the Packet data network Gateway (PGW). The MME handles user mobility, hand-offs, and the recording and paging of users. The SGW handles routing and forwarding of user's data packets as well as hand-over management, while the PGW provides the connection between the core network and other external networks such as IP networks. The RAN on the other hand is comprised of and supports two types of nodes, Base Station (BS) nodes known as eNodeB and the user equipment (UE) nodes. The eNodeB is the entity in charge of carrying out radio resource allocation processes.

Different methods can be used to share radio resources in a network. In the downlink direction LTE employs Orthogonal Frequency Division Multiple Access (OFDMA), while in the uplink direction, single channel orthogonal frequency division multiple access (SC-FDMA) is used. OFDMA is based on OFDM, which

combines TDMA (Time Division Multiple Access) and FDMA (Frequency Division Multiple Access) methods. OFDM is a multi-carrier transmission scheme where the available bandwidth is divided into multiple narrow sub-carriers and data is transmitted on these sub-carriers in parallel. OFDM was originally proposed to solve the problems of inter-symbol interference[1] and frequency selective fading.[2] Its ability to be used as a multiple access scheme is what is exploited in OFDMA. In each time interval, OFDMA can assign a fraction of the system bandwidth to a number of UEs, so several mobile users are allowed to receive data at the same time.

Resource allocation in LTE is concerned with deciding which UEs are assigned resources in each time slot. The resources are considered in three dimensions: time, frequency and power. Power allocation is outside the scope of this thesis and is not determined by the same factors that will be considered here. We will focus on downlink resource allocation in the time and frequency domains. In the time domain, channels are divided into 10ms frames, with each frame consisting of 10 subframes of 1ms. This subframe interval is referred to as the Transmission Time Interval (TTI). In the frequency domain, the total bandwidth available is divided into sub-channels of 180 kHz, further split into 12 equally spaced sub-carriers of 15 kHz. This organization is depicted in Figure 2.1. A single unit of radio resource covering 0.5 ms in the time domain and a 180 kHz sub-channel in the frequency domain is called a Resource Block (RB) and it is these units which are individually distributed to UEs at each TTI. All the RBs within the available bandwidth are collectively called the Resource Grid. The number of RBs contained in a resource grid and therefore available to be distributed to UEs, depends on the size of the

[1] If a wireless receiver starts to receive a new symbol, while it is still receiving a previous symbol on a close-by frequency the two symbols will overlap at the receiver, causing a problem known as inter-symbol interference (ISI) [9]

[2] Destructive interference caused by waves overlapping each other, resulting in a drop in received signal power [9]

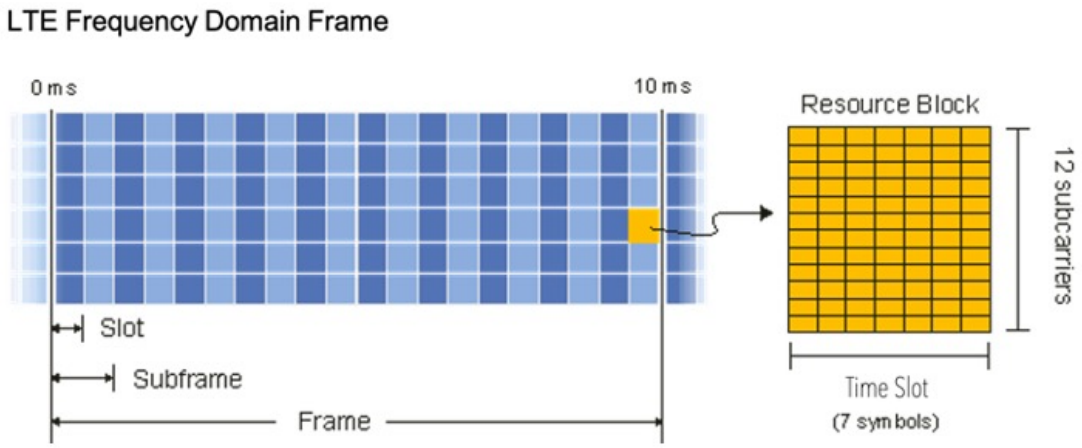


Figure 2.1: LTE Frame [37]

LTE system bandwidth, which can take on a set number of values ranging from 1.4 MHz to 20 MHz.

2.2 Scheduling in LTE

The task of resource allocation is not simply to fit all UE data into the available resource blocks. Resources need to be split up in ways that meet metrics important to the network operator as well as service requirements for the users based on their application type e.g. delivering a coherent video call. To achieve this, the MAC Layer at the eNodeB needs to consider information not only about the number of packets to be scheduled, but also information about their requirements, for example delivery time guarantees. Additionally, the eNodeB must consider the condition of the available radio resources as experienced by the UE. Wireless media is subject to a variety of disruptive factors including the weather, urban factors and user behavior. The effects of these factors on the wireless channel is captured in a measure called the channel quality. Channel quality at the UE in a given instance

can be evaluated through measurements reported by the UE.

In each TTI, the eNodeB sends reference signals which can be received by all UEs in the cell. Using this reference signal, each UE calculates a value called the signal to noise ratio (SNR) and maps it onto 15 channel quality indicator (CQI) values defined by the 3GPP, with 15 indicating the best possible conditions and degrading conditions as the value decreases. The UE reports this value to the eNodeB. Each CQI value corresponds to a modulation scheme (QPSK, 16 QAM, 64 QAM), a code rate, and an efficiency value, designed to allow the UE to have reasonably reliable transmission, taking into account the channel conditions it is currently experiencing. This CQI value is used by the scheduler in its allocation of resource blocks (RBs). If a UE is allocated a RB in a TTI, the Physical Downlink Control Channel (PDCCH) will be used to communicate the RBs allocated to it and the selected modulation scheme to be used. The UE then uses this information to receive and decode its information. CQI is not the sole piece of information used to make a scheduling decision. However, it is a fundamental one because it is used to determine the modulation scheme for the UE. At the eNodeB there is other information available to the scheduler which could be used in the decision making and most scheduling algorithms use a combination of available data. These include: the Head of Line delay, a measure of how long a packet has been waiting to be scheduled; QoS parameters such as the GBR vs non-GBR designation which informs the QCI (QoS Class Identifier) a ranking system for packet priorities i.e. (does this packet require reserved bandwidth or can it be treated as best effort); Queue Length, a measure of how much data a UE has pending transmission; Buffer status, which is similar to queue length but considered from the perspective of how much space there is left in the UE's buffer, as well as measures indicating the allocation history of the UE.

2.3 Scheduling Metrics

The choice of scheduling algorithm by a network provider is typically made on the basis of its ability to meet certain metrics depending on the services which are typically accessed within the network. Common metrics include average cell throughput, edge throughput, block error rate, fairness, delay and quality of service requirements. For the work in this thesis we will focus on throughput and fairness as metrics so these will be described in Subsections 2.3.1 and 2.3.2.

2.3.1 Average Cell Throughput

In general, throughput refers to a rate at which some entity passes through a system or process. In the context of communication networks, throughput refers to the amount of data delivered through the network and is typically measured in multiples of bits per second e.g. Mega bits per second (Mbps). Throughput can be measured at different levels, for instance the throughput (downlink) of a single UE for time period t can be given as:

$$R_i(t) = \frac{totBitsRcvd}{t} \tag{2.1}$$

i.e., the total number of bits received divided by the period of time being evaluated t e.g. 1s.

Network operators are more concerned with aggregate performance rather than single UE performance, so this idea can be built up into a more general metric. First the individual throughput for each UE is combined to give a total throughput.

$$R_{tot}(t) = \sum_{i=1}^{NumUEs} R_i(t) \tag{2.2}$$

Then, the total throughput is averaged over the total time period of interest

to represent the average throughput of the cell.

$$\bar{R}_{tot} = \frac{\sum_{t=1}^{T} R_{tot}(t)}{T} \tag{2.3}$$

2.3.2 Fairness

In general, fairness refers to the equality of resource allocation [17]. In our context (un)fairness can be defined as some UEs having significantly higher throughput than others or some UEs being starved of resources despite having a backlog of data to be received. Algorithms which quantify fairness seek to present this observation as an objective metric which is capable of being scaled to an arbitrary number of entities i.e., the same metric should be able to describe fairness among 3 UEs equally well as fairness among 3000 UEs. One such metric is Jain's fairness index (JFI) and it is computed as follows:

$$JFI = \frac{(\sum(x_i))^2}{n\sum(x_i)^2} \tag{2.4}$$

Where x_i denotes the i-th user's throughput and n is the total number of UEs. Jain's fairness index is a value in the range [0, 1] with one indicating perfect fairness, i.e., all users receive the same allocation. Values less than one indicate a degree of unfairness.

Fairness is an important metric in networks because the aim in most cases is to offer acceptable service to all users, including those with strict requirements and those without, as well as those experiencing ideal channel conditions and those in known difficult spots such as at the cell edge.

2.4 Scheduling Algorithms

The choice of scheduling algorithm has direct effects on the metrics described above and as such algorithms are designed to achieve different network preferences. For example, Max-CQI or Max Throughput algorithms schedule only the UEs with the best channel conditions and in so doing achieve very high throughput but considerable unfairness. There are however algorithms which do a better job at balancing these two metrics. The ones utilized for comparison in this thesis will be described in Subsections 2.4.1 to 2.4.3.

2.4.1 Round Robin

Round Robin (RR) is one of the simplest scheduling algorithms. It is channel-unaware, meaning it does not take into account the channel conditions being experienced by the UE. RR assigns resource blocks in a sequential and cyclical manner, with a UE being scheduled based on the last time it was scheduled among all the UEs. Along with being very simple to implement, Round Robin also performs reasonably well in terms of throughput and guarantees a high level of fairness when channel conditions are generally good. It loses efficiency in situations with mixed channel conditions such as the presence of UEs at the cell edge or some other area of the cell experiencing interference. These UEs will not be able to receive as many bytes in the time slot they are allocated compared to UEs experiencing good radio conditions. They would benefit from being scheduled more often to maintain their service quality. Channel aware algorithms aim to solve this problem.

2.4.2 Proportional Fair

The Proportional Fair algorithm is a channel aware algorithm which aims to balance throughput and fairness among all users [30]. The procedure tracks the

average throughput for each UE and performs a calculation as follows:

$$m_{i,k} = \frac{d_{i,k}(f)}{R_i(f)} \tag{2.5}$$

Where $R_i(f)$ is the the average throughput of user i in subframe f and $d_{i,k}(f)$ is an expression for the channel capacity, which describes the achievable throughput or instantaneous data rate of user i in resource block k and in f.

$$d_{i,k}(f) = log[1 + SNR_{i,k}(f)] \tag{2.6}$$

The calculations are stored as data points in a matrix M. The scheduler selects, for the current RB, the UE with the highest metric. The algorithm continues to assign RBs to the user with next highest metric until all RBs are assigned or all users have been served with RBs. If no data is available, a user is not scheduled. More formally, the scheduler selects UE m to be assigned the next RB using on the following equation:

$$m = \underset{i}{argmax} \frac{d_{i,k}(f)}{R_i(f)} \tag{2.7}$$

2.4.3 Max Throughput (Frequency Domain)

Maximum Throughput Scheduler is another channel aware algorithm which exploits the instantaneous channel conditions to maximize the cell throughput. It achieves this by scheduling the users with the most favorable channel conditions. The scheduler selects UE m to be assigned based on the metric:

$$m = argmax[d_{i,k}(f)] \tag{2.8}$$

where $d_{i,k}(f)$ is the instantaneous rate that can be achieved by user i (as

above in Proportional Fair) when assigned the RBGs in the fth subframe. This is calculated based on the CQI reported by the user. While Max Throughput schedulers achieve maximum cell throughput, they typically do so at the expense of fairness. If users are spread across the cell and are experiencing widely varying CQIs, the UEs with the worst channel conditions will be starved.

2.5 Automating Resource Scheduling

While the 'vanilla' Proportional Fair algorithm works largely as advertised, researchers have continued to adapt it to try to gain further improvements in performance for varying and sometimes niche scenarios. This practice illustrates the major draw-back to 'hand-developing' and upgrading scheduling algorithms: the rapid increase in use cases and the continuous development of the Physical Layer attributes may lead to the development complicated systems that will eventually be extremely difficult to maintain and improve [2]. This points to a need to have adaptable algorithms, not only for scheduling but in many areas of network management. This need has given rise to a movement which seeks to leverage machine learning to automate network management tasks, and ambitiously create 'self-driving' networks [12].

2.5.1 Machine Learning

Machine learning refers to algorithms which are able to make decisions or predictions from data in training examples [42]. Training examples can be explicitly given to the system such as when labeled images are provided to train an object recognition system, or the system can gain examples or experience by doing the actual task and receiving feedback on the outcome of its decisions. Systems which are given labeled training data are said to be utilizing a supervised learning method.

There are also unsupervised methods where the system seeks to discover similarities by explorations (without labels) and then there are reinforcement learning methods which we will utilize in this thesis.

2.5.2 Supervised and Unsupervised Learning

Supervised learning is a machine learning technique predicated on the availability of labeled examples. Supervised learning algorithms are given a labeled training data set and from that build a model of the system representing the observed relations between the inputs and outputs. After training, when a new input is given into the system, the model can be used to generate an expected output which should ideally be very close to the actual target output. The process of defining such a model is called 'learning' and is achieved by optimizing over a family of functions [5]. Some common supervised learning algorithms include k-nearest neighbor, decision tree, random forest, neural network, support vector machine, Bayes' theory, and hidden Markov models.

An unsupervised learning algorithm, on the other hand, is given a set of inputs without labels as its training data set. These algorithms aim to find patterns or structures in unlabeled data by clustering data points into different groups according to some similarity metric. Some common unsupervised learning algorithms include k-means clustering, DBSCAN clustering and principal component analysis.

Supervised learning can be successfully applied to aid in the solving of combinatorial problems. The labeling acts as an expert source of knowledge or oracle which demonstrates to the algorithm what the optimal decision would be in a given situation, until the algorithm learns a model capable of making similar decisions [5]. This is not useful in the context of downlink resource scheduling in LTE

because, given that we are in search of a solution for an always changing future, there is no source of expertise. Unsupervised learning is also not useful because it is more suited to classification tasks and is rarely considered for combinatorial problems [5]. With neither of these two algorithms being suitable we shift our attention to the third, reinforcement learning.

2.5.3 Reinforcement Learning

In reinforcement learning, a software agent aims to discover which actions lead to an optimal outcome by trial and error or by following some learned policy. A goal is set, for example to find a scheduling method which minimizes packet delay. The agent will then experiment with scheduling decisions, taking its feedback on how well it is meeting the goal from a reward function, and refine its approach to try to maximize the long term goal. This scenario illustrates the 3 components to any reinforcement learning method: 1. a policy, which informs the behavior of the actor, or no policy where the behavior is learned by trial and error, 2. a reward signal, which is used by the actor to judge its actions, and 3. a value function, which represents the overall goal of the optimization problem [34]. There is a fourth and optional component, a model of the environment. It is important to note that this option is not suitable for the downlink resource scheduling problem because of the stochastic nature of wireless communication, it is very difficult to create a useful model of the system and therefore we will see that model-free algorithms are better suited to this problem as they remove the need to have an accurate mathematical model of the network dynamics.

Reinforcement learning is a good solution for problems which can be formulated as a Markov Decision Process (MDP). MDPs provide a mathematical framework for describing a decision-making scenario where the outcomes are partly under the

control of the decision maker and partly random. A MDP is generally denoted by the triple (S, A, R),where S is either a complete or partial representation of the state of the system, A is the set of possible actions (discrete or continuous), and R is the set of rewards [34]. There are also variations on this model which add other factors like a discount factor to prioritize current rewards or possible future rewards.

The downlink scheduling problem is commonly framed by researchers as a Markov decision process. Several pieces of information can be combined to provide a partial description of the state of the system, these include CQI and length of the buffer queues for each UE among others. The action space is comprised of the RBs available to be scheduled and the UEs eligible to be scheduled. And finally, from the scheduling metrics deemed important for the network, reward functions can be developed to steer the agent towards maximization of the long term results. This formulation opens up the downlink resource allocation problem to the promise of reinforcement learning methods which could potentially replace the existing heuristic schedulers.

