THOMPSON RIVERS UNIVERSITY

A Novel Deep Unsupervised Learning Method for Sum-Rate Optimization in Device-to-Device Networks with a Quality-of-Service Constraint

By

Bindubritta Acharjee

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science in Data Science

KAMLOOPS, BRITISH COLUMBIA

September, 2023

SUPERVISOR

Dr. Omer Waqar

CO-SUPERVISOR

Dr. Muhammad Hanif

ABSTRACT

This study introduces a new Deep Unsupervised Learning (DUL) approach based on an optimization problem with box constraints coupled with polytope constraints for maximizing the sum rate in Device-to-Device (D2D) networks, a key factor in enhancing network capacity and efficiency. Current deep learning methods struggle with managing resource-intensive projection steps and need multiple iterations to optimize the sum rate in varying D2D environments. The proposed approach overcomes these challenges by minimizing the loss function and satisfying constraints when dealing with a monotone matrix. The novel approach controls transmit power through a fully connected, multi-layer Deep Neural Network (DNN), solving the complex, non-convex optimization problem associated with optimizing the sum rate in a symmetric interference channel model. The result shows that this method outperforms other power control methods regarding average sum rate, hit rate, and complexity when applied to a standard symmetric K-user Gaussian interference channel.

Key Words: D2D communication; Sum-rate optimization; Deep Learning (DL); Unsupervised Learning (UL); Box constraints, Monotone matrix.

ACKNOWLEDGEMENTS

I am tremendously grateful for the opportunities and support I have received throughout this journey, which has culminated in the successful completion of this Master's thesis. The experience has been an academic challenge and a journey of significant personal growth, and this page is dedicated to acknowledging everyone who has contributed to making this journey possible.

First and foremost, my profound gratitude goes to my thesis supervisor, Dr. Omer Waqar, and co-supervisor, Dr. Muhammad Hanif. Your expertise, guidance, and unwavering faith in my capabilities have been integral to my accomplishments. Your patience and wisdom have helped me develop a robust research project and taught me the essential qualities of a thoughtful researcher and academic.

I would also like to sincerely thank my committee members, Dr. Jabed Tomal and Dr. Yasin Mamatjan, for their invaluable input and constructive feedback throughout the development of this work. Your perspectives and insights have played a significant role in shaping and refining my research.

I would also like to thank my MScDS program coordinators, Prof. Roger Yu and Prof. Mohamed Tawhid, who have facilitated a smooth and conducive environment for research.

My heartfelt thanks go to my family, who have supported and encouraged me during this endeavor. Your faith in my abilities, patience, and understanding during the stressful periods of this research have been my foundation.

Finally, I acknowledge the financial support from my supervisor, Dr. Omer Waqar, as a form of a research assistantship, without which this research would not have been possible.

As I write this acknowledgment, I realize that this journey, despite its challenges, has been a privilege that I could not have navigated without the combined efforts and encouragement of all those mentioned. Thank you all for being a part of this meaningful voyage.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The evolution of wireless communication systems and the exponential increase in mobile devices have sparked a critical requirement for more sophisticated network management mechanisms. The proliferation of smart devices and the exponential growth of data traffic have necessitated the development of efficient wireless communication methods. D2D communication, a significant constituent of 5G and beyond networks, has been seen as a promising solution to meet these evolving demands. Direct data exchanges among two nearby mobile devices, without data relay via Base Stations (BSs), may provide better spectrum, energy, and transmission latency performance [1, 2, 3].

As illustrated in Figure 1.1, D2D communication refers to the technology that allows devices to communicate with or without network infrastructure involving access points or BSs [4]. This technique allows new device-centric communication that frequently does not require direct interaction with the network infrastructure; therefore, it is expected to alleviate certain aspects of the network capacity issue as 5G promises to connect more devices to faster, more reliable networks. However, the success of D2D communication relies heavily on achieving the optimal sum rate, a major parameter defining the total data transmission capacity of the network.

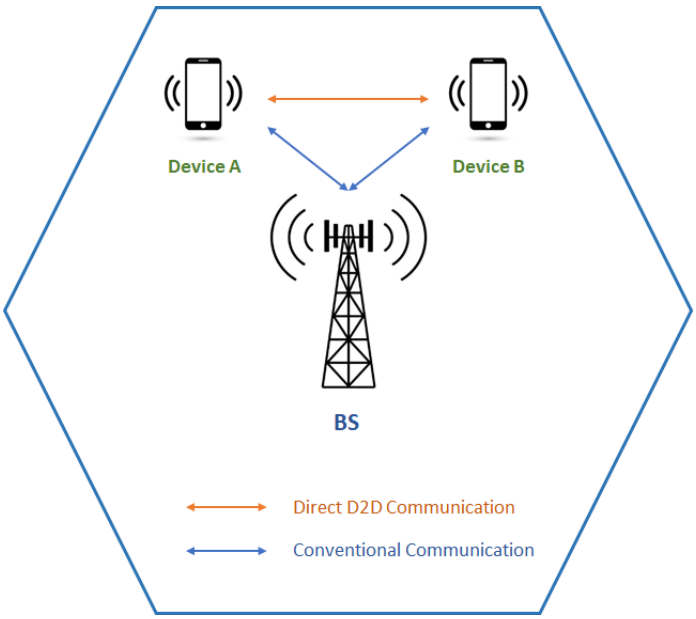

Figure 1.1: Direct D2D communication between devices and conventional communications with BS

A crucial challenge in D2D networks is to optimize the utilization of limited wireless resources to achieve higher data rates while maintaining the Quality of Service (QoS) requirements. Sum-rate maximization, which aims to maximize the total achievable data rate in the network, plays a pivotal role in addressing this challenge. However, the quest to achieve a higher sum rate in D2D wireless networks comes with several challenges, including:

- **Interference Management:** With D2D communication, devices operate close to each other, leading to significant cross-tier and co-tier interference. This interference can significantly degrade the network's performance and impact the overall sum rate.
- **Resource Allocation:** Efficient resource allocation, including power, spectrum, and time, is crucial for optimizing the sum rate. However, it is challenging due to the dynamic nature of the wireless environment and the large number of D2D pairs sharing the same resources.
- **Scalability Issues** With the increase in the number of devices, managing and maintaining the quality of service becomes significantly more challenging. Traditional methods may scale poorly with the rapid increase in the number of

devices.