Three types of classical reinforcement learning (RL) methods are commonly used to solve MDPs: Q-learning, policy gradient, and actor-critic methods which are originally described in [19, 35, 36]. Some of these have been applied in the context of networking and will be explored in Chapter 3.

2.5.4 Deep Reinforcement Learning

The "deep" in deep reinforcement learning refers to multiple layers of artificial neural networks which are capable of computing high dimensional parametric functions. These neural networks can be used to supplement any machine learning algorithm, thus turning it into a deep learning algorithm. A neural network is a

computing system made up of a large number of simple processing units, which operate in parallel to learn experiential knowledge from historical data [14]. The layered structure is comprised of individual connections called neurons which, when combined with activation functions (non-linear functions applied to each connection), can be used to perform complex calculations and are widely used in many applications, such as pattern recognition. The value in deep learning is that it can be used to find important features of the data and to model them as high-level abstractions, thus avoiding manual description of a data structure as with feature engineering[3], by automatically learning features from the raw data [23]. For the scheduling problem that would mean learning features from the available data at the eNodeB and passing those to the reinforcement learning algorithm rather than taking the plain data features as input to the learning algorithm.

Deep reinforcement learning (DRL) is suitable for the task of automating downlink resource scheduling in LTE for a few reasons. One of these is that the eNodeB provides a centralised location within the E-UTRAN where an agent can collect a range of information about the network and execute the scheduling actions. With data being key to the functioning of any machine learning application, the ability to generate lots of training data is a fundamental consideration. According to [24], decisions made by scheduling systems are highly repetitive, which leads to them generating an abundance of training data for DRL algorithms. This is also true of scheduling in LTE. Decisions are made every TTI and there are several data points and metrics which can be considered for use as training data. Additionally, rather than being designed offline and then deployed on the network, because the network is continuously generating data, DRL agents can be trained and continue to learn and improve while the network is online. This is critical to the idea of

[3]Feature engineering in ML includes feature selection and extraction. It is used to reduce dimensionality in voluminous data and to identify discriminating features that reduce computational overhead and increase accuracy of ML models. [6]

flexibility: as the network changes, rather than attempting to tweak an existing algorithm and redeploy it to handle new demands or conditions, the same DRL agent is capable of continuously learning new ways of meeting the defined metrics.

2.5.5 Available Research Tools

Access to live large scale networks is a challenge for academic researchers because they are privately owned and costly to set up. Additionally, tools to integrate machine learning and control directly into networks are not yet fully developed and commercialized. Because the application of reinforcement learning to networking tasks is a current area of research interest, academic tools are being developed and released which allow researchers to work on a smaller scale. OpenAI's Gym,[4] for example, is a toolkit for developing reinforcement learning algorithms which can be used in conjunction with numerical computation libraries, such as TensorFlow,[5] which are used to build neural networks. One of the problems that Gym aims to solve is the lack of a large open source collection of environments on which reinforcement algorithms can be developed and tested. To that end they have provided an API which can be used to connect any environment to their toolkit and by extension to the computation libraries. ns-3[6], which is an open source network simulator that contains implementations of many networking protocols including LTE, is a popular network research tool and an ideal environment for use with Gym. It provides access to and control of many of the features of interest in networks and can generate data for a suite of analysis tools including visualisers. The authors of [13] recognized this fit and developed ns3gym; a tool for representing an ns-3 simulation as an environment in Gym, thus providing a compete sandbox for

[4]https://gym.openai.com
[5]https://www.tensorflow.org
[6]https://www.nsnam.org

RL research in networking. These tools will enable the exploration in this thesis.

Chapter 3

Literature Review

Techniques that leverage machine learning for downlink scheduling have become interesting as a means of replacing traditional heuristic schedulers, because continually updating 'handmade' schedulers could quickly create overly complex systems given the pace of change of network requirements. Deep reinforcement learning-based schedulers are even more interesting because they can learn how to schedule directly from the network data via trial and error. This means the scheduler can develop and change its own rules without human intervention and is therefore a promising way to handle large amounts of changing requirements.

In the following sections we will explore the space of deep reinforcement learning and the ways in which it has already been applied to downlink scheduling in LTE. We will then narrow our focus to policy gradients and their use in downlink scheduling and finally look at recurrent neural networks as a candidate for improving flexibility versus feed forward neural networks.

3.1 Applying Deep Reinforcement Learning to Resource Allocation and Scheduling

The authors of [24] applied the deep reinforcement learning (DRL) technique to a resource management problem. They designed and evaluated a multi-resource cluster scheduler which learned to optimize various objectives such as minimizing average job slowdown or completion time. Their solution performed comparably or better than standard heuristic for scheduling such as Shortest-Job-First. It even learned unique strategies such as "favoring short jobs over long jobs and keeping some resources free to service future arriving short jobs". Their approach used a policy gradient method and the REINFORCE algorithm with a fully connected feed forward network. Following on their successful effort, many others have explored DRL techniques to solve scheduling problems including scheduling problems within the networking domain.

3.1.1 DRL applied to Downlink Scheduling

In networking, situations where there are no ready to use algorithms are common targets for the use of the DRL method. These attempt to optimize a solution for a specific purpose. For example, [1], [20] and [21] each attempted to solve the problem of allocating bandwidth to network slices in 5G cellular networks. Their choice of method differed based on what they were trying to optimise for.

In [21], the number of arrived packets in each slice (for a specific time window) was used to represent the state of the environment, the action taken was the allocation of bandwidth (RBs) per slice and the reward was a weighted sum of two metrics describing the spectrum efficiency and the QoE requirement (wait time per flow). The authors employed the DQN (Deep Q Network - Q-learning

with a neural network) algorithm and carried out comparisons against a no slicing scenario (using Round Robin as MAC Scheduler), hard slicing and two demand-prediction based allocation algorithms, with slices containing video, VoIP and URLLC (Ultra-Reliable Low-Latency Communications) data. The DQN solution was better at satisfying the URLLC requirements when the resources were sufficiently constrained. When there were enough resources, "no slicing" worked better. The gain in QoE satisfaction came at the cost of spectrum efficiency and the authors demonstrated that by adjusting the weight of the QoE part of the reward metric the algorithm could learn a different scheduling policy which provided better spectral efficiency. The algorithm did not significantly improve the satisfaction of the video or VoIP slices.

In a slightly different approach [1] attempted to handle the case of a dynamic number of network slices. They identified 8 metrics to describe their system state, the actors' task was to select an amount of resource blocks to allocate to the slice and the reward was a combination of a metric which defined whether the UEs in the slice were able to satisfy their requirements and a metric which compared the amount of resource blocks used by the slice versus those assigned to it (i.e. spectrum efficiency). In order to handle a changing number of network slices, the authors proposed a solution where the actor would only allocate resources to one slice at a time and would be called as many times as needed based on the number of slices. The authors employed the DQN algorithm with the Ape-X [16] approach for using prioritized experience replay in a distributed manner (multiple actors with a single learner and replay experience buffer). Their method successfully handled changing slice numbers and performed better than the situation where no slicing was employed (all RBs scheduled to all UEs with a Proportional Fair MAC scheduler) and the situation with hard slicing (equal amounts of bandwidth

to each slice).

Koo et al. [20] attempted to solve a joint optimization problem; allocating CPU resources to the slice and allocation of bandwidth at the same time. This expanded the action space beyond what we have seen in the previous two solutions and introduces a new consideration. While the DQN algorithm is good at handling relatively small action spaces, when the action space becomes large or continuous, the algorithm is negatively impacted because it requires checking the value function for each possible action. The authors [20] defined the problem in a continuous rather than discrete manner, representing the state variables as distributions and then applied a policy gradient method based on the REINFORCE [35] algorithm seen in [24] (which was a discrete problem with a large action space) to solve it. The environment state was represented by the amount of received resource requests, the buffer level, and the last request arrival time. Two separate policies are used to output the resource allocation (CPU and Bandwidth) and the reward is a combination of processing delay and resource use cost. The method was compared to a fixed allocation of bandwidth per slice and performed significantly better in regard to the metric.

3.1.2 Policy Gradients

As mentioned, DQN as applied in the previous solutions is ill-suited for problems with a large number of actions or continuous actions because it evaluates the Q-function for every action in a state. The solutions reviewed above were allocating resources to fewer than 10 slices. Policy gradient methods and their deep learning variants however, can be easily applied to continuous action spaces as evidenced by the solution in [20] and have also been adapted for large discrete action spaces by mapping the algorithm's continuous output to discrete actions [10].

DDPG (Deep Deterministic Policy Gradients), which is the policy gradient and continuous method analogous to DQN, has been directly applied to attempt to solve the downlink scheduling problem in LTE (4G) networks. Wang et al. [41] defined the problem using instantaneous rate and average rate of transmission for each UE to represent the state of the system. They tested a scenario with a single resource block such that the action is to decide which UE is scheduled to occupy the only RBG in the scheduling period and a weighted sum of the total throughput and UE fairness is used as the reward. The DDPG agent was utilized in three different learning scenarios: 1. Directly interacting with the environment alone 2. Two agents interacting with the environment and learning from each other and 3. an agent interacting with the environment and learning from a Proportional Fair scheduler. All scenarios were compared to the Proportional Fair scheduler. All 3 agents approached the Proportional Fair standard with the one learning directly from it converging in the shortest amount of updates.

This result seemed to suggest that the best case scenario may be that a DRL agent will learn a policy similar to the Proportional Fair algorithm. However, [33] also deployed DDPG to schedule UEs and Resource blocks in a LTE Scenario. Their state representation was different from that in [41], they optimized for reducing bit delay by presenting the buffer state and CQI of each UE as the state. They provided a reward function which reflected a change in the buffer state of each UE from the previous time period to the current one. Their solution was compared to PF, Max weight and Max CQI schedulers and performed comparatively well and in some cases better than all 3 in terms of throughput, fairness, delay fairness and delay (measured per bit and using Little's Law [1]). This later result suggests that there is potentially more to gain over the Proportional Fair

[1] $W = L/\lambda$ where W is the average waiting time, L is the queue length and λ is the average arrival rate per unit time

scheduler with the use of DRL agent based schedulers.

3.2 Generalizing DRL solutions

Downlink scheduling is an example of a combinatorial optimization problem; of the UEs with data pending, given their channel conditions, what is the combination of them which, when scheduled in the available resource blocks in the current time slot, will optimize the chosen metric. In the previous works described, fully connected feed forward neural networks are utilized in the solutions. One of the problems observed in these works is that feed forward neural networks do not naturally lend themselves to varying input sizes. Papers which consider varying inputs used various methods to work around the problem. For example, [1], which noted that most papers that utilized machine learning for resource allocation to network slices only investigated a fix number of slices, designed their solution to handle changing numbers of slices by employing the RL agent to one slice at a time and calling it as many times as needed to handle all slices. The authors of [33], who considered a changing numbers of UEs, retrained their agent each time to handle 10, 20 and 30 UEs, each taking an increasing amount of time to train. All other papers in this survey did not address changing numbers of inputs i.e. they used a fixed number of slices or UEs etc. In DL scheduling, the UEs which are eligible for scheduling in a time slot do not constitute a fixed list, their number changes based on factors such as which UEs have data present for them or which UEs have left the cell or which UEs are already scheduled through HARQ. One common way of handling varying inputs in a fully connected feed forward network is to estimate the largest number of inputs expected and pad smaller inputs with zeros up to that limit. This creates a potential problem for generalized use of neural networks in real scenarios because the network will require retraining if conditions

shift beyond those limits, which is akin to the drawback of modifying heuristics.

3.2.1 Recurrent Neural Networks

There exists however, a class of neural networks called Recurrent Neural Networks (RNNs) designed specifically for handling varying input sizes and these have been explored for solving combinatorial problems. RNNs handle varying inputs by unrolling (reusing) a single neural network as many times as there are inputs taking each input one by one and passing it through the network, producing a 'hidden state' which is combined with the next input in the series to produce its hidden state and so on until the end of the input sequence is reached. This method of combination allows the network to 'remember' information from previous inputs by capturing important bits in the hidden state. In an RNN, outputs can be generated at each time step or only once at the end of the sequence. RNNs are commonly used in applications where the inputs can be interpreted as a variable length sequence such as in sentences for language translation or audio clips for music generation and have been formulated to produce classification type outputs such as assigning a genre to a music clip, prediction type outputs such as predicting the next number given the beginning of a series of numbers or outputs which generate a corresponding sequence given the input such as translating from English to French. The latter idea is referred to as the sequence-to-sequence (seq-to-seq) paradigm.

In addition to applications where sequence-to-sequence is used to perform mappings to outputs that are strictly sequences from inputs that are strictly sequences, there are examples of the technique being used to map to and from objects that are not necessarily sequences [38]. Examples are image captioning, which maps from an image to a sentence [40], or solving the traveling salesman problem (TSP)

by outputting an ordered tour given a set of random location coordinates [39]. The latter example is also a combinatorial problem. The authors in [39] aimed to expand 'vanilla' seq-to-seq architectures to not just handle varying input but also varying output dictionaries. In a typical seq-to-seq architecture, the size of the output dictionary needs to be fixed beforehand, as in the case of translation where there is a known fixed size corpus of words for any given language, which becomes the pool of options for the network to draw output sequence members from. This prevented the seq-to-seq solution from being applied directly to combinatorial problems where the size of the output dictionary varied with the input sequence, as is the case with downlink scheduling: a changing list of eligible UEs means a changing output dictionary. The authors of [39] addressed this issue by generating pointers to input elements as the output, and successfully used this mechanism to compute TSP tours. Inspired by the use of pointer networks described in [39], the authors in [4] again tackled the TSP, this time training their algorithm using reinforcement learning instead of supervised learning to remove the need to provide the optimal solutions as the target. They were also successful in producing near optimal tours. They further extended this RNN-RL solution to the knapsack problem, which involves fitting items into a knapsack given their weights and values, aiming to maximize the sum of value without exceeding the weight capacity of the knapsack. The algorithm generated pointers for items to include in the knapsack, stopping when the total weight of the items collected exceeded the weight capacity [4]. This is in contrast to the TSP problem, which produced an output sequence with length equal to the input length. This demonstrates that the seq-to-seq architecture can be used with constraints on the length of the output. In the case of DL scheduling, the output sequence should stop when the available RBs are utilized.

Structurally the sequence-to-sequence system takes the form of an Encoder-Decoder architecture. It is comprised of an Encoder RNN, which takes the inputs and produces a fixed size embedding vector, and a Decoder RNN (which is modified in the pointer network), which takes the embedding vector and produces the output sequence. In a regular seq-to-seq model, the output at each step is generated from the preset dictionary. In a pointer network the outputs are pointers generated by a mechanism known as Attention.

A vanilla RNN can produce its entire output sequence using the embedding vector after processing the whole input sequence. This presents a computational constraint as the size of the input sequence grows larger [39]. Attention [3] is a mechanism designed to solve this problem by augmenting the Encoder and Decoder RNNs with an additional, small, neural network which computes a vector indicating the context of the current observation in the series. For example, for a translation task, looking at the words immediately before and after a target word is more beneficial to deciding how to translate it than a word 10 words away from it. The pointing mechanism proposed in [39] makes use of this computed vector of 'relevancy' and uses it to produce a probability distribution over the elements in the input, thus identifying some as being a good next candidate at each step of the output sequence. [The authors also point out that the RNN network could learn to output the location (or index) of the target input point directly. However, this solution apparently degrades as sequences become longer.]

3.2.2 Conclusions

In this section we have explored several deep reinforcement learning methods which are promising solutions for downlink scheduling. While many approaches to downlink scheduling have relied on fully connected feed forward networks, we have identified a drawback to this network structure. Feed forward neural networks do not naturally lend themselves to varying input sizes. However, UEs are free to join and leave cells regularly and traffic demands or patterns fluctuate over time, so in a realistic setting, the candidate set of UEs to be scheduled varies over time. We observed that solutions to this problem have included retraining the model each time a change in input is needed (a solution which is only feasible in research context and not necessarily production networks) and padding the input values to allow room for change. We explore recurrent neural networks which is another class of machine learning structures better suited to handling changing inputs by treating them as sequences. We also identified a model (sequence-to-sequence) which treats not only the inputs as sequences but also the outputs. We conclude that this sequence-to-sequence structure appears to be a good choice for creating a generalized solution, capable of handling changing numbers of eligible UEs in downlink scheduling. UEs (and their state representations) can be fed in as a variable length sequence without attempting to estimate a largest possible input and a valid output sequence drawn only from the UEs present in the input can be produced by using a pointing mechanism.

Chapter 4

Solution Design

4.1 Overview

In our design, we are ultimately interested in a structure that will be capable of fulfilling two requirements:

1. Handle changing numbers of UEs
2. Ensure that only the UEs presented at the input are candidates for scheduling.

The first requirement directly addresses the drawback of feed forward neural networks identified in Chapter 3 where they do not naturally lend themselves to handling varying numbers of UEs when applied to downlink scheduling. The second requirement arises from the fact that we have identified the sequence-to-sequence model as a potential way to solve the variable input problem but in doing so face another challenge in that the output dictionary in seq-to-seq (list of possible output values) is typically predefined. In the case where UEs are leaving and joining cells the set of UEs which are eligible to be scheduled at any time is not fixed and therefore a pre-set dictionary would not be useful. In Chapter 3 we identified a promising modification for handling this second challenge, Pointer Networks. We propose that the use of this pointer network structure as an improvement on the

sequence-to-sequence structure is a potential solution to having a fixed dictionary of output UEs. The structure we design will be optimised by training with a deep reinforcement learning algorithm using data generated by a LTE simulation. This chapter explains in more detail what this structure will look like in the context of the downlink scheduling problem and Figure 4.1 provides a visual overview.

4.2 Inputs and Data Preparation

Recall that representing a Markov Decision Process requires the triple (S, A, R), where S is either a complete or partial representation of the state of the system, A is the set of possible actions, and R is the set of rewards. The starting point of the data flow is the state representation. We define the state as a combination of the CQIs and buffer sizes (also called queue length) of all UEs at the time of the scheduling decision. The values are collected prior to each scheduling decision in a data structure and then normalised to the range [-1, 1] before being passed into the machine learning network as shown in Figure 4.1. Normalization is employed in situations where input values are not in similar ranges (in our case 1-15 for CQI and 0 to some very large value decided by the amount of storage available for buffer size). During the process of back propagation, large mismatches in input ranges can cause the gradient to approach excessively large or zero values, these phenomena are called the exploding gradient and vanishing gradient problems [29]. They arise from the derivative calculations done during back propagation which are multiplied across the layers so if the values are too large they will get exponentially larger and if they are too small, exponentially smaller. The accumulation of large derivatives results in the model being unstable and incapable of effective learning. In the worst cases the gradients will cause overflow values (NaN). The accumulation

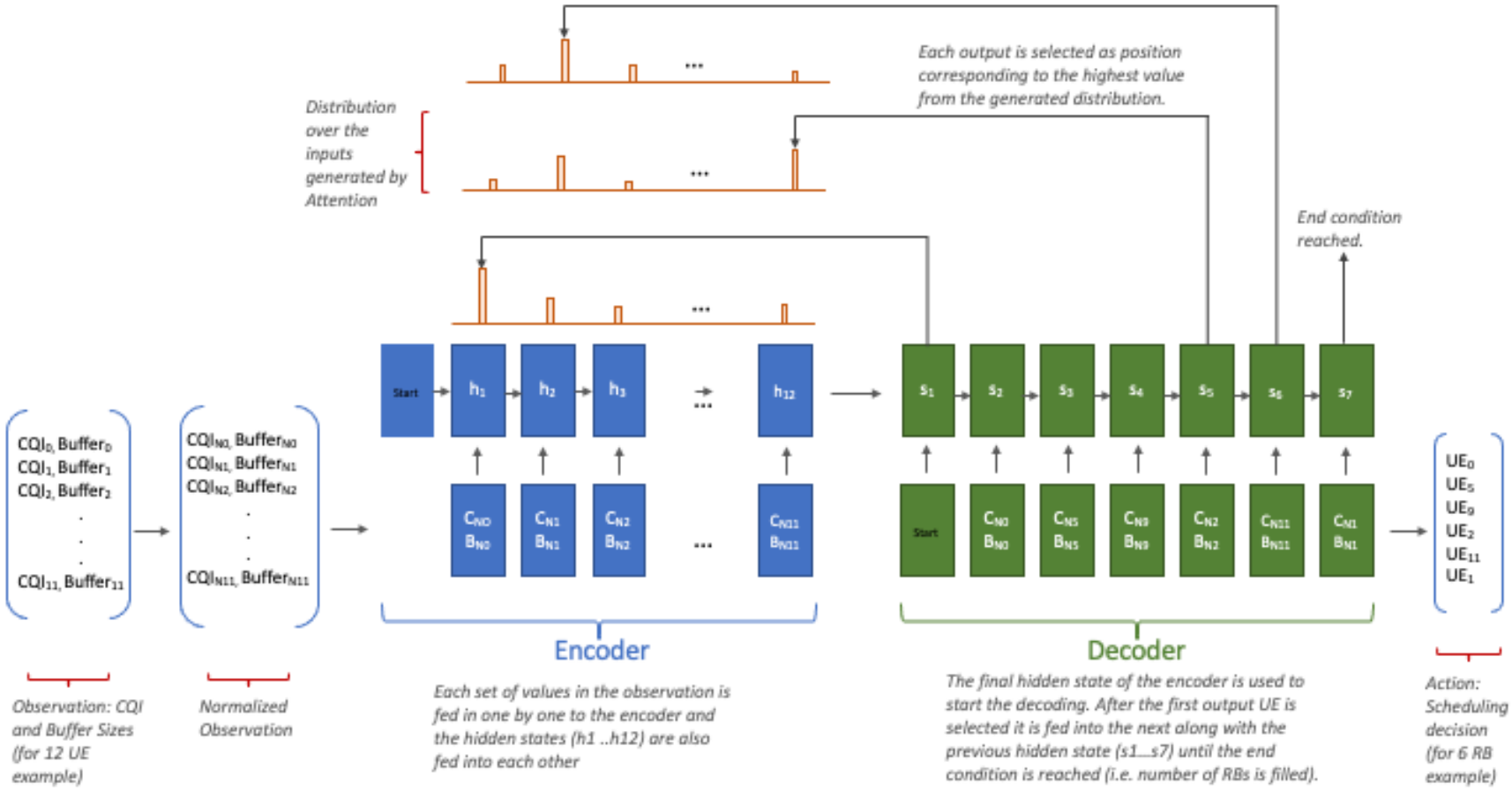


Figure 4.1: Using a Pointer Network to Generate Scheduling Decisions

of small gradients results in a model that is incapable of learning because, in the worst case, the gradient will be zero which will prevent the network from further training.

In general, the set of possible actions (outputs from the network) for this problem is a combination of the set of UEs passed in as the inputs and the number of scheduling slots available. E.g. if there are N UEs to be scheduled in the cell and M RBs available to be scheduled, the input to the network would be of length N and the action space would be of size: N^M. The total set of possible states (possible inputs) would be: $B^N \times C^{N \times M}$, where B is the set of possible buffer states (which is very large) and C is the number of possible CQI values (16). The size of B is what necessitates the reinforcement learning approach because the state space is now very large.

The reward function is defined as: $R_t = \sum_{i=1}^{N} r_t$ where:

$$r_t = \begin{cases} -2 & d < 0 \\ +2 & d > 0 \\ -1 & d, c, p = 0 \text{ and } UE_i \in A_{t-1} \\ +1 & d = 0 \; and \; p \neq 0 \end{cases} \tag{4.1}$$

for each UE in the cell and p = buffer size at t-1, c = buffer size at t, $d = p - c$, A = Allocated UEs in t-1. This function rewards reducing the buffer size of a UE while penalizing the scheduling of UEs with no data in the buffer or allowing a UE to go unscheduled for a long time.

4.3 Network - Encoder/Decoder LSTM Model

Encoder-Decoders are neural network approaches primarily developed for use with deep learning in Natural Language Processing. The Encoder-Decoder architecture has been adapted for use in combinatorial tasks and is made up of two basic components: an Encoder to encode input data into a context vector and a Decoder which decodes the context vector to the output sequence. The overall architecture of the solution follows the Encoder-Decoder model. UE states are sent first to the Encoder and the Decoder returns the scheduling decision as seen in Figure 4.1.

LSTM

Both the Encoder and Decoder are comprised of several recurrent units. A unit is a single neural network structure which, as described previously, is unrolled or reused for each element of the input sequence. Information is collected for each element and stored as a hidden state, which is passed to the unit in the following time step. This process is explained visually in Figure 4.2 where the unit s takes information in a single time step x (interpreted here as CQI and Buffer size for one UE in the list of UEs in the input) along with the hidden vector W from the previous time step as its inputs and produces an output O. W, U and V are additional parameters in the RNN.

LSTM (Long Short-Term Memory) cells are a modification to regular RNN units proposed in [15] to improve the memory capability of RNNs. The proposal in [15] was to alter the way that the hidden states are computed so they would hold more information for longer sequences. The block labelled s in Figure 4.2 is modified to function like the entire block between x_t and h_t in Figure 4.3, which demonstrates one version of an LSTM cell. Unlike the regular RNN structure in

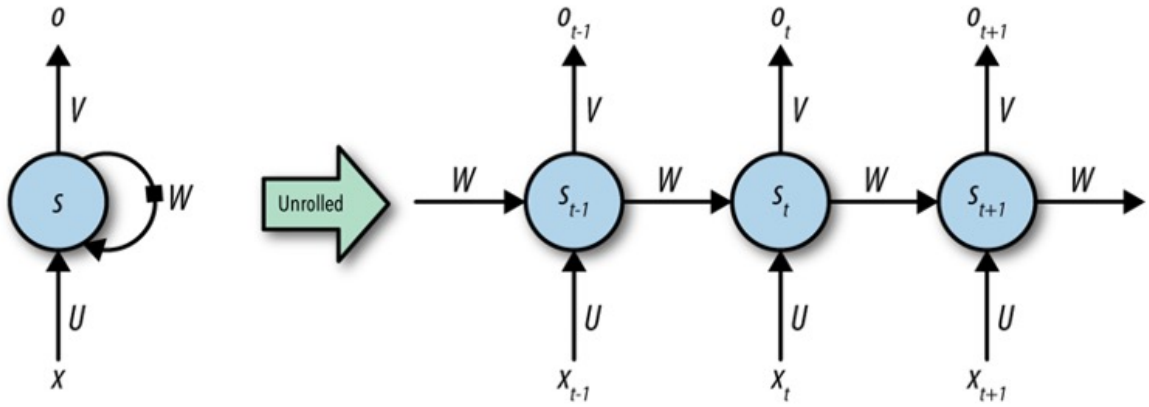


Figure 4.2: Typical Recurrent Neural Network [32]

Figure 4.2 which passes two inputs (the value at the current time step and the hidden state vector from the previous time step) through a single function to produce its output, the LSTM passes the inputs through several additional functions (called gates) before determining an output, allowing the LSTM to memorize data that is relevant and to forget information that is no longer needed when a new input arrives. In Figure 4.3, the cell state (C_t) represents the long-term memory of the system and stores the relevant information. The hidden state (h_t) is akin to working memory and is passed on to the subsequent recurrent unit so that memorized information along with the new input is used to produce the output at each step.

The LSTM functions by duplicating the input from the current time step 4 times, passing it to an 'input gate' which determines the amount of information from the current time step to store in the cell state. This information is concatenated with a copy of the original input passed through a sigmoid function (which constrains the value between zero and one) before being sent to the cell state. Duplicates are also passed to a 'forget gate' which decides what information to omit from the cell state, and an 'output gate' which acts like an attention mechanism

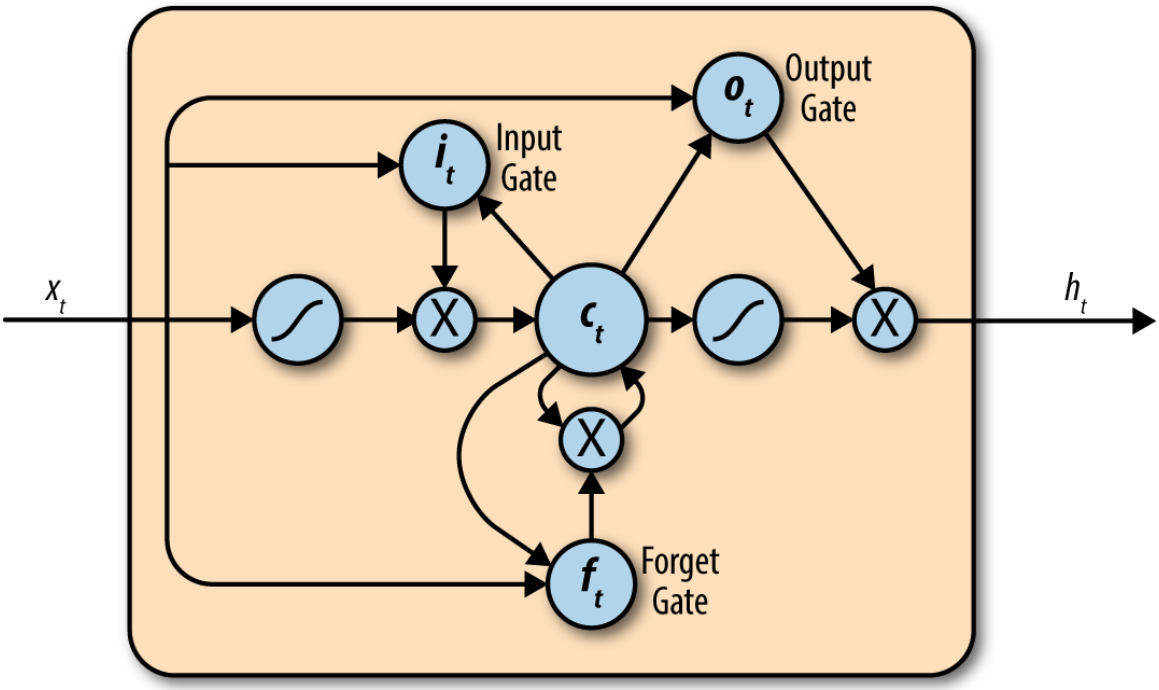


Figure 4.3: One Variation of an LSTM Cell [32]

to decide what parts of the data should be focused on. All these mechanisms are implemented as small neural networks which learn what to do via gradient decent.

4.3.1 Encoder

The Encoder in our solution is comprised of LSTM units and accepts input pairs of CQI and Buffer state at each step and outputs an embedding vector along with the final hidden state and final cell state vectors. These are passed to the Decoder where they will be used to produce the final scheduling decisions.

4.3.2 Decoder

The Decoder is implemented as a pointer network as described earlier. It takes the hidden state and cell state outputs of the Encoder and returns A (the number of RBs available to be scheduled) vectors of length N (the number of UEs in the cell)

with a softmax function (details below) applied across them. Note that the hidden and cell states are sequences, so the Decoder is also comprised of a set of LSTM units whose outputs are then modulated by the attention mechanism. From the softmax distribution produced, as shown in Figure 4.1, the UE with the highest probability is scheduled at each output step.

Attention

In the attention component of the model, the sequence of Encoder hidden states are used to produce the output. At each output step, an attention vector is computed as [39]:

$$u_j^i = v^T tanh(W_1 e_j + W_2 d_i) \quad j\epsilon(1, ..., n) \tag{4.2}$$

$$a_j^i = softmax(u_i^j) \quad j\epsilon(1, ..., n) \tag{4.3}$$

$$d_i^{'} = \sum_{j=1}^{n} a_i^j e_j \tag{4.4}$$

where e_j is the Encoder hidden state and d_i is the output of the Decoder LSTM given the Encoder state at the previous time step. The softmax function is calculated over the vector u_i (of length n interpreted in this case as the number of RBs) and the resulting vector is referred to as the attention "mask". Softmax is a function which transforms a vector into a set of values between zero and one. These values can be interpreted as probabilities, so the input transformed into the highest value can be interpreted as having the highest probability of being the correct choice given the others. It is computed via the following equation:

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{4.5}$$

where x is an input vector, and the softmax value is being computed for each input i. e is the standard exponential constant, K is the number of values in the input vector minus one ($x = (x_0, x_1, ...x_K)$). The other variables in Equation 4.2, i.e. $v, W1$, and $W2$, are learnable parameters of the model with v being a vector and $W1$ and $W2$ square matrices.