- **User Mobility** Users' movement in D2D networks introduces additional challenges to maintaining consistent and reliable connections. The dynamic topology due to mobility affects the overall network performance and sum rate.

- **Security and Privacy Concerns** Security and privacy are significant challenges in D2D communications. Malicious attacks and eavesdropping can disrupt network operations, affecting the sum rate.

- **Signal Propagation Conditions** Factors like path loss, shadowing, and multipath fading can adversely affect the signal quality, thereby impacting the sum rate in D2D networks.

- **Device Heterogeneity** Device heterogeneity poses a significant challenge for D2D wireless networks. It refers to the variance in capabilities, functionalities, and technical specifications among different devices participating in the network. These variations could be in processing power, memory, battery life, communication range, network protocols, operating systems, or software applications. This heterogeneity adds complexity to the optimization of the sum rate.

- **Energy Efficiency** A high sum rate requires high transmission power, which can drain device batteries quickly. Therefore, balancing energy efficiency and sum rate is a significant challenge.

Considering all the challenges, especially the nature of the wireless communication medium, careful control of the transmit power of the User Equipment (UE) is considered the most appropriate method for managing interference and enhancing the overall system performance [5].

Advanced techniques such as machine learning and optimization algorithms are being explored to manage these challenges of D2D networks. These techniques aim to provide flexible and intelligent solutions that can adapt to the dynamics and diversity of D2D networks and can help optimize resource allocation, manage interference, and enhance the overall performance of D2D networks. However, implementing these

techniques also has its own set of challenges, such as computational complexity and model training requirements.

DL, a subset of machine learning techniques that uses a layered structure of algorithms called Neural Networks (NNs), has shown its potential in solving complex and nonlinear problems in various domains, mainly due to the latest low-cost and advanced computational power. Recent advancements in processing capabilities, including the development of Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), and distributed computing technologies, have made it feasible to train complex deep learning models. Therefore, DL has been introduced recently to wireless communication and has shown promising improvements in optimizing network performance parameters.

Supervised Learning (SL), UL, and Reinforcement Learning (RL) are the three basic categories of procedures used to train NNs in DL. Because labeling a data collection is frequently prohibitively time-consuming, SL is regarded as unfeasible, especially for large wireless networks. Again, due to the inherent exploitation-exploration trade-off, RL is only appropriate for problems described as Markov Decision Processes (MDPs), and convergence of these methods is typically challenging [6]. However, as prior knowledge regarding the analytical frameworks of communication theory can be utilized to train NNs through UL effectively, it is preferred over SL and RL for various complex non-convex optimization issues in many recent studies [6, 7, 8, 9, 10]. UL leverages the underlying data distribution to learn functional patterns without supervisory signals. This approach can provide novel insights into optimizing the sum rate in D2D wireless networks.

Training a NN to solve constrained optimization problems, especially when multiple constraints are involved, is challenging. A common strategy is to include a penalty term in the loss function, allowing flexibility in managing various constraints. However, this method only provides a "soft" boundary, penalizes infeasibility, and requires tuning an additional hyperparameter. It does not always achieve optimality or feasibility. As an alternative, projection-based approaches set up a 'hard' boundary to ensure feasibility [11], but they increase computational complexity due to the requirement of an optimization solver. Approaches like OptNet, a differentiable quadratic

programming layer within the NN architecture, also suffer from high computational complexity and limited performance. One solution proposed is the double description approach, which iteratively constructs a feasible region and performs optimization within it [12]. Moreover, this method [12] is explicitly tailored for homogeneous linear constraints. Modifying such a method to accommodate non-homogeneous linear constraints may present substantial challenges. For example, numerical methods for homogeneous systems might not be stable for non-homogeneous ones. Then, in the optimization context, moving from homogeneous to non-homogeneous constraints can change the nature of the feasible region and the optimization problem's characteristics [13]. This approach was recently extended for non-homogeneous linear constraints in [14], where a correction process using gradient descent, named the Deep Constraint Completion and Correction (DC3) algorithm, ensures the feasibility of linear constraints. However, this requires an iterative process for training and testing, contradicting the objective of using deep learning models as a substitute for non-data-driven optimization algorithms.

1.2 Problem Statement

Based on the above, the existing methods require substantial computational power and time to achieve the optimal sum rate in D2D networks. In addition, they are not always adept at handling the dynamism and nonlinearity of wireless communication environments. These challenges emphasize the need for innovative approaches that can enhance the sum-rate performance of D2D networks efficiently and adaptively.

So, a new DUL-based framework is proposed that guarantees feasible solutions for any optimization problem featuring non-homogeneous linear inequality and box constraints. It uses a particular property of a monotone matrix to handle the polytope constraints and ensure the solution is possible without iterations, projections, or tuning of additional hyperparameters. Its application to power allocation for D2D networks illustrates the functionality of this approach, showing that it guarantees constraint satisfaction and improves network performance in terms of the average sum rate.

1.3 Research Objectives

The main objective of this research is to devise a novel DUL method that can be used to optimize the sum rate in D2D wireless networks. Specifically, the study aims to:

- Investigate the potential of DUL methods for D2D wireless communication and provide an overview of the current state of the art.
- Propose novel DUL-based models for sum rate optimization in D2D networks.
- Develop a framework to evaluate the performance of the proposed models and compare it with current optimization methods.

1.4 Research Questions

The study seeks to answer the following research questions:

- How can DUL methods improve the sum-rate performance of D2D networks?
- What DUL models can be proposed for sum rate optimization in D2D wireless networks?
- How does the performance of the proposed DUL models compare with existing sum rate optimization methods?

1.5 Thesis Structure

The rest of this thesis report is organized as follows:

- Chapter 2 provides a literature review on D2D wireless networks, sum rate optimization methods, and DUL techniques.

- Chapter 3 introduces the methodology of the study and presents the proposed models.
- Chapter 4 discusses the performance evaluation results.
- Finally, Chapter 5 concludes the thesis with a summary of the findings, contributions, and suggestions for future work.

1.6 Significance of the Study

This research is expected to significantly contribute to the body of knowledge in wireless communication and deep learning. By developing and evaluating novel DUL models for sum rate optimization in D2D networks, this study can pave the way for more efficient, adaptive, and intelligent wireless communication systems, which are crucial in the context of the emerging era of the Internet of Things (IoT) and 5G/beyond networks.

Chapter 2

Literature Review

This chapter comprehensively reviews the existing literature on applying unsupervised deep learning or DUL methods for achieving the optimal sum rate in D2D networks. The review focuses on understanding the limitations of traditional mathematical and heuristic-based optimization methods, exploring the dynamism and nonlinearity of wireless communication environments, and outlining the potential and effectiveness of DUL methods.

2.1 Introduction

The fifth-generation technology standard for broadband cellular networks, known as 5G, has made significant progress in connectivity and telecommunications. This innovation promises Ultra-Reliable Low-Latency Communication (URLLC), greater bandwidth, vast connectivity, increased coverage, higher data rates, and improved support for mobility. However, the deployment and maximization of 5G networks still have some challenges, particularly in achieving comprehensive coverage.

By definition, network coverage is the area covered by one BS or cell where the user can send a service request or receive a service. Enhanced coverage is one of the most significant issues in 5G and beyond networks, impacting system performance and the end-user experience. Various factors, including network density, the loca-

tion of user equipment and base stations, and environmental conditions, influence network coverage. Coverage issues include coverage holes, overshoot, poor coverage, mismatched channel coverage, and cell edge issues [15].

To meet the key parameters of 5G and beyond, i.e., ultra-low latency, ultra-high availability, ultra-speed, and ultra-reliability, various technologies are proposed, including Multiple Input and Multiple Outputs (MIMO) for massive connectivity, extreme Mobile BroadBand (eMBB) for high data rates and low latency, and millimeter Wave (mmWave) spectrum for large bandwidths [16].

D2D communication is a crucial technology for 5G networks and beyond because it can enhance network efficiency, scalability, and latency and support new use cases. It allows for direct communication between devices without the need to route the data through a central base station or access point, resulting in more efficient network use. However, while D2D communication offers many advantages, it also presents several challenges.

2.2 D2D Communication Networks

D2D wireless networks are a networking paradigm that allows devices to communicate directly without intermediary infrastructure, such as cellular towers or WiFi routers. This form of networking utilizes the principles of peer-to-peer connectivity, enabling devices in close proximity to establish a communication link directly, reducing latency and relieving network congestion from central nodes. D2D communication is integral to emerging technologies such as the IoT, Vehicular Ad hoc Networks (VANETs), and 5G and beyond wireless networks, where seamless and efficient data exchange between devices is critical. Despite the advantages, D2D networks also bring challenges in security, interference management, and power control, which are actively researched topics in this domain.

2.2.1 Background of D2D Communication

Several crucial technological advancements and changes in communication protocols over the years have shaped the development of D2D communication. In the early days of telecommunications, devices typically communicated through a centralized network infrastructure. This infrastructure included base stations or satellites that routed calls and data from one device to another. However, this system had limitations, especially regarding bandwidth and latency.

The first significant step towards D2D communication came in the late 1990s with the introduction of Bluetooth technology. This wireless technology standard was designed for exchanging data between fixed and mobile devices over short distances. Bluetooth facilitated the development of Personal Area Networks (PANs), where devices could interact directly without needing a centralized network. It was primarily used for file transfers and to connect peripheral devices.

Building on Bluetooth's short-range connectivity, Wi-Fi Direct emerged in the late 2000s as another form of D2D communication. Wi-Fi Direct allows for direct connection between devices without needing a traditional Wi-Fi network or hotspot, further expanding the scope and potential of D2D communication. This allowed for faster and more reliable connections over longer distances than Bluetooth. In academics, D2D communication was initially presented on paper [17] to allow multi-hop relays in cellular networks.

LTE-Direct, introduced in the early 2010s, was a crucial step toward mainstream D2D communication. It introduced device discovery and communication over longer distances than Bluetooth or Wi-Fi Direct and was integrated into 4G LTE cellular technology. This allowed mobile devices to discover and connect directly, bypassing the cellular base station and reducing latency and network congestion. FlashLinQ [18], a PHY/MAC network architecture for D2D communications underlying cellular networks, was the first attempt to implement D2D communication in a cellular network.

The advent of 5G technology has brought D2D communication to the forefront.

The enhanced Machine-Type Communication (eMTC), NarrowBand IoT (NB-IoT), and Vehicle-to-Everything (V2X) communication in 5G networks all heavily rely on D2D communication. With 5G, D2D communication is envisioned for high data rates and low latency applications, including autonomous driving, smart grids, and the IoT. Technologies such as 6G will likely further advance D2D, paving the way for applications that we can only begin to imagine today.