In the traditional seq-to-seq with attention model, $d_i^{'}$ and d_i are concatenated and used as the hidden states from which predictions $p(C_i|C_1, ..., C_{i-1}, P)$ (which element C_i of the output dictionary should come next in the sequence given the output dictionary $C_1, ..., C_{i-1}$ and P being the input sequence) are made and fed to the next output time step. This setup cannot be used as is in situations where the output dictionary size depends on the input as we have in downlink scheduling, where we want only the UEs in the input to be scheduled, because it is predicated on having a pre-provided output dictionary. Recall that the aim here is to handle varying numbers of input UEs, which implies that the UEs that comprise the output dictionary are also changing and therefore cannot be preset. This is where the pointer network modification of [39] is useful.

Pointer Network

The authors in [39] addressed the dictionary size limitation by re-purposing the attention mechanism to create pointers to input elements as the output.

$$u_j^i = v^T tanh(W_1 e_j + W_2 d_i) \quad j\epsilon(1, ..., n) \tag{4.6}$$

$$p(C_i|C_1, ..., C_{i-1}, P) = softmax(u^i) \tag{4.7}$$

Having the softmax output as the final stage means that for every step in the output (i.e. the number of RBs) the pointer network generates a probability distribution over all the UEs in the input and the UE having the highest probability

at that output time step is scheduled in that single RB slot.

4.4 Training

The networks were trained using a DDPG version without any exploration noise. DDPG uses function approximation to learn in continuous or large state and action spaces. It is implemented using four deep neural networks, denoted actor (θ^Q), critic (θ^μ), target-actor ($\theta^{Q'}$) and target-critic ($\theta^{\mu'}$). The target networks are copies of the actor and critic networks structurally. The weights from the actor and critic networks are copied to the target networks at a rate slower than the regular networks update. This helps to stabilise the training on the target networks because, while the regular networks may be affected by transient factors, these effects will not be copied over at the same magnitude. The result is that while the training rewards of the actor and critic networks may fluctuate wildly the target networks will train in a more gradual way.

The purpose of the actor is to take the states and return the actions while the critic outputs a score which describes the goodness of the state action pair. This goodness value is calculated as:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_i + 1|\theta^{\mu'})|\theta^{Q'}) \tag{4.8}$$

using the target network values.

After obtaining the critic value, a loss value is calculated between the current critic value and the previous critic value computed using the regular actor and critic networks:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{4.9}$$

This loss is used in updating the gradients of the critic network, the aim is to minimize the loss.

For updating the gradients of the actor network the following formula is used to calculate the loss:

$$\nabla\theta^{\mu} J \approx \frac{1}{N} \sum_{i} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla\theta^{\mu} \mu(s|\theta^{\mu})|_{s_i} \tag{4.10}$$

where N is the size of the set of samples being used to update the gradient.

The weights for the copied target-actor and target-critic networks are updated more slowly than the main networks. The update rate is determined by a parameter τ which is a value < 1. The updates are performed according to the equations:

$$\theta^{Q'} \longleftarrow \tau\theta^{Q} + (1 - \tau)\theta^{Q'} \tag{4.11}$$

$$\theta^{\mu'} \longleftarrow \tau\theta^{\mu} + (1 - \tau)\theta^{\mu'} \tag{4.12}$$

Additionally, DDPG makes use of a replay buffer, which stores past state, action, reward tuples from which small random samples ('mini-batch') are drawn at each time step and used in the gradient calculation which updates the weights. This allows the algorithm to learn from uncorrelated past events. Lastly, DDPG encourages exploration of the action space by adding noise sampled from a noisy process to the actor policy. DRL methods including DDPG often try to encourage exploratory behavior through injecting noise in the action space [31]. DDPG as described in [22] uses Ornstein-Uhlenbeck noise to encourage action space exploration. However, we found that the method caused our training rewards to oscillate over a large range rather than approach a specific value. Consequently, we chose to omit the noise in our solution. The algorithm is summarized in Algorithm 1.

Algorithm 1: Pseudocode for DDPG algorithm without noise

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.

Initialize target network Q' and μ' with weights $\theta^{Q'} \longleftarrow \theta^Q, \theta^{\mu'} \longleftarrow \theta^\mu$

Initialize replay buffer R

for $episode = 1$ *to* M **do**

  Receive initial observation state s_1

  for $t = 1$ *to* T **do**

    Select action $a_t = \mu(s_t|\theta^\mu))$ according to the current policy

    Execute action a_t and observe reward r_t and observe new state s_{t+1}

    Store transition (s_t, a_t, r_t, s_{t+1}) in R

    Sample a random minibatch of N transitions (s_t, a_t, r_t, s_{t+1}) from R

    Set $y_i = r_i + \gamma Q'((s_{i+1}, \mu'(s_i + 1|\theta^{\mu'})|\theta^{Q'})$

    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

    Update the actor policy using the sampled policy gradient:

$$\nabla\theta^\mu J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla\theta^\mu \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \longleftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \longleftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for

end for

Chapter 5

Implementation

5.1 System Overview

The major components required to implement a RL solution are the environment, which should be capable of reporting its state and implementing any actions decided by the agent, and the learning agent, which should be capable of accepting an environment's states and providing corresponding actions which optimize some reward function. In deciding on components for implementing our own set-up, it was important to use tools that were already regarded as credible in the research community. The system as shown in Figure 5.1 comprises ns-3[1]: an open-source discrete-event network simulator for networking systems, targeted primarily for research and educational use, ns3gym [13]: a tool for representing an ns-3 simulation as an environment in the widely accepted toolkit for RL research: OpenAI Gym [7] and a learning agent implemented with the Python numerical libraries Keras[2] and Tensorflow[3]. Data flows from the ns-3 simulator through the ns3gym middleware which consists of two parts: a) the environment gateway (implemented as part of the ns-3 simulation environment) and b) the environment proxy (inherited from

[1]https//www.nsnam.org/
[2]https://keras.io/
[3]https://www.tensorflow.org/

the OpenAI Gym API). They are responsible for receiving the state information from the simulator, encoding it in the numerical formats accepted by learning agents and doing the reverse for actions received from the agent, converting them back to function calls which correspond to the simulation.

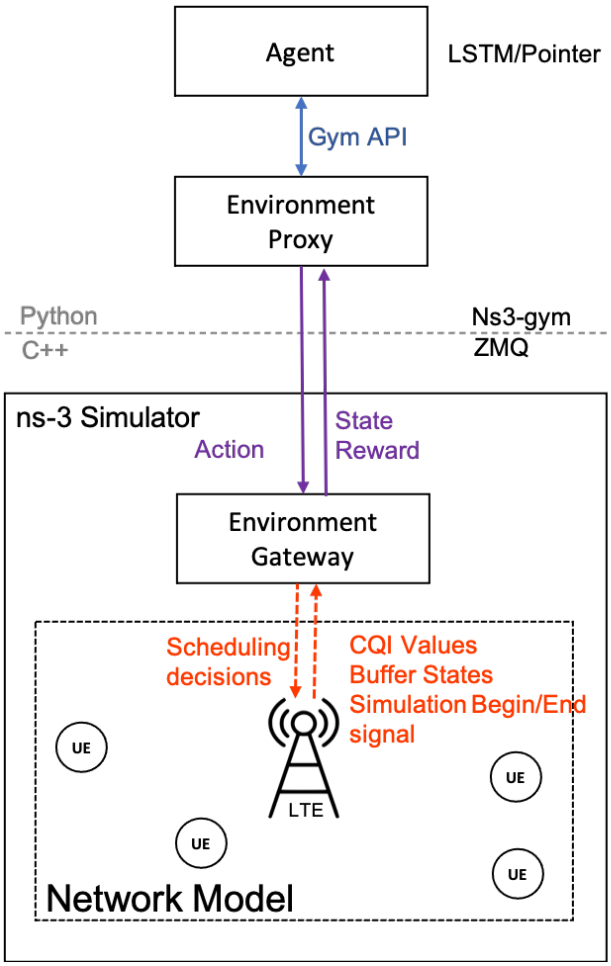


Figure 5.1: System Diagram.

5.2 The Simulation Environment

ns-3 is a discrete-event network simulator for Internet systems. It is targeted primarily for research and educational use and is free and open source under the GNU GPLv2 license. ns-3 provides models of how packet data networks work and perform, as well as a simulation engine for conducting experiments. The tool is designed as a set of libraries that can be combined together as well as with other external software libraries. Scripts in ns-3, used for setting up and running simulations are written in C++ or Python while the models themselves are written in C++. ns-3 is primarily used on Linux or macOS systems.

There are a few useful constructs in ns-3 for experimenters, one of which is the tracing system. The tracing system is built on the idea of having independent tracing sources and tracing sinks, and a uniform mechanism for connecting sources to sinks [27]. Trace sources are entities that can signal events that happen in a simulation. For example, a trace source could indicate when a packet is received by a device and provide access to the packet contents. Trace sources are connected to other pieces of code that are capable of interpreting and using the information gathered, these are called trace sinks. For example, one could create a trace sink that could print out parts of a received packet. ns-3 allows users to define new types of sinks and sources, through scripts or by editing the simulation core.

For the purposes of this investigation, the simulation implemented in ns-3 is an LTE network model where a base station (BS) allocates resource blocks to connected UEs. The UEs are receiving data packets at regular intervals and the BS allocates resources in each transmission time interval (TTI). The BS is configured with a MAC scheduler where the downlink scheduling sequence collects the buffer status and CQI value of all UEs and passes this information to the environment gateway (also implemented in ns-3) to be transformed and used to represent the

state of the environment. The scheduler also receives from the gateway (which is connected to the learning agent) the scheduling decision corresponding to the sent state information and based on this decision performs the necessary sequence for assigning RBs to the UEs in the simulator.

5.2.1 Tracing CQI and Buffer Data, Scheduling Decisions

As mentioned earlier, the ns-3 tracing system is the mechanism used to retrieve information from a simulation. The pieces of information which needed to be accessed in this investigation were the CQI and Buffer data. As mentioned in Chapter 2, CQI is reported by each UE to the eNodeB and buffer data is maintained for each UE at the eNodeB. This data was needed each time a scheduling decision was to be made. Scheduling decisions are made by the MAC scheduler which is represented by a module in the ns-3 simulation. ns-3 already contains implementations for some of the common scheduling algorithms. Each is modeled on a generic mac scheduler API provided by the core and so new schedulers can be implemented using this API. We created a new scheduler which would collect the CQI and buffer data, request a decision from the machine learning model and implement the received scheduling decision following the standard protocols. To explain where the source and sinks for the CQI and buffer data were placed, we need to look at the path that data takes within the eNodeB.

Buffer and CQI Data Path

After an IP packet generated by some host device on the Internet is successfully routed to the eNodeB to which the destination UE is attached, it passes through a protocol stack to prepare the packet for delivery. This data plane protocol stack consists of the following layers (in descending order): PDCP (Packet Data

Convergence Protocol), RLC (Radio Link Control), and MAC (Medium Access Control). The packet will first arrive at the EpcEnbApplication, which performs tasks such as stripping headers, attaching a bearer ID to the packet and then forwarding it to the LteEnbNetDevice. The LteEnbNetDevice determines, based on the BID (Binding Identity), the Radio Bearer instance (controlled by the Radio resource control protocol, RRC) and the corresponding PDCP and RLC protocol instances which will be used to forward the packet to the destination UE via the LTE radio interface. In the simulation environment, each of these layers and the services which support them are organized as separate modules. Figure 5.2 shows how these planes are connected as well as the services which support them and their organization in ns-3.

In the downlink direction, the PDCP is responsible for header decompression for the data packets. The outputs of a protocol layer is generally referred to as a PDU (Protocol Data Unit) so the output from the PDCP Layer is referred to as PDCP PDU. When the PDCP is ready to transmit a PDU it first calls a service in the RLC. The RLC entity processes this notification according to the data transfer procedures for its type. [4] The following procedures are performed:

1. The data received by the PDCP is placed in the Transmission Buffer
2. The size of the buffers are computed, and a service of the eNodeB MAC entity is called in order to notify it of the sizes of the buffers.
3. The eNodeB MAC entity then updates the buffer status in the MAC scheduler.
4. When the MAC scheduler decides that data can be sent, the MAC entity notifies the RLC, which will create a new PDU from the data in the Transmission Buffer, reformatting it to fit the size required by the MAC Layer's transport block, which is dependent on the system bandwidth. It will then

[4]The RLC entity as specified by 3GPP can take three different formats: Transparent Mode (TM), Unacknowledged Mode (UM) and Acknowledged Mode (AM).

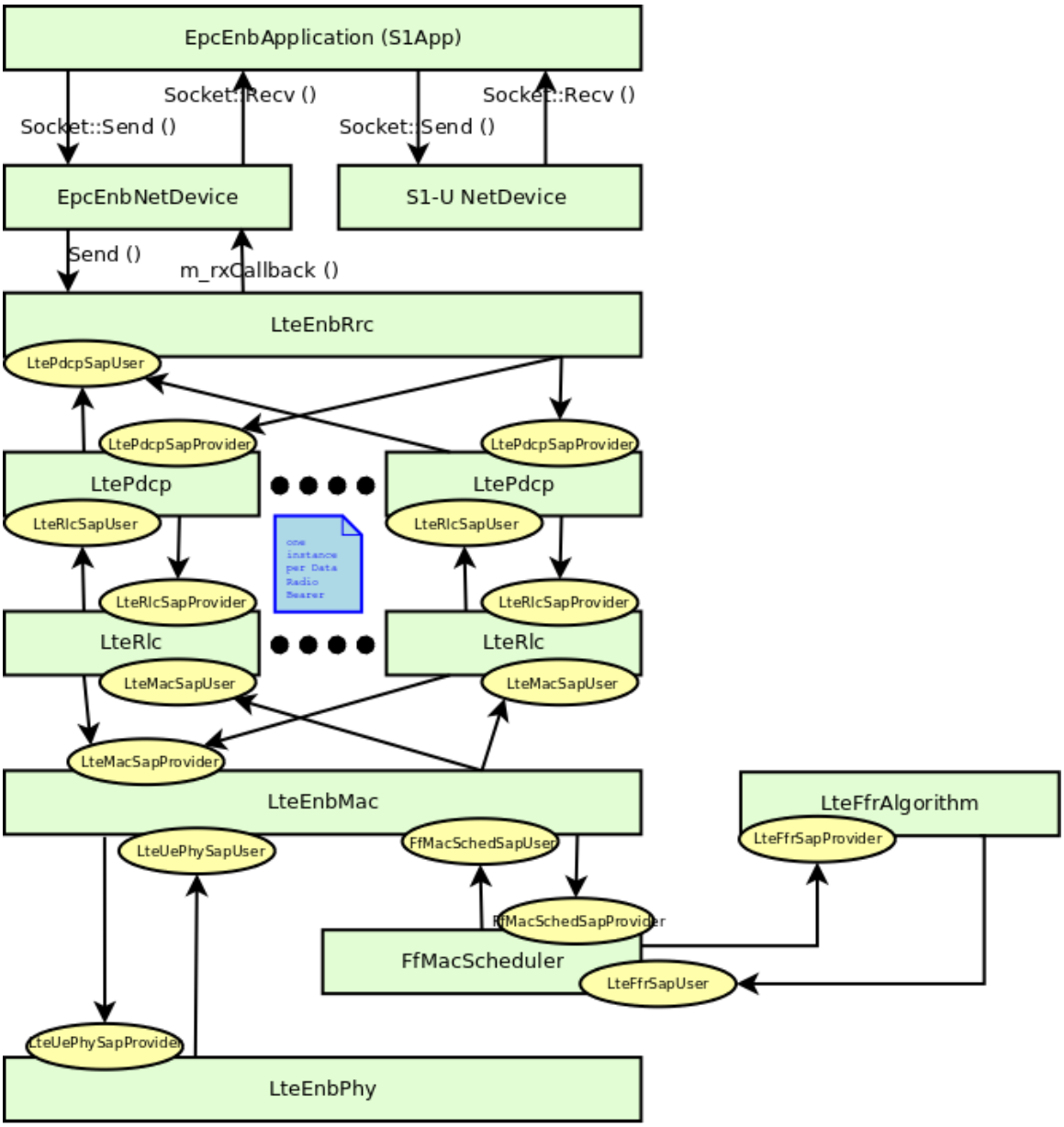


Figure 5.2: LTE Radio Protocol Stack Architecture, eNodeB, Data Plane [26]

move the PDU from the transmission buffer to the transmitted PDUs buffer (in case it needs to be re-transmitted) and send the PDU to the MAC entity.