2.2.2 D2D Architecture

The cellular architecture of D2D communication is divided into two tiers [17]. The first is the macro-cell tier, where communication is between the BS and the device in the macro-cell layer. In the second tier, also called the device tier, devices communicate with each other.

It is worth mentioning that radio base stations known as Node B, evolved Node B (eNodeB), and next generation Node B (gNodeB) enable mobile phones to connect to 3G, 4G, and 5G mobile networks, respectively. gNodeB is usually abbreviated as gNB in 5G network architecture diagrams.

There are four different categories for the two-tier D2D network [19].

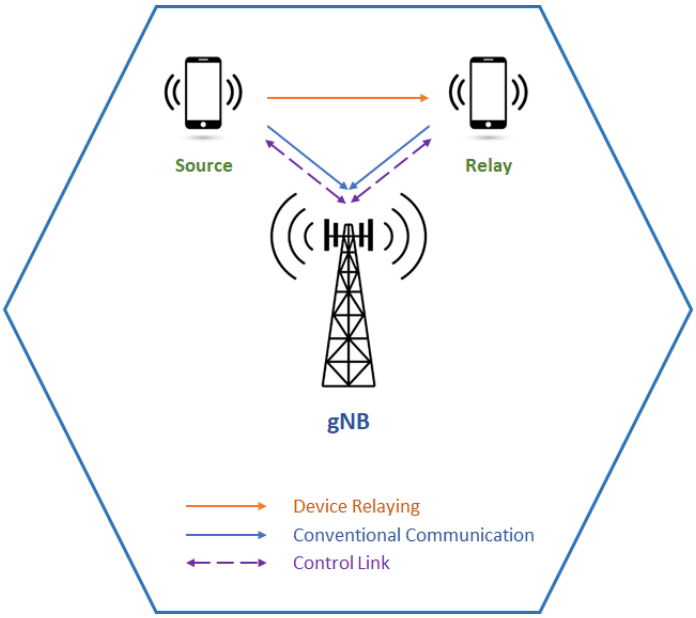

Figure 2.1: DR-OC Link Establishment

- **Device Relaying with Operator Controlled Link Establishment**

Device Relaying with an Operator-Controlled (DR-OC) link establishment system involves the interaction of devices within a network through relay points regulated and managed by an operator's base station (gNB for 5G networks), where base stations serve as relay points for connecting mobile devices. This operator's role is to establish, maintain, and terminate connections or links between these devices. Figure 2.1 shows the process of operator-controlled link establishment, which involves setting up these communication links in an organized and effective manner. The operator can monitor and control the communication traffic by allocating resources, adjusting parameters, or rerouting connections to optimize the network's performance.

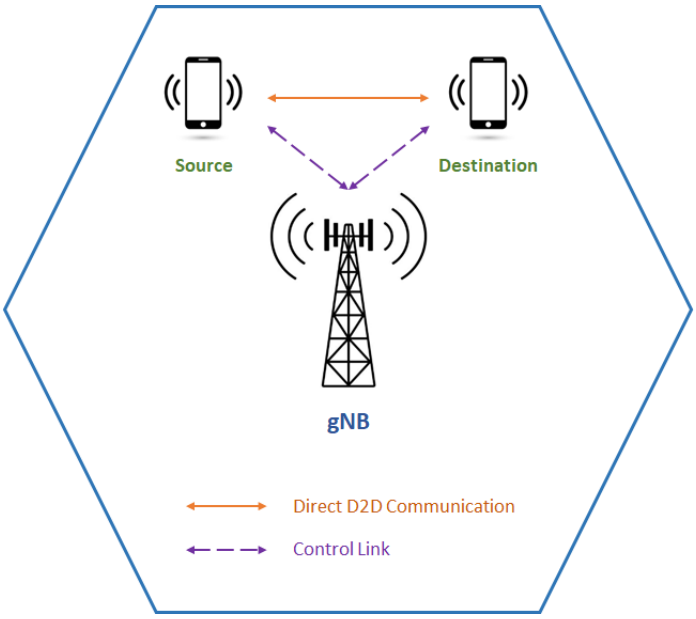

Figure 2.2: DC-OC Link Establishment

- **D2D Communication with Operator Controlled Link Establishment**

D2D Communication with Operator-Controlled (DC-OC) link establishment enables direct communication between devices without data traffic passing through the core network, thus increasing system capacity and overall efficiency. As depicted in Figure 2.2, the operator-controlled link establishment aspect provides a mechanism to effectively manage and control these connections. It involves setting up, maintaining, and

terminating the links based on network policy, resource availability, or the communication requirements of the devices. It allows for better management of resources, ensuring optimal performance, and avoiding interference with other network operations. D2D communication with operator-controlled link establishment is already being utilized in several areas. For instance, in vehicular networks, cars can use D2D communication to share information about traffic, road conditions, or safety alerts. D2D communication can enable direct communication between emergency responders in public safety scenarios, even when traditional network infrastructure is unavailable.

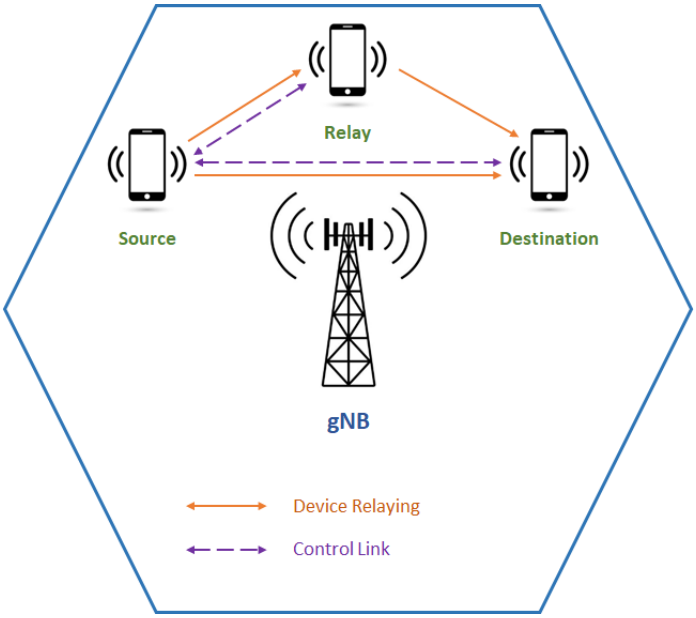

Figure 2.3: DR-DC Link

- **Device Relaying with Device-Controlled Link**

Device Relaying with Device-Controlled (DR-DC) Link Setup allows devices to independently form, manage, and disconnect links within a network without requiring a central operator. Device relaying refers to the mechanism where intermediary devices, known as relays, forward signals or data from one device to another. Device-controlled link establishment, however, empowers devices to establish, manage, and terminate their links based on predefined protocols, conditions, and requirements, as shown in Figure 2.3. This decentralization facilitates a more robust and efficient network, allowing devices to react swiftly and dynamically to changes in the network environment. Device relaying with device-controlled link establishment finds applications in various

domains, such as vehicular networks, drone swarms, and IoT networks, where devices often need to communicate directly and quickly adjust to changing conditions.

- **D2D Communication with Device-Controlled Link**

D2D Communication with a Device-Controlled (DC-DC) Link enables devices in close proximity to establish direct communication links autonomously, bypassing traditional routing through a central base station or network operator. It allows nearby devices to communicate directly, resulting in lower latency, greater spectral efficiency, and less burden on the central network infrastructure, as illustrated in Figure 2.4. Device-controlled link establishment gives devices the autonomy to initiate, manage, and terminate these D2D links based on their requirements and prevailing network conditions. DC-DC link establishment is particularly relevant when direct and rapid communication is crucial, such as in vehicular networks for sharing traffic updates or safety alerts, drone swarms for coordinated movement, and IoT networks for real-time data exchange.

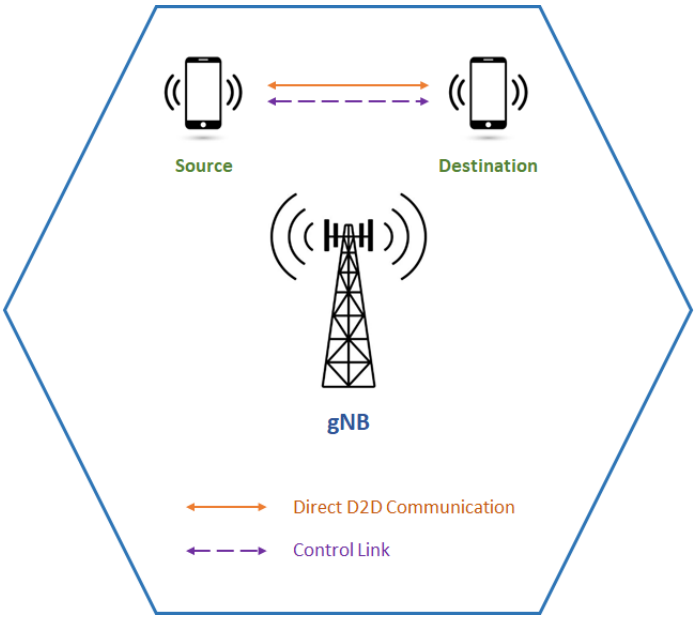

Figure 2.4: DC-DC Link

2.2.3 D2D Communication Classification

As Figure 2.5 shows, in general, there are two types of D2D communications [20].

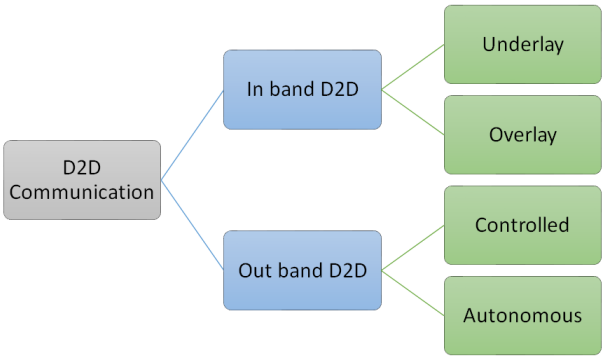

Figure 2.5: Classifications of D2D Communication

- **In-band D2D Communication**

In-band D2D communication refers to a form of communication where devices directly communicate using the same frequency band as cellular communication. Typically, the high control over the cellular (i.e., licensed) spectrum is the catalyst for adopting in-band communication. Inband communication can be further classified as underlay or overlay. Cellular and D2D communications use the same radio resources for underlay D2D communication. Whereas in overlay communication, D2D links are allocated dedicated cellular resources. By reusing spectrum resources (i.e., underlay) or allocating dedicated cellular resources to D2D users that accommodate a direct connection between the transmitter and the receiver (i.e., overlay), inband D2D can enhance the spectrum efficiency of cellular networks. Managing interference between D2D and cellular communications, especially when operating on the same frequency band, is complex. This interference can be mitigated by implementing resource allocation methods with a high degree of complexity, which increases the computational burden of BS or D2D users.

- **Out-band D2D Communication**

Out-band D2D communication refers to a type of D2D communication where the devices communicate directly using a different frequency band than the one used for traditional cellular communication. It can be further classified as controlled or autonomous. The devices might utilize unlicensed frequency bands (like the ones

used for Wi-Fi or Bluetooth) or licensed bands different from the cellular band to communicate with each other directly. The key idea is to offload traffic from the primary (cellular) band, thereby improving spectral efficiency and capacity.