This process is illustrated in the sequence diagram in Figure 5.3.

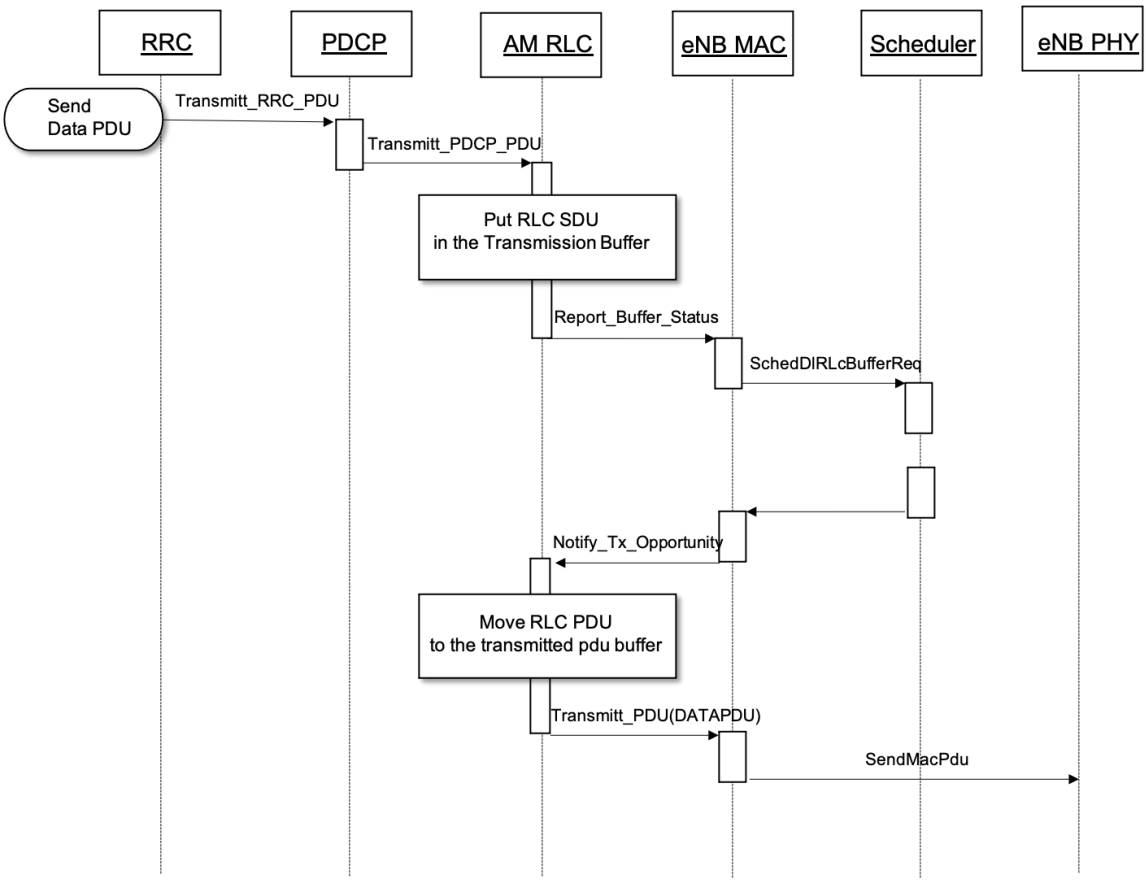


Figure 5.3: Data PDU Transmission in the Downlink [26]

The CQI feedback we are interested in capturing is the inband CQI. This is a value calculated to represent each UE's channel condition. It is reported by first obtaining a SINR measurement and then passing this measurement to the Adaptive Modulation and Coding module, which will map it to the CQI index. Each MAC scheduler is responsible for handling CQI values when needed and so this data is received directly by the scheduler via a call to the MAC and PHY Layers.

Given that both values of interest were accessible via the scheduler module, the trace source for the observation of both values was placed there. When the information corresponding to the eligible UE is available, a tuple is created containing the CQI data, the buffer data, the number of data flows, and the number of resource blocks available. This source is accessible by the sink located in the environment gateway. In the reverse direction, the source for the action (scheduling decisions) is updated when new information is available in the environment gateway and is accessible by the sink in the scheduler.

5.3 The Reinforcement Learning Environment

OpenAI Gym is "a toolkit for developing and comparing reinforcement learning algorithms" [7]. The primary feature of the tool is that its developers maintain a collection of benchmark problems (environments) which researchers can use via a common interface to test and compare against their own RL algorithms. Important to our investigation is the fact that OpenAI Gym makes this interface available via an API so that custom environments can be developed and used. Any environment can be integrated into Gym as long as all the observations, actions, and rewards can be represented as numerical values. Gym then provides the interface between the environment and any numerical computation library and is developed in Python.

The interface provided by Gym manages the processes required to generate observations and actions from the environment and the reinforcement agent (defined by the developer, Gym has no prescriptions for this). For example, Gym has functions to indicate when to render and reset an environment, to run the environment only for the number of time steps specified, and to pass the actions and observations between the environment and agent. In our example this would mean starting an ns-3 simulation as for as many runs as specified, collecting the

packaged CQI and buffer data, and returning the scheduling decision made by the agent to the ns-3 simulation.

Integrating any non-trivial environment to Gym requires some software in the middle to handle the data collection and management of the environment's controls. For example, while the agent is performing calculations in order to return the scheduling decisions, the ns-3 simulation would need to be paused. The OpenAI Gym framework can indicate when calculations would start or end but it does not have the controls to pause the simulation itself. Also, recall that ns-3 is C++ based and OpenAI Gym is Python based so there is a need for an integrator. In our setup, that role is filled by the open-sourced ns3gym [13] toolkit.

ns3gym [13] consists of two software components, the Environment Gateway, written in C++, and the Environment Proxy, written in Python. The gateway resides in the ns-3 simulator and requires the simulation script to implement a set of callback functions for receiving and sending the relevant information, while the Environment Proxy is a Python class which inherits from the generic Gym environment provided by the OpenAI Gym API [13]. The communication between these two components is handled with ZMQ sockets and using the Protocol Buffers library. ZMQ is an asynchronous messaging library and Protocol Buffers is a cross platform library for data serialization.

5.4 Agent Implementation with Keras and Tensorflow

At each scheduling decision, the collected and formatted information flows from the scheduler in the simulation through the ns3gym middleware and to the learning agent. The agent, during training, will be attempting to learn an action model

which maximizes the long term reward by trial and error. This is done by a series of mathematical computations involving neural networks, computing high dimensional (on the order of several thousand nodes) parametric functions. Software libraries have been developed to quickly set up and manage these computations. Tensorflow is one of the popular and powerful open source projects written in Python, however it is still cognitively demanding to use as a developer. In response to the need to further simplify the process of setting up a machine learning project, Keras was created as a high-level interface that can be used with Tensorflow as the backend. It allows users to abstract away some of the finer details of tensor algebra and optimizing computations and thus allow faster prototyping etc.

In this investigation we took advantage of these Keras wrapper for Tensorflow to simplify the process. As mentioned in Chapter 4, the training algorithm used was a modification of DDPG and consisted of two actor-critic networks with a replay buffer. The base of the structure was modeled on the sample network provided by the Keras documentation [18] to solve the pendulum benchmark environment in OpenAI Gym and then modified as described in Chapter 4 for the pointer network structure.

Figures 5.4 and 5.5 show the Keras model summary of the networks. For each model, the figure depicts its layers, output shape, number of parameters (i.e. connections in the network that will be updated by gradient decent during training) and which layers it is connected to. For example, the decoder layer in the actor model produces an output of size (None, 6, None) translating to (any batch size, 6: the number of resource blocks available in that example, any number of UEs). Defining a size parameter as none means the size is not predetermined. Further, the decoder layer has 164480 trainable parameters and is connected to the 3 outputs from the encoder (recall that the encoder produces an embedding

vector, a hidden state and a cell state).

Model: "Actor Model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 12, 2)]	0	
encoder (LSTM)	[(None, 12, 128), (None, 128), (None, 128)]	67072	input_1[0][0]
decoder (PointerLSTM)	(None, 6, None)	164480	encoder[0][0] encoder[0][1] encoder[0][2]

Total params: 231,552
Trainable params: 231,552
Non-trainable params: 0

None

Figure 5.4: Actor Model Summary

The actor network is comprised of the input layer, the Encoder LSTM and the Decoder LSTM with the pointer network structure. The critic network (designed to produce a single value) takes two inputs: a state representation and the corresponding action generated by the actor model. Both are drawn from the replay buffer. These two values are passed through several neural network layers and then concatenated into a single vector which is again filtered through NN layers to produce the single critic value. A mean squared error loss is computed for both actor and critic networks at each training step and the is used to update the network weights via back propagation. The target-actor and target-critic network weights are updated periodically by copying a fraction of the actor and critic networks weights.

Model: "Critic Model"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, None, 12)]	0	
critic-action (LSTM)	[(None, None, 32), (None, 32), (None, 32)]	5760	input_3[0][0]
input_2 (InputLayer)	[(None, 12, 2)]	0	
dense_3 (Dense)	(None, 32)	1056	critic-action[0][1]
critic-encoder (LSTM)	[(None, 12, 128), (None, 128), (None, 128)]	67072	input_2[0][0]
batch_normalization (BatchNormalization)	(None, 32)	128	dense_3[0][0]
concatenate (Concatenate)	(None, 160)	0	critic-encoder[0][1] batch_normalization[0][0]
dense_4 (Dense)	(None, 400)	64400	concatenate[0][0]
batch_normalization_1 (BatchNormalization)	(None, 400)	1600	dense_4[0][0]
dense_5 (Dense)	(None, 300)	120300	batch_normalization_1[0][0]
batch_normalization_2 (BatchNormalization)	(None, 300)	1200	dense_5[0][0]
dense_6 (Dense)	(None, 1)	301	batch_normalization_2[0][0]

Total params: 261,817
Trainable params: 260,353
Non-trainable params: 1,464

None

Figure 5.5: Critic Model Summary

Chapter 6

Experiments and Results

The performance of the reinforcement-learning-based solution implemented in this thesis is evaluated in this chapter. We will first describe details of the experiments which were run and the parameters used. Then, we will share the data obtained and the insights generated.

The solution involves a LSTM neural network structure trained using reinforcement learning. In Chapter 3 we referenced feed forward neural networks and some of the attempts at designing machine-learning-based schedulers that involve them. So for comparison to our solution, a feed forward neural network was also trained and used in the experiments. This network was modeled on the work done in [33] and is implemented using fully-connected deep neural networks trained using the standard DDPG algorithm. NNs are also capable of solving the downlink scheduling problem and the authors of [33] were able to produce a network which performed well in terms of throughput and fairness compared to some traditional algorithms, even though their primary target was head of line delay reduction. Their limitation is their inability to handle changing numbers of UEs without retraining the model or training. The other schedulers used in the comparisons are the three described in Chapter 2. The Max Throughput scheduler is expected to consistently provide the highest throughput at the expense of fairness while

Round Robin and Proportional Fair are expected to better balance throughput and fairness in different scenarios.

Neither the feed forward neural network nor our solution network were optimised for performance. So while we expect them to perform reasonably well, we cannot argue that they are better or worse in terms of performance against the traditional schedulers because they have already been tuned to be optimal for their desired output. What we can however show is that the developed solution is suitable for the task, delivers the benefit of flexibility and that its performance remains consistent with expectations in the face of the changing external factors to which all schedulers in the evaluation will be exposed.

6.1 Simulation Parameters

Table 6.1 lists all relevant parameters for the common ns-3 simulation used to generate the results. All simulation parameters were held constant in the experiments except for the parameters of interest: the scheduler, the number of UEs, the number of RBs and the PRNG run variable used to seed the simulation. The trace files provided as outputs from the ns-3 simulations were collected and analysed to determine the throughput (in Mbps) and fairness (using Jain's index) achieved in each scenario.

Parameter	**Current Value**
Simulator	ns-3.30.1
OS	Train: Fedora 33 64-bit Test: macOS 10.14.6
Processor	Train: Intel Core i7-8700 CPU @ 3.20GHz Test: Intel Core i5 CPU @ 1.6 GHz
PRNG	Seed = 400 Run = Change for each run – guarantees independence of streams
Number of Runs per Scenario	31
Number of UEs	Default: 12 Otherwise: Varies from 6 - 96
DL Bandwidth	Default: 1.4 MHz (6 RBs) Otherwise: 3 MHz, 5 MHz, 10 MHz or 15 MHz (as defined in 3GPP specification)
Number of eNodeBs	1
eNobeB Power transmission	43 dBm
Frequency Reuse Scheme	Hard Frequency Reuse
Pathloss model	ThreeGppUmaPropagationLossModel (Urban Macrocell)
Channel model	ThreeGppUmaChannelConditionModel
Center frequency	500 MHz
Antenna configuration	SISO
Mobility model	Stationary
Simulation duration	1s
Traffic Type	CBR/UDP
Packet Size	4096 bytes
Packet Interval	10 ms

Table 6.1: Simulation Parameters

6.1.1 Topology

The topology used in the simulation is depicted in Figure 6.1. The single eNodeB (one of the yellow nodes) is positioned at coordinates (50, 50, 0) along with the nodes representing the remote host, evolved packet core and a point to point helper. The UEs (numbered and shown in red) are distributed in a disc of radius 1km around the eNodeB. The concentric lines on the image show the range of the signal from the eNodeB. The UEs are distributed using a uniform disk allocator which positions the UEs "randomly within a disc D lying on the plane z and having center at coordinates (x, y, z) and radius ρ such that, for any subset $S \subset D$, the expected value of the fraction of points which fall into $S \subset D$ corresponds to $\frac{|S|}{|D|}$, the ratio of the area of the subset to the area of the whole disc." [28]. This distribution ensures that the CQIs of the UEs are spread across the full range from 1 - 15.

6.1.2 Training, Evaluation

The training and data collection portions of this investigation are separated. First, the models are trained for the specified number of episodes, 300 in the case of the NN and 500 for our LSTM/Pointer. These training lengths were chosen because they represented the a point at which the network seemed to have learned to achieve reward values within a small range and would not improve significantly with more training. After training, the target-actor networks are used to make scheduling decisions in the evaluation scenarios. The final weights from the target-actor networks are copied to a replica of the network and the training sequence is discontinued.

In order to evaluate the performance of the trained scheduler we test it on

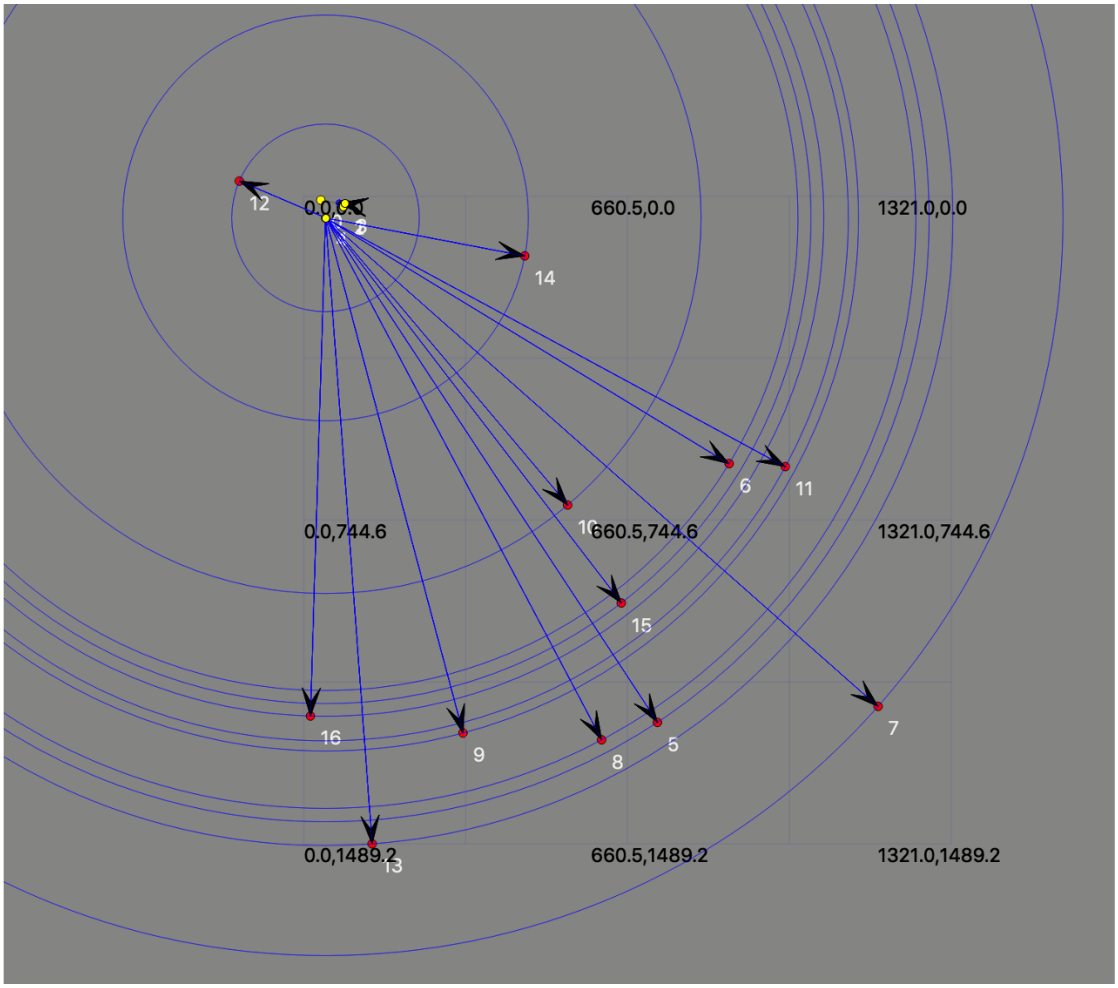


Figure 6.1: Simulation Topology

unseen simulation runs. To achieve this the PRNG run variable for the evaluation simulations are set outside the range which would have been seen in the training episodes. As noted in Table 6.1, in the ns-3 environment holding the PRNG seed variable constant and changing the run variable guarantees independent random number streams so we can be confident that the exact simulation conditions were not present in both the training runs and the evaluation runs.

6.1.3 Training Parameters

Tables 6.2 and 6.3 lists all relevant parameters for the training of the Neural Network (for comparison) and the LSTM-based pointer solution. The parameters were introduced in Chapters 4 and 5. In order to monitor the progression of the training, a moving average of the reward from training episodes was plotted for each model. These plots are shown in Figures 6.2(a), 6.2(b) and 6.3. They illustrate how the network improves over time and when it has learned a reasonable policy. Reasonableness here means it is relatively consistent, fluctuating only around a small value compared to the range of possible values as would be seen if the policy were random. The number of training episodes chosen for the final models correlate to how long it appeared to take to learn a reasonable policy. Two plots are provided for the neural network because it had to be retrained to accommodate a change in the number of UEs, whereas the LSTM/Pointer solution was trained once and reused to generate all the scenarios.

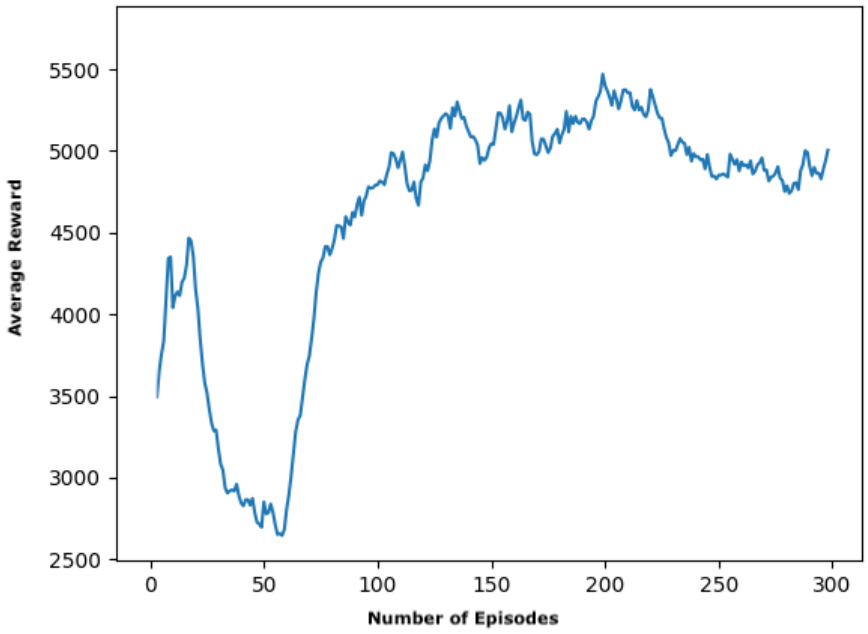
Note that Figures 6.2(a), 6.2(b) and 6.3 cannot be used to make any inference about the performance of the networks on the target metrics (throughput and fairness) compared to each other. There is no unit assigned to the episodic rewards received. That is to say, a maximum average reward of approximately -500 in Figure 6.3 does not imply 'better or worse' performance than a max of approximately 5500 in Figure 6.2(a). The numerical difference in the average reward value ranges can be accounted for by the difference in reward functions used to train the NN and LSTM/Pointer solutions and also by the fact that the rewards were assigned on a per UE basis such that the reward sum for episodes with twelve UEs versus six UEs could be different. The performance of the networks can only be evaluated

Training Hyperparameters - NN	
Number of Episodes	300
Simulation Length	2s
Noise	Ornstein-Uhlenbeck
Replay buffer capacity	1000000
Mini-batch size	64 samples
Network weights initialization	uniform initializer min value = -1.0 max value = 1.0
Optimiser	Adam
Critic learning rate	0.002
Actor learning rate	0.001
Tau	0.005

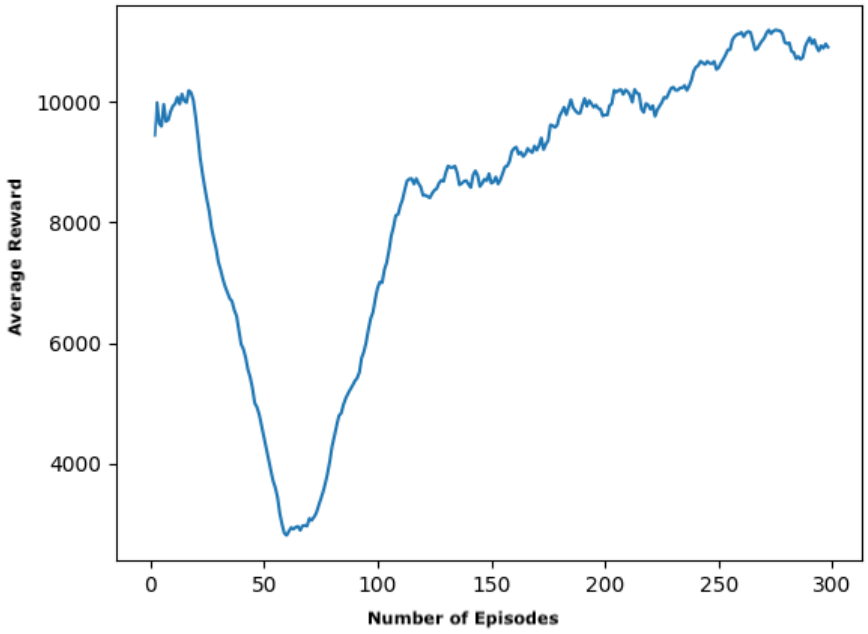
Table 6.2: NN Training Hyperparameters

Training Hyperparameters - LSTM/Pointer	
Number of Episodes	500
Simulation Length	2s
Replay buffer capacity	1000000
Mini-batch size	64 samples
Network weights initialization	uniform initializer min value = -1.0 max value = 1.0
Optimizer	Adam
Critic learning rate	0.002
Actor learning rate	0.001
Tau	0.005

Table 6.3: LSTM/Pointer Network Training Hyperparameters



(a) 6 UEs



(b) 12 UEs

Figure 6.2: Moving Average Reward, Neural Network

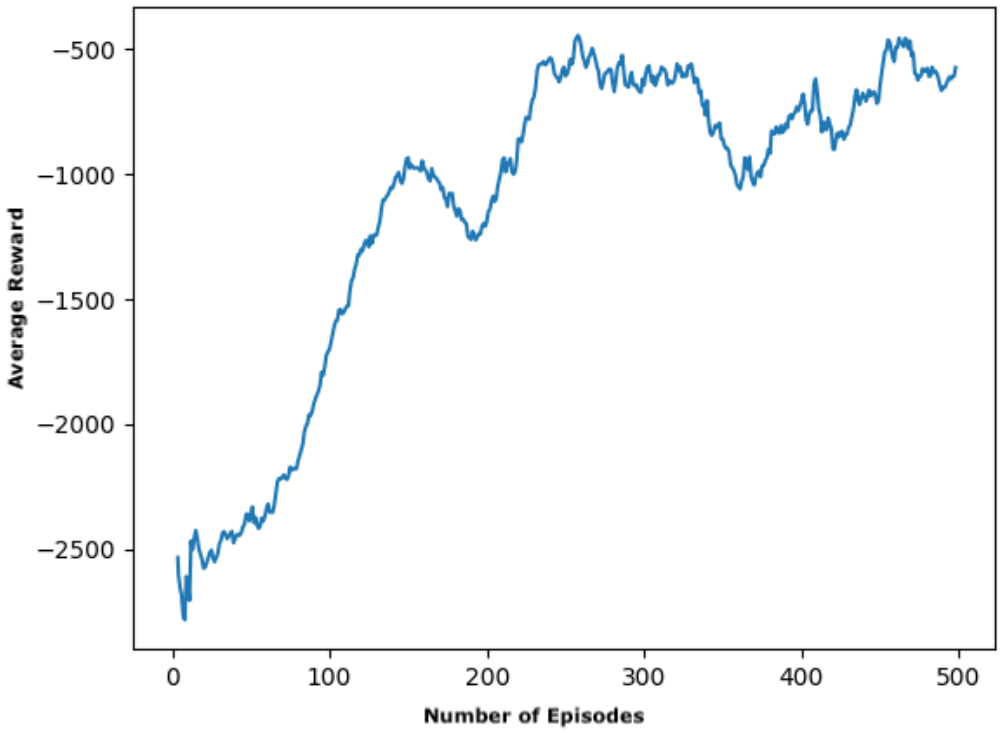


Figure 6.3: Moving Average Reward, LSTM/Pointer Network

by comparing the target metrics after simulation.

6.2 Results

We compared the performance of the different algorithms in three scenarios.

1. Where there are an equal amount of resource blocks as UEs
2. Where there are more resource blocks than UEs
3. Where there are more UEs than resource blocks

This was to investigate how the solution performed in comparison to the established scheduling algorithms as well as to the feed forward NN model. It also allows us to explore how the change in ratio of UEs to resource blocks changes the performance of the solution in comparison to the established scheduling algorithms and NN model. The UEs were sent traffic at a rate which ensured that they were always backlogged with data to be received. The results are plotted in Figures 6.4 to 6.6.

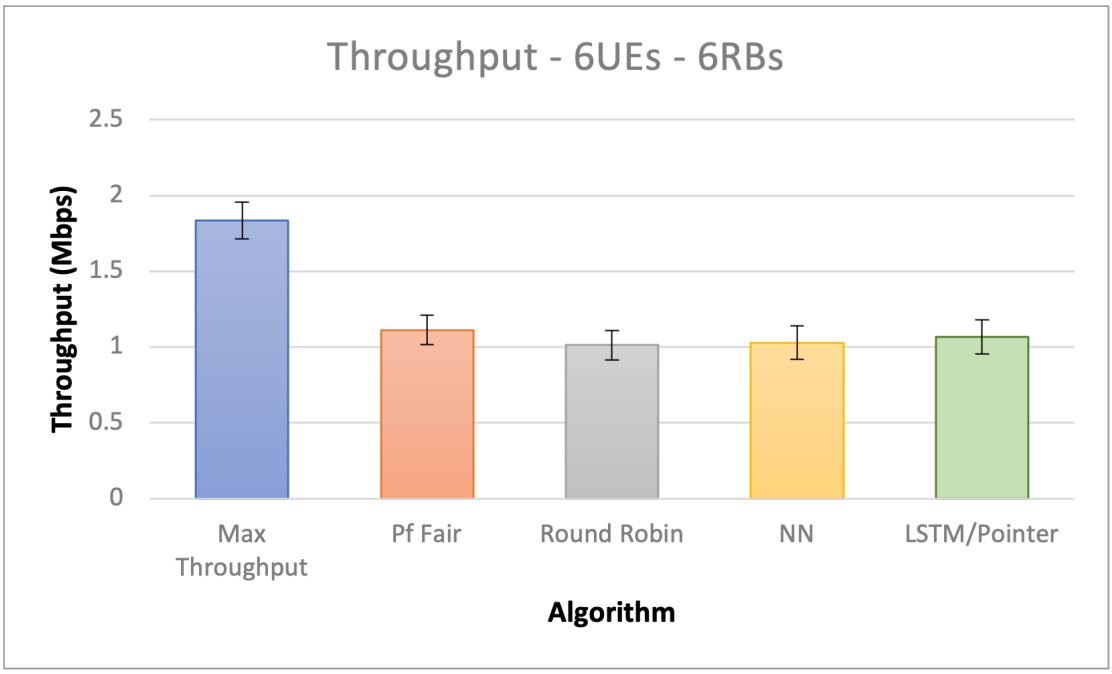
In all three scenarios the proposed LSTM/Pointer-based scheduler performed similarly to the PF Fair, Round Robin and Neural Network algorithms in terms of average throughput. In terms of fairness, the proposed scheduler was better than Max throughput but worse than the other three algorithms. The error bars in Figures 6.4 to 6.6 represent the 95% confidence intervals, the narrowness of these intervals confirms that the learned policy of the network is stable rather than random. Further, the fact that the network learned a policy which generates results that are within the range of the established schedulers shows that the solution is suitable for the task and presents an opportunity to tune for better results. Additionally, in the face of changing simulation conditions the solution responds with the same trends as the established schedulers i.e. throughput and fairness rise and fall when all other schedulers rise or fall, suggesting that the flexibility of the solution does not impact the stability or cause unexpected behaviors.

Figures 6.4 to 6.6 also show that while the implemented solution tracks Proportional Fair and Round Robin in terms of throughput, it lags both in fairness. This we think can be attributed to the reward function defined in Chapter 4. While the function rewards reducing the buffer size of UEs and penalizes the scheduling of UEs with no data in the buffer or allowing a UE to go unscheduled for a long time, it does not explicitly nudge the solution towards fairness. This could be attributed to the fact that more points are awarded for a reduction in buffer size than are removed by ignoring a UE for a long time. It is likely, once the network finds a combination of UEs which are consistently resulting in a reduction of the buffer size, that it may keep scheduling these UEs until the buffer is drained (and therefore producing a negative reward). This leaves an opportunity to design a different reward function to better encourage fairness.