Compared to in-band D2D, out-band D2D has its advantages and challenges. On the one hand, out-band D2D can mitigate interference with the cellular network because it uses a different frequency band. On the other hand, using an unlicensed spectrum brings challenges of its own, such as coexistence with other wireless systems using the same spectrum (like Wi-Fi or Bluetooth), and the quality of service may not be as robust due to the potential for interference from other devices.

2.2.4 Challenges in D2D Communication

With the evolution of 5G and the advent of 6G, the relevance of D2D communication with device-controlled link establishment is expected to increase dramatically. Autonomous and efficient link management will become increasingly critical as more devices join the network and move towards a truly interconnected world. However, there are still numerous complex challenges to resolve in 5G and beyond networks, particularly in physical layer characteristics, network architectures, and performance requirements. Interference Management is one of the most significant barriers to D2D communication. D2D communication may enhance spectral efficiency if interference caused by D2D is minimized and minimal QoS is ensured [21].

2.3 Sum Rate Optimization for D2D Networks

An essential performance metric for D2D networks is the sum rate, or the total data rate supported by the network. The sum rate is the aggregate of the individual data rates of all active D2D links in the network. Sum-rate maximization aims to optimize the network's total throughput, or data rate. By maximizing the sum rate, the network can support more data traffic and provide a better user experience.

Maximizing the sum rate is a critical challenge in D2D network design, involving

complex considerations around interference management, power control, and resource allocation. One key challenge is maximizing the sum rate and maintaining user fairness. A network might achieve a high sum rate by favoring users with good channel conditions, but this could lead to poor service for other users. Also, the dynamic and decentralized nature of D2D networks complicates the sum-rate maximization problem. As devices move and network conditions change, the optimal settings for resource allocation, interference management, and power control can change rapidly.

2.3.1 Sum Rate Optimization Techniques

Sum-rate maximization in D2D networks involves sophisticated techniques that address several key factors:

1. **Resource Allocation:** Efficient allocation of resources such as frequency bands and power can significantly enhance the sum rate. Advanced algorithms are often used to dynamically allocate resources based on network conditions and data traffic requirements.
2. **Interference Management:** In D2D networks, device interference can significantly degrade the network's performance. Techniques such as interference alignment, cancellation, and avoidance can help manage interference and improve the sum rate.
3. **Power Control:** Power control refers to adjusting the transmission power of devices to balance achieving good link quality and minimizing interference from other devices. Optimal power control is critical to achieving the maximum sum rate.

A list of research studies on the sum rate optimization problem for D2D communication has been presented here, starting with the authors names and then the focus and coverage of the research.

1. **Naderializadeh, Navid and Avestimehr, A. Salman** [22]
   The paper explores the issue of spectrum sharing in device-to-device commu-

nication systems, introducing a novel concept called the Information-Theoretic Independent Set (ITIS). Based on treating interference as noise, ITIS represents a subset of users who can theoretically transmit data simultaneously within a wireless network. This concept forms the basis for their innovative Information-Theoretic Link scheduling (ITLinQ) spectrum-sharing scheme. ITLinQ schedules simultaneous data transmission for users in a single ITIS during each time slot. In analyzing ITLinQ's capacity, the researchers set a lower boundary on the proportion of the information-theoretic capacity region it could achieve, factoring in a specific gap in a network model with randomly distributed nodes. However, ITLinQ is heuristic. This means that while it aims to approximate the globally optimal solution, it does not always guarantee the attainment of the global optimum. Moreover, depending on the implementation, the computational complexity of ITLinQ can still be substantial, especially for large-scale networks.

2. **F. Hussain, M. Y. Hassan, M. S. Hossen, and S. Choudhury** [23]
   In a cellular network featuring D2D communication, an optimization challenge arises when sharing Resource Blocks (RBs) between D2D pairs and cellular user devices to increase the system's total data transfer rate, or sum rate. It has been discovered that sharing does not always improve the sum rate and can sometimes reduce it. To address this, a new algorithm was developed based on weighted bipartite matching, which prevents rate decline and optimizes the sharing process for the maximum sum rate. However, the objective of a weighted bipartite matching algorithm is to find an optimal one-to-one mapping based on the given weights. This might not always translate to the global maximum sum rate, especially when considering interference and other complex network dynamics.

3. **Lin, Shijun and Fu, Liqun and Li, Kewei and Li, Yong** [24]
   The paper explores optimizing the sum rate in D2D communication within cellular networks, where numerous Cellular Users (CUs) share uplink resources with D2D pairs. The authors present system sum-rate maximization as an NP-hard mixed integer non-linear programming problem. They tackle this by

employing optimization decomposition. 1) Given a specific resource allocation policy, they deduce the optimal Signal-to-Interference plus Noise Ratio (SINR) threshold that elevates the system sum rate. 2) They propose a coalition game methodology to enhance the resource allocation policy further, demonstrating that this strategy can reach a Nash-stable partition in a finite time. Simulation results exhibit that the coalition game's performance closely matches the exhaustive search but with a significantly reduced runtime. However, optimization decomposition and coalition games are computationally intensive due to iterative processes. They may not converge quickly in dynamic settings, and stable coalition structures are not always guaranteed, especially in complex networks.

2.3.2 Traditional Methods for Sum Rate Optimization of D2D Networks

Traditional, i.e., non-data-driven, methods for optimizing the sum rate in D2D networks often involve mathematical optimization techniques. Several traditional optimization methods have been used to optimize the sum rate for D2D networks, as follows:

- **Convex Optimization Techniques:** In some situations, the problem of sum rate optimization in D2D networks can be formulated as a convex optimization problem. Convex optimization techniques can then ensure convergence to a global optimum. For instance, these techniques often involve linear programming, quadratic programming, or geometric programming.

- **Graph Theory Approaches:** In this case, the problem of sum rate optimization is modeled as a graph, where vertices represent users and edges represent possible D2D links. Techniques from graph theory, such as maximum matching or minimum cut, can then be used to optimize the sum rate.

- **Game Theory Approaches:** These approaches model the interactions between D2D pairs and between D2D pairs and cellular users as a game, where

each player aims to maximize their data rate. The Nash equilibrium of the game represents a stable state where no player can increase its data rate by unilaterally changing its strategy.

- **Heuristic Algorithms:** When the sum rate optimization problem is too complex or does not have a precise mathematical formulation, heuristic algorithms such as genetic algorithms, swarm optimization, or simulated annealing can be used. These algorithms generate satisfactory, although not always optimal, solutions to the problem.

Below is a list of studies on the sum rate optimization problem for D2D communication using traditional methods, starting with the authors names and then the focus and coverage of the research.

1. **Chiang, Mung and Tan, Chee Wei and Palomar, Daniel P. and O'neill, Daniel and Julian, David** [25]
   In wireless networks where interference impacts QoS, power control issues can be presented as nonlinear optimization problems. These aim to maximize system or user throughput while adhering to individual QoS constraints. The study showed that in high Signal-to-Interference Ratio (SIR) environments, these complex problems can be transformed into geometric programming, facilitating effective solutions even with numerous users. Researchers proposed a systematic method for power control based on geometric programming-based distributed algorithms, showing its implications through numerical examples. However, these algorithms depend on how accurate and good the information about the channel state is. In dynamic environments with fluctuating channel conditions, achieving optimal solutions can be challenging due to potential delays and inaccuracies in information exchange, leading to suboptimal power allocations and reduced sum rate performance.

2. **Qian, Li Ping and Zhang, Ying Jun and Huang, Jianwei** [26]
   Achieving Weighted Throughput Maximization (WTM) through power control in wireless networks with interference has been a complex challenge due to the non-convex optimization problem that arises from link interference. This paper

introduces a MAPEL algorithm that converges to an optimal solution for the WTM problem in the general SINR regime. MAPEL identifies the optimal power control solution by creating increasingly precise approximations of the feasible SINR region. Adjusting the approximation factor also allows a trade-off between optimality and convergence time. However, this algorithm suffers from high computational complexity, which limits its practicality to small scenarios only. MAPEL served as a benchmark for evaluating other algorithms for the same problem and was used in this study to assess the performance of several existing algorithms through extensive simulations.

3. **Liu, Liang and Zhang, Rui and Chua, Kee-Chaing** [27]
   Identifying the highest possible value of the Weighted Sum Rate (WSR) for a K-user Gaussian Interference Channel (GIC) is critical in wireless communication. However, conventional convex optimization strategies struggle to meet this challenge due to the mutual interference between users, causing the problem to be non-convex. The study introduces a novel approach, merging monotonic optimization and rate profile strategies, to discover the best power control and beamforming solutions for maximizing the WSR. The proposed framework can be employed in GICs equipped with single-antenna transmitters and single- or multi-antenna receivers (SISO or SIMO) or multi-antenna transmitters and single-antenna receivers (MISO). Uniquely, this study aims to maximize the WSR within the GIC's achievable rate region directly, leveraging the understanding that this region is a "normal" set and the users' WSR steadily rises. Hence, WSR maximization is recognized as a monotonic optimization over a normal set, solvable globally through the existing outer polyblock approximation algorithm. Numerical evidence underpins the suggested algorithms' capability to achieve the highest WSR for SISO, SIMO, or MISO GIC, setting a performance standard for other heuristic algorithms. However, a fundamental limitation of merging monotonic optimization and rate profile strategies for power control and beamforming solutions is that they often require precise and up-to-date channel state information. Any mismatch or delay in acquiring this information can lead to suboptimal beamforming vectors and power levels, potentially compromising the WSR maximization objective.

2.3.3 Limitations of Conventional Methods for Sum Rate Optimization of D2D Networks

Conventional methods primarily involve traditional mathematical optimization or heuristic-based algorithms. However, the main limitation associated with these methods is their requirement for perfect Channel State Information (CSI). Moreover, these methods can be computationally expensive, posing significant challenges in their practical implementation. Traditional approaches often struggle with the dynamism and nonlinearity of wireless communication environments.

Dynamism refers to the continually changing aspects of the environment, such as the number of active devices, their location, signal strength, and available spectrum. Conversely, nonlinearity refers to the complex and often unpredictable interactions between signals as they propagate through the environment. Both of these factors significantly influence the performance of wireless communication systems.

Dynamism impacts the wireless communication environment in several ways. For instance, devices may frequently join or leave the network, leading to variable traffic. Mobile devices can move in and out of coverage areas, creating fluctuations in signal strength. The available spectrum for wireless transmission may also vary depending on the congestion in the network, leading to unpredictable changes in the network capacity.

Nonlinearity manifests in multipath propagation, where a signal may travel along multiple paths before reaching its destination, creating signal interference and distortion. The complex topography and the presence of obstacles also lead to unpredictable signal behaviors, contributing to the nonlinearity of the wireless environment.

Traditional methods for sum rate optimization have several limitations in addressing the dynamism and nonlinearity inherent in wireless environments.

- **Static Algorithms:** Many traditional sum rate optimization algorithms are static, meaning they are not designed to adapt to the dynamic changes in the wireless environment. They may work well in a static scenario but fail to deliver

optimal results in real-world, dynamic conditions.

- **Limited Capabilities for Nonlinearity Management:** Traditional methods often rely on linear mathematical models that may struggle to capture the complexity and unpredictability of the wireless environment.
- **Assumption of Ideal Conditions:** Traditional algorithms often make assumptions about the wireless environment, such as perfect channel knowledge or the absence of interference. These assumptions often do not hold in real-world scenarios, leading to suboptimal performance.
- **High Computational Complexity:** Traditional methods can have high computational complexity, making them unsuitable for real-time applications requiring quick decisions.