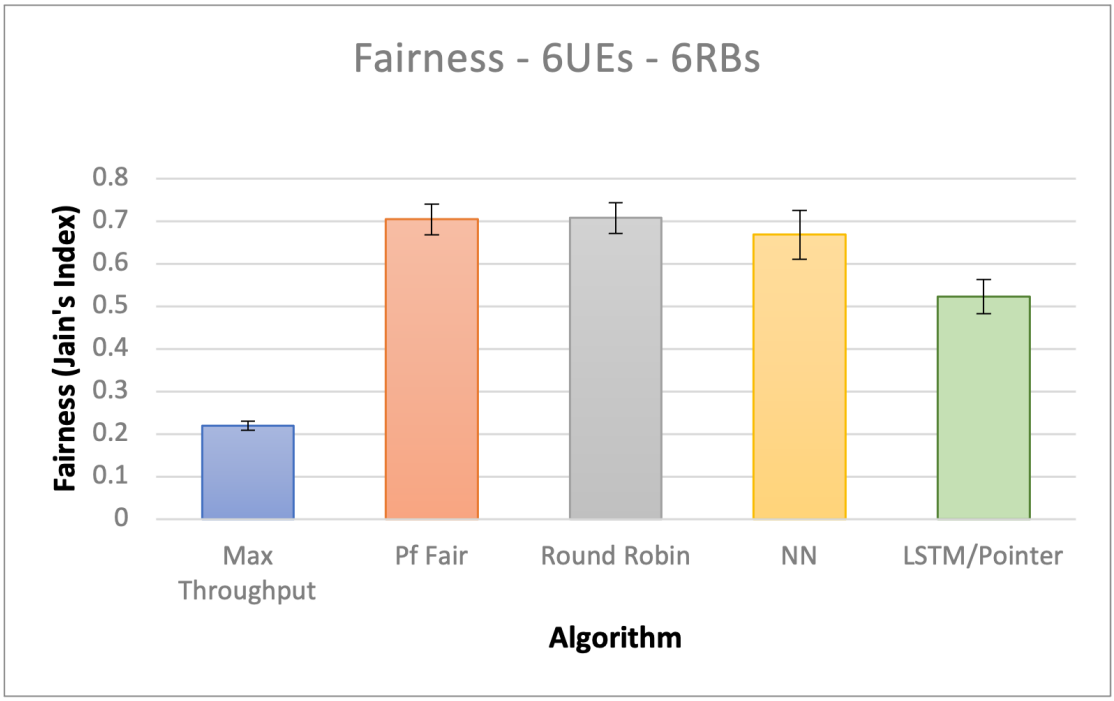
Future Stability

We further explored how far into the future (from the number of UEs it was trained on) the implemented solution would remain useful. The plots in Figures 6.7(a) and 6.7(b) show that with regard to throughput, the solution continued to track the performance of Round Robin and the Proportional Fair algorithm, while with respect to fairness it tracks the max throughput curve in trajectory but has a value approximately 1.5 times higher than max throughput.

This result confirms that the trained LSTM/Pointer structure has generalized beyond its specific training conditions (meaning it is able to handle scenarios not seen during training) and can in fact be useful in the context of changing numbers of UEs. The ability to maintain performance at three times the number of UEs used in training suggests that the LSTM/Pointer scheduler will remain viable even

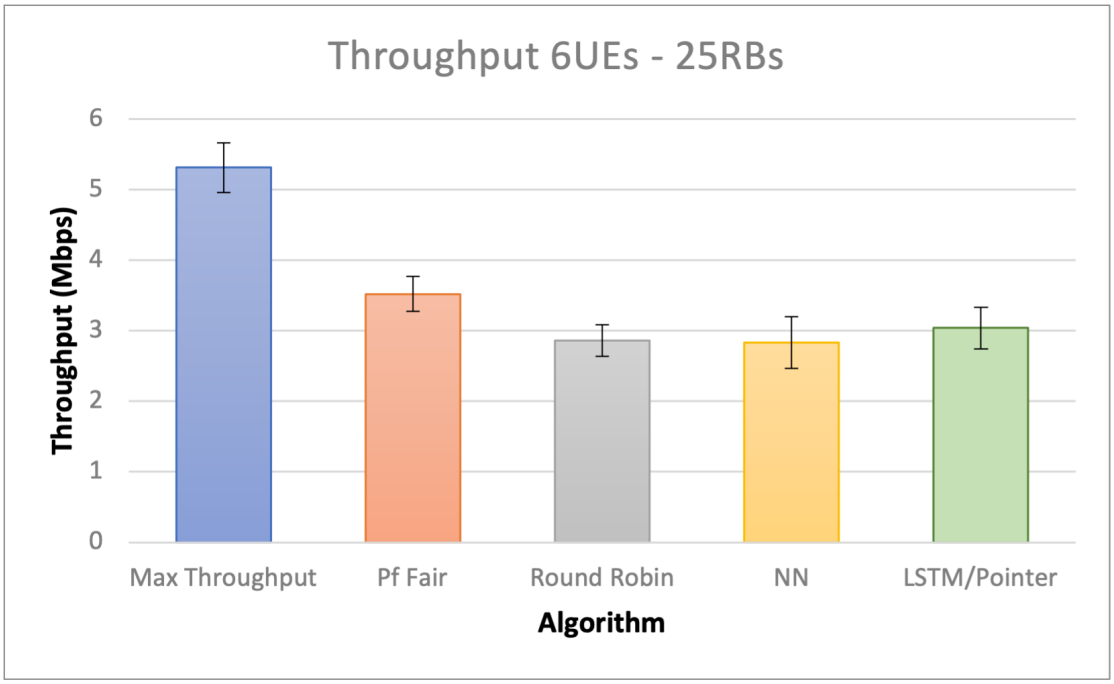


(a) Throughput

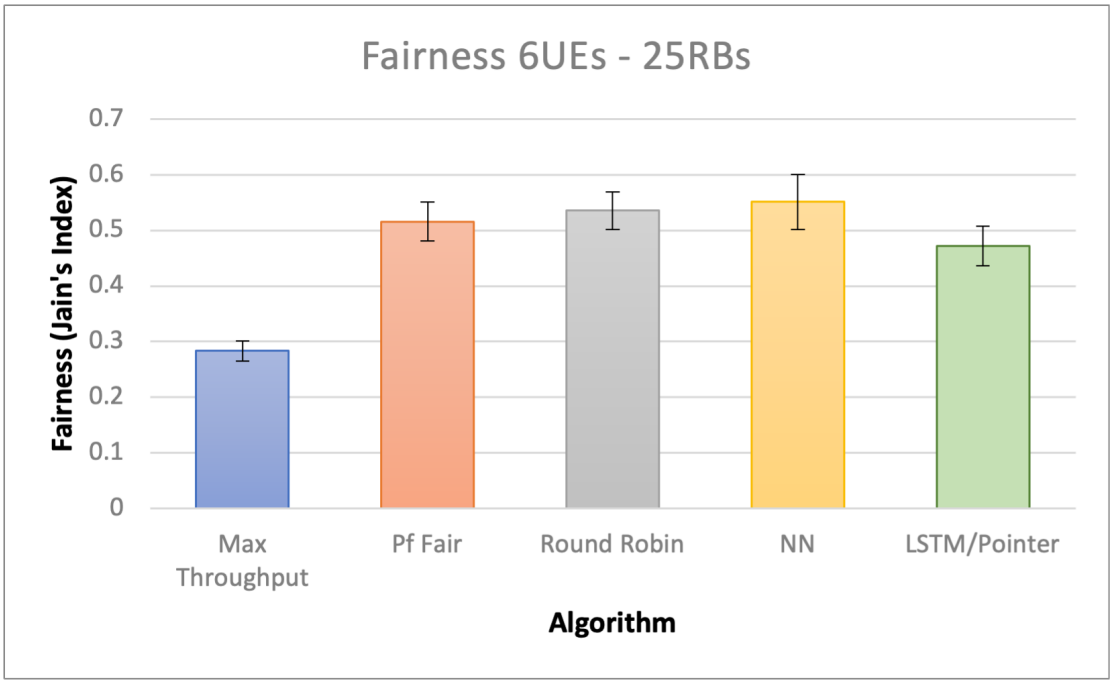


(b) Fairness

Figure 6.4: Comparison of Throughput and Fairness for 6 UEs, 6 RBs

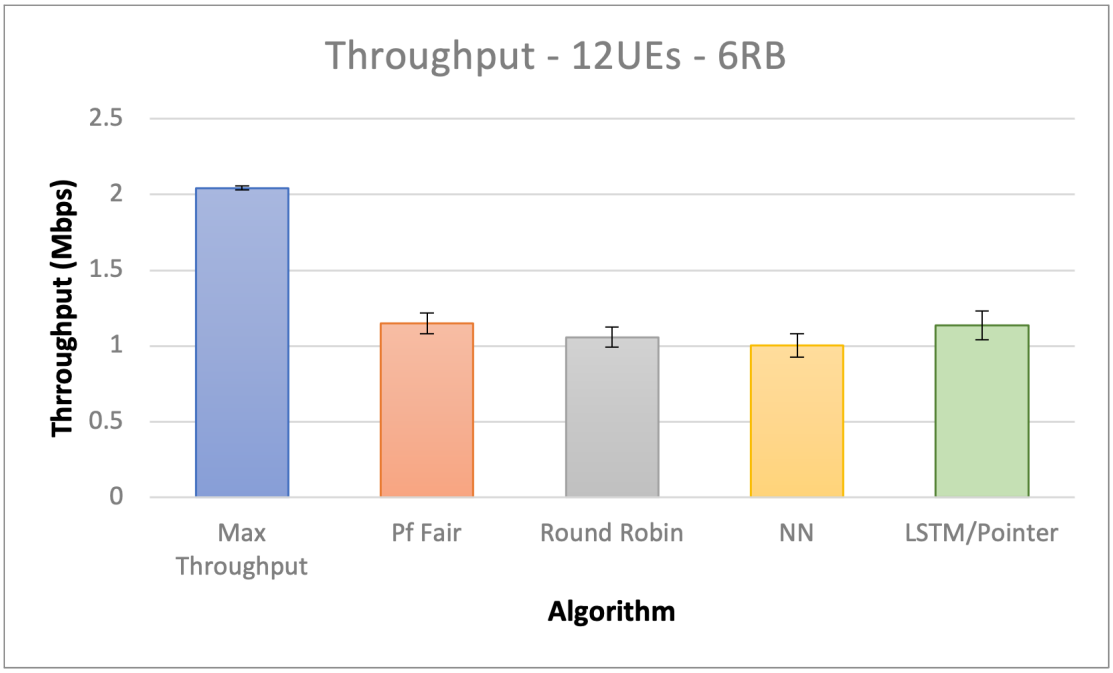


(a) Throughput

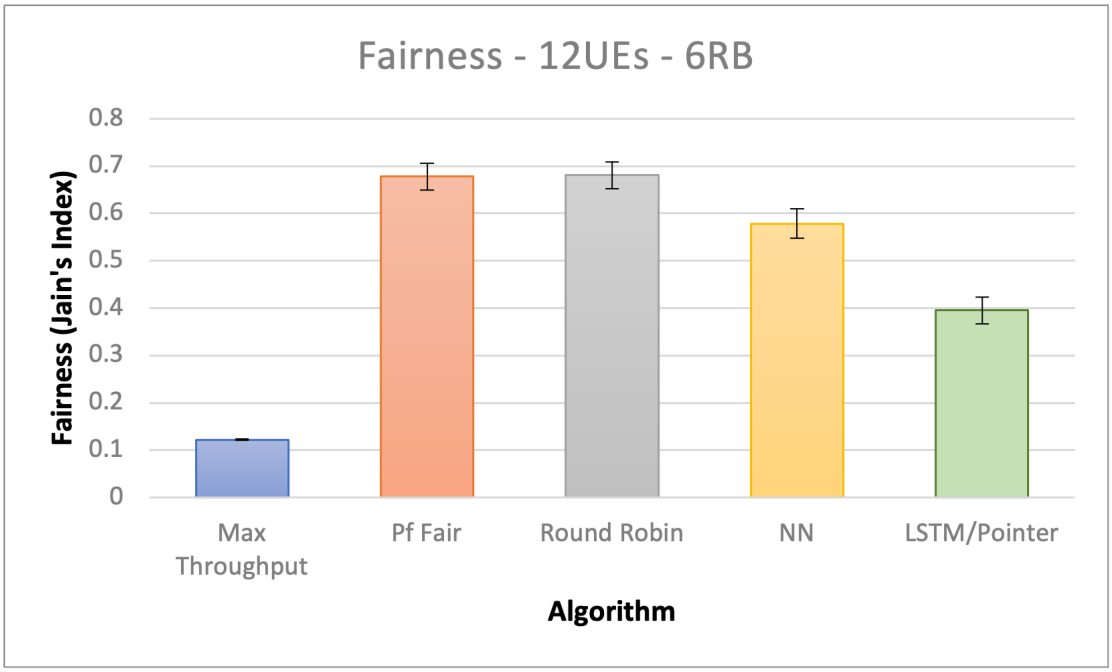


(b) Fairness

Figure 6.5: Comparison of Throughput and Fairness for 6 UEs, 25 RBs



(a) Throughput

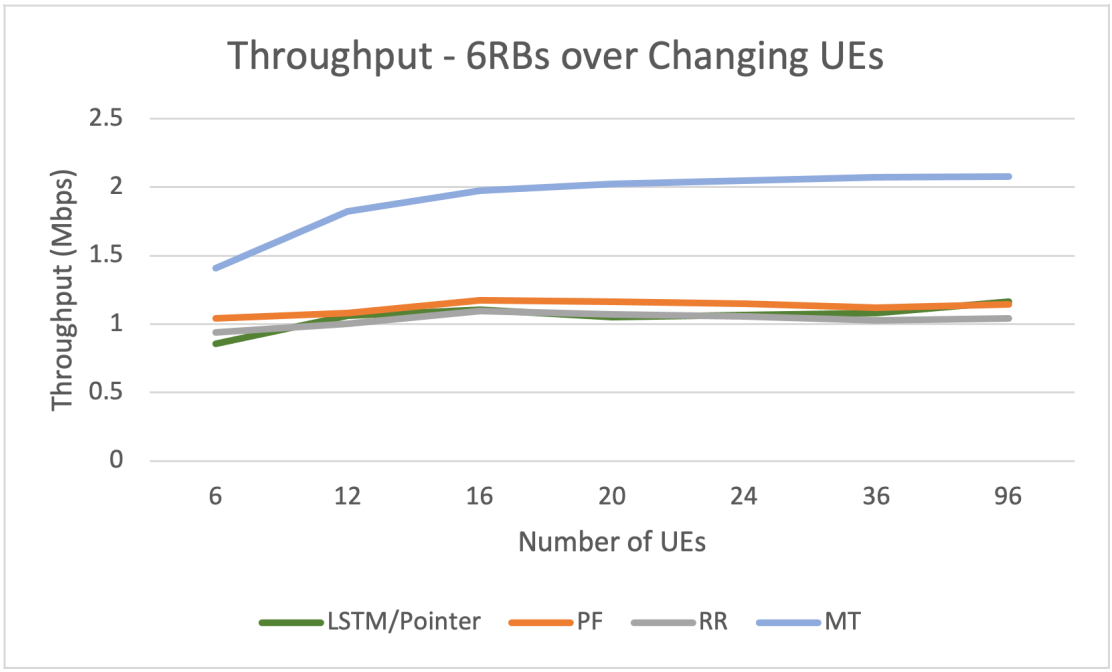


(b) Fairness

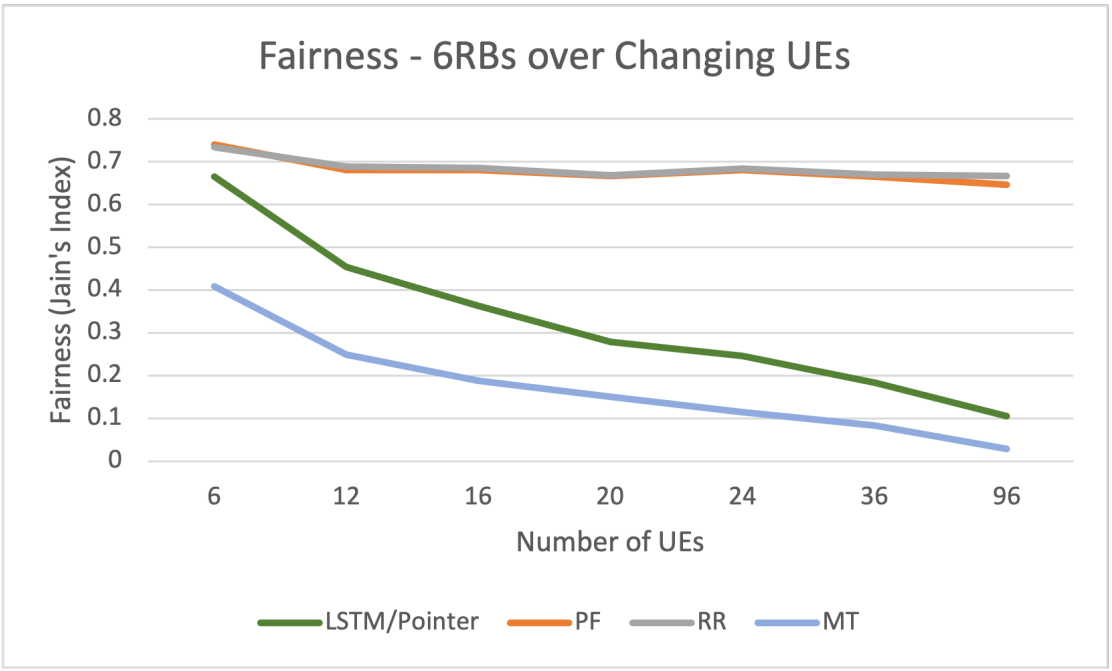
Figure 6.6: Comparison of Throughput and Fairness for 12 UEs, 6 RBs

in surge demand conditions on the network.

Finally, though the solution was developed to handle changing numbers of UEs, which occurs frequently in an LTE cell, the use of LSTM units in the Decoder also lends itself easily to handling changing numbers of RBs. This would not happen frequently within a cell, but having the ability to reuse a trained model in a new cell with more or less resource blocks is also useful, as well as being able to reuse the model within a cell if more spectrum is obtained by the operator. To test this viability we ran a set of scenarios, again comparing the solution to the traditional scheduling algorithms while changing the number of RBs (with the number of UEs fixed to 12). Figures 6.8(a) and 6.8(b) show that the LSTM/Pointer scheduler reasonably maintains its performance up to approximately 4x the number of RBs it was trained on before degrading slightly in comparison to the other algorithms. This result is comparable to the scenario where the number of UEs was changed. LSTMs were designed to improve the memory of recurrent neural networks but there are still several factors which could have them lose effectiveness, including the ordering of the input sequences as noted in [38] and [25]. More experiments would have to be designed to determine what factors lead to the eventual decline in performance.

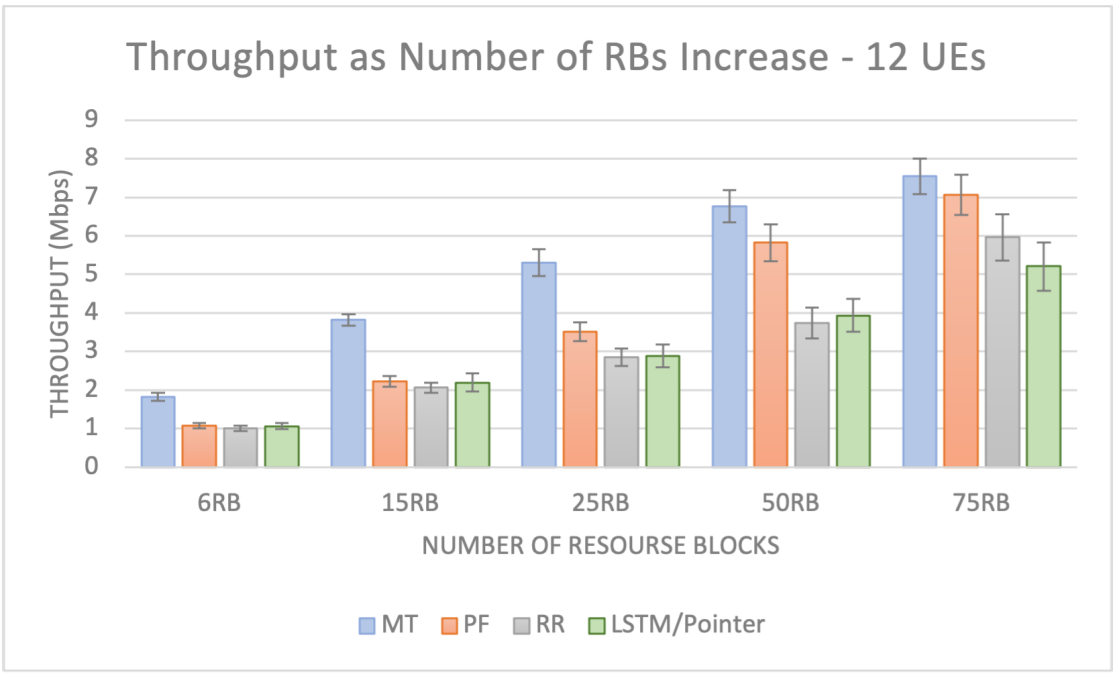


(a) Throughput

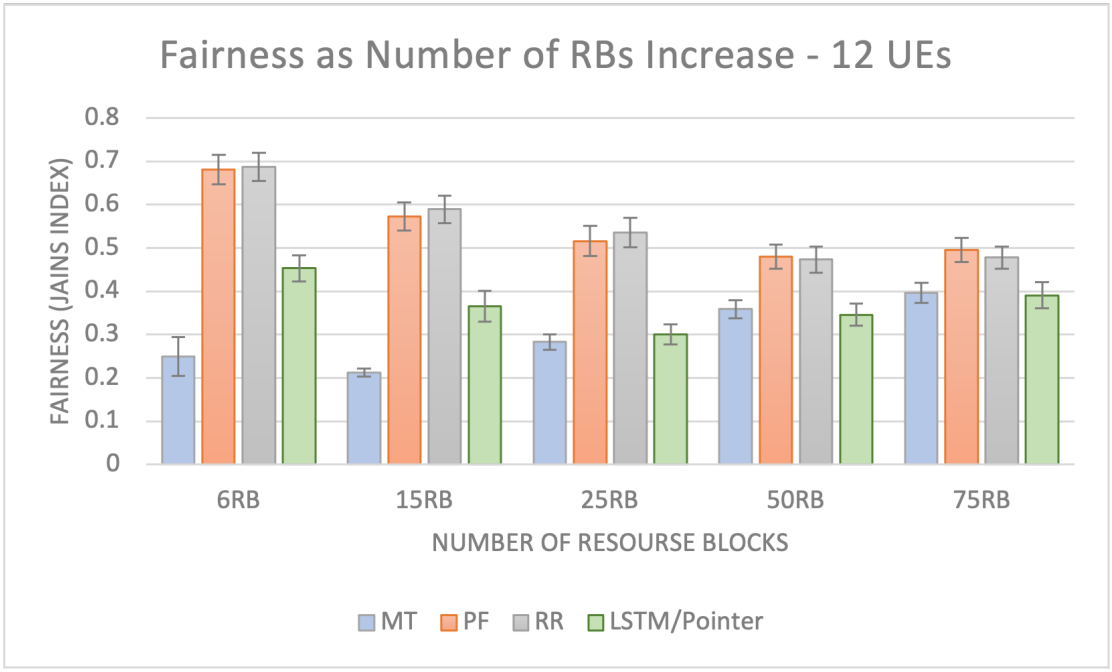


(b) Fairness

Figure 6.7: Throughput and Fairness as the Number of UEs Increase.



(a) Throughput



(b) Fairness

Figure 6.8: Comparison of Throughput and Fairness as the Number of RBs Increase.

6.3 Summary

The data generated by the experiments described in this chapter demonstrate that the implemented solution delivers the flexibility it was designed for while maintaining performance in the range of traditional schedulers. Some areas for improvement were identified including refactoring the reward function to explicitly encourage fairness, as well as optimization i.e. exploring the training hyper-parameters space to determine the set of parameters which deliver the best performance. Additionally, other forms of optimization can be pursued, such as tweaking the network structure itself. LSTM units can be replaced with other types of recurrent units and the Actor and Critic models could incorporate more or fewer layers and layer types.

Chapter 7

Conclusion and Future Work

7.1 Summary and Conclusions

In this thesis we set out to develop a reinforcement learning solution which can flexibly handle changing numbers of UEs without retraining the model or resorting to padding the inputs. We believe that such a structure could potentially be more suitable for the growing demands of modern LTE networks.

In Chapter 2 we give an overview of the relevant factors for this study including: the functioning of an LTE network and the mechanics of scheduling, scheduling metrics, existing scheduling algorithms, and machine learning. In Chapter 3 we explore some of the existing attempts to design schedulers using reinforcement learning and the open areas for development. We identify that many solutions have focused on utilizing feed forward neural networks which have fixed dimensions. Those solutions which sought to handle changes in UE numbers resorted to padding the input. We take this as an opportunity to design a more flexible solution.

In Chapters 4 and 5 we introduce our proposed solution utilizing LSTM units to handle variable input and outputs. This is further augmented by an attention mechanism resulting in a pointer structure which produces output sequences

consisting only of the UEs present in the input sequence. This structure is implemented using ns-3 as the simulator, Keras and Tensorflow for the reinforcement learning agent and ns3gym as the bridge between the two.

Finally we share the experiments run and results generated in Chapter 6. From these results we conclude that the implemented solution allows for flexibility in handling training conditions where the number of user equipment and resource blocks may be fixed, and testing or real world environments where these may fluctuate without the need to retrain the model. The solution is shown to generalize to the new conditions and maintains a performance within the range of traditional schedulers.

7.2 Future Work

The training of the solution was not optimised, meaning experiments were not run to tune the training parameters to explore the full range of results and an optimal set selected. This leaves the opportunity to gain more performance out of the network through hyper-parameter tuning. Optimization efforts should also include refactoring the reward function to encourage better fairness.

Additionally, we remove the noise portion of the original DDPG algorithm when training the solution because we found that the action space noise used produced worse results than the no noise situation. Some papers such as [31] propose the use of parameter noise instead of action space noise to promote exploration which may also be a valuable avenue to pursue for more performance.

One other avenue for exploration is in changing the structure itself. In this thesis we made use of LSTMs. However, there are other forms of recurrent units such as GRUs (Gated Recurrent Units) which are also improvements over the regular RNN and employ different gating mechanisms. In some use cases GRUs

can perform better than LSTM units. It may be useful to swap the LSTM units for other types and see if increased performance can be achieved.

List of References

[1] Yu Abiko, Daisuke Mochizuki, Takato Saito, Daizo Ikeda, Tadanori Mizuno, and Hiroshi Mineno. Proposal of allocating radio resources to multiple slices in 5G using deep reinforcement learning. *2019 IEEE 8th Global Conference on Consumer Electronics, GCCE 2019*, (17):129–130, 2019.

[2] Faroq Al-Tam, Noelia Correia, and Jonathan Rodriguez. Learn to schedule (LEASCH): A deep reinforcement learning approach for radio resource scheduling in the 5G MAC Layer. *IEEE Access*, 8:108088–108101, 2020.

[3] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *5th International Conference on Learning Representations, ICLR 2017 - Workshop Track Proceedings*, pages 1–15, 2019.

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[6] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-solano, and Oscar M Caicedo. Comprehensive survey machine learning. *Journal of Internet Services and Applications*, 9(16):99, 2018.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, arXiv:1606.01540, 2016.

[8] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda. Downlink packet scheduling in LTE cellular networks: Key design issues and a survey. *IEEE Communications Surveys and Tutorials*, 15(2):678–700, 2013.

[9] Christopher Cox. *An introduction to LTE: LTE, LTE-Advanced, SAE and 4G mobile communications*. Wiley Publishing, 1st edition, 2012. ISBN: 9781118818039.

[10] Gabriel Dulac-Arnold, Richard Evans, Peter Sunehag, and Ben Coppin. Reinforcement learning in large discrete action spaces. *CoRR*, arXiv:1512.07679, 2015.

[11] ETSI. Universal Mobile Telecommunications System (UMTS); LTE; Requirements for Evolved UTRA (E-UTRA) and Evolved UTRAN (E-UTRAN) 3GPP TR 25.913. Technical report, 2009.

[12] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. *CoRR*, arXiv:1710.11583, 2017.

[13] Piotr Gawłowicz and Anatolij Zubow. Ns-3 meets OpenAI Gym: The playground for machine learning in networking research. *MSWiM 2019 - Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 113–120, 2019.

[14] Simon Haykin. *Neural Networks: A comprehensive foundation*. Prentice Hall PTR, USA, 1st edition, 1994. ISBN: 9780132733502.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[16] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pages 1–19, 2018.

[17] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *ACM Transaction on Computer Systems*, 1984.

[18] Keras. Keras documentation: Deep deterministic policy gradient (DDPG). https://keras.io/examples/rl/ddpg_pendulum/. Last Accessed: 2021-05-16.

[19] Vijay R. Konda and John N. Tsitsiklis. Actor-Critic algorithms. *Advances in neural information processing systems*, pages 1008–1014, 2000.

[20] Jaehoon Koo, Veena B. Mendiratta, Muntasir Raihan Rahman, and Anwar Walid. Deep reinforcement learning for network slicing with heterogeneous resource requirements and time varying traffic dynamics. *15th International Conference on Network and Service Management, CNSM 2019*, 2019.

[21] Rongpeng Li, Zhifeng Zhao, Qi Sun, Chi-Lin I, Chenyang Yang, Xianfu Chen, Minjian Zhao, and Honggang Zhang. Deep reinforcement learning for resource management in network slicing. *IEEE Journal on Selected Areas in Communications*, 38(2):334–349, 2018.

[22] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[23] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys and Tutorials*, 21(4):3133–3174, 2019.

[24] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. *HotNets 2016 - Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[25] Shikib Mehri and Leonid Sigal. Middle-out decoding. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 5523–5534. Curran Associates Inc., 2018.

[26] nsnam.org. LTE design documentation. https://www.nsnam.org/docs/release/3.30/models/html/lte-design.html. Last Accessed: 2021-06-23.

[27] nsnam.org. ns-3 tutorial. https://www.nsnam.org/docs/tutorial/html/. Last Accessed: 2021-06-23.

[28] nsnam.org. ns3::uniformdiscpositionallocator class reference. https://www.nsnam.org/doxygen/classns3_1_1_uniform_disc_position_allocator.html. Last Accessed: 2021-06-21.

[29] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1310–III–1318. JMLR.org, 2013.

[30] Doan Perdana, Aji Nur Sanyoto, and Yoseph Gustommy Bisono. Performance evaluation and comparison of scheduling algorithms on 5G networks using network simulator. *International Journal of Computers, Communications and Control*, 14(4):530–539, 2019.

[31] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pages 1–18, 2018.

[32] Bharath Ramsundar and Reza Bosagh Zadeh. *TensorFlow for deep learning: From linear regression to reinforcement learning*. O'Reilly Media, Inc., 1st edition, 2018. ISBN: 9781491980453.

[33] Nikhilesh Sharma, Sen Zhang, Someshwar Rao Somayajula Venkata, Filippo Malandra, Nicholas Mastronarde, and Jacob Chakareski. Deep reinforcement learning for delay-sensitive LTE downlink scheduling. In *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–6, 2020.

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. The MIT Press, second edition, 2018. ISBN: 9780262039246.

[35] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, pages 1057–1063, 2000.

[36] Csaba Szepesvári. *Algorithms for reinforcement learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010. ISBN: 9781608454921.

[37] Keysight Technologies. LTE-advanced physical layer overview. http://rfmw.em.keysight.com/wireless/helpfiles/89600b/webhelp/subsystems/lte-a/content/lte_overview.htm. Last Accessed: 2021-06-23.

[38] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pages 1–11, 2016.

[39] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[40] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

[41] Jian Wang, Chen Xu, Yourui Huangfu, Rong Li, Yiqun Ge, and Jun Wang. Deep reinforcement learning for scheduling in cellular networks. *2019 11th International Conference on Wireless Communications and Signal Processing, WCSP 2019*, 2019.

[42] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys and Tutorials*, 21(3):2224–2287, 2019.