These attributes, which result from user mobility, signal variations, and interference from other devices, significantly hinder the effectiveness of traditional sum rate optimization methods.

Emerging technologies such as Artificial Intelligence (AI) and Machine Learning (ML) provide potential solutions to these challenges. They can help design intelligent algorithms that dynamically adjust network parameters to maximize the sum rate. They can also enable devices to learn from past experiences and make better resource allocation and power control decisions.

2.4 Machine Learning (ML) for Optimized Sum Rate in D2D Networks

ML is a subset of AI that uses statistical techniques to enable machines to improve at tasks with experience. With its ability to learn from data and improve over time, machine learning can offer effective solutions to the sum-rate maximization problem. The complexity of D2D networks, characterized by many devices and rapidly changing network conditions, makes ML a suitable approach for tackling challenges such as

power control, resource allocation, and interference management, which are critical for maximizing the sum rate. Several machine learning techniques have been proposed for sum-rate maximization in D2D networks, each with strengths and limitations.

2.4.1 Supervised Learning (SL)

SL is a type of machine learning where a model is trained using labeled data. Labeled data refers to datasets with input data and corresponding desired output, often called "labels" or "targets." Supervised learning aims to develop a model that, given input data, can accurately predict the corresponding output.

In many supervised learning tasks, such as regression problems, the model's performance is evaluated using a loss or cost function. One commonly used loss function is the Mean Squared Error (MSE). The aim is to minimize this MSE to improve the model's accuracy.

The MSE is the average of the squared differences between the actual output (label) and the predicted output (prediction from the model). The objective of supervised learning, therefore, is to adjust the model's parameters to minimize the MSE. It is often done using optimization algorithms like Gradient Descent.

However, to minimize the MSE, labeled data is crucial in supervised learning for the following reasons for training the model and minimizing the mean squared error:

- **Model Training:** The model learns the relationship between input data and output (labels) during training. In other words, it adjusts its parameters so that the predicted output is as close as possible to the actual output. Without labels, the model has no way of knowing what output to aim for given the input data.
- **Performance Evaluation:** The MSE requires both the predicted output (from the model) and the actual output (from the label) to be computed. Without the actual output, there is no way to calculate the MSE and, thus, no way to evaluate or improve the model's performance.

- **Model Adjustment:** The minimization of MSE guides adjusting the model's parameters during the learning process. The gradient of the MSE for the model's parameters provides a "direction" in which to adjust the parameters to reduce the MSE. Without labeled data, this gradient cannot be calculated.

2.4.2 Unsupervised Learning (UL)

UL is a type of machine learning where a model learns to identify patterns and structure in the input data without any labeled outputs to guide the learning process. The primary goal here is to model the underlying structure or distribution of the data in order to learn more about it.

Unlike supervised learning, which minimizes the discrepancy between predicted and actual labels (such as in MSE), unsupervised learning seeks to minimize an objective function defined directly on the input data. This objective function typically quantifies the model's ability to capture the underlying patterns or structures in the input data.

For instance, in a clustering problem (a common unsupervised learning task), the objective might be to minimize the intra-cluster distances (i.e., ensure that points within the same cluster are close to each other) while maximizing the inter-cluster distances (i.e., ensure that points from different clusters are far from each other). No labeled data is needed; the model looks for natural groupings in the input data.

Another common unsupervised learning method is dimensionality reduction, such as Principal Component Analysis (PCA), which aims to minimize information loss when representing high-dimensional data in a lower-dimensional space.

This approach somewhat resembles "self-supervised" learning, a relatively new paradigm in machine learning. In self-supervised learning, the model generates its labels from the input data, effectively creating a supervised learning problem out of an unsupervised one. For instance, a model might be trained to predict the next word in a sentence (the label) given the previous words (the input data).

However, in wireless communications literature, "unsupervised learning" describes any learning approach that does not require labeled data. It is because, in many wireless scenarios, it is not practical or possible to obtain labeled data. In these cases, unsupervised learning methods that can work directly with the raw data, extracting meaningful insights and improving system performance, become particularly valuable.

2.4.3 Reinforcement Learning (RL)

RL involves an agent learning to make decisions by interacting with its environment to maximize cumulative reward. In the D2D network context, an RL agent could represent a network controller or individual devices, and the environment would be the D2D network. The RL agent learns the optimal actions, such as power control or resource allocation decisions, that maximize the sum rate, thereby serving as the cumulative reward.

There are various ways reinforcement learning techniques have been utilized for sum-rate maximization in D2D networks:

- **Q-Learning:** A simple yet powerful RL algorithm, Q-learning can be applied to learn the optimal policy for power control or resource allocation in D2D networks. The Q-values, representing the expected future reward of taking a certain action in a given state, guide the decision-making process.
- **Multi-Agent Reinforcement Learning (MARL):** In large D2D networks, multiple devices must make coordinated decisions to maximize the sum rate. MARL, where multiple RL agents learn to cooperate or compete with each other, provides a framework for learning such coordinated strategies.

While RL holds great promise for sum-rate maximization in D2D networks, several challenges must be addressed. These include RL algorithms' instability and slow convergence, the exploration-exploitation dilemma, and the complexity of coordinating multiple agents in MARL settings. Furthermore, real-world implementation of RL in D2D networks requires addressing issues like communication overhead and privacy

concerns.

However, advances in RL research and the computational capabilities of wireless devices provide optimistic prospects for overcoming these challenges. Techniques like experience replay and target networks can improve the stability and convergence of RL, while strategies like epsilon-greedy and Thompson sampling can balance exploration and exploitation. Furthermore, paradigms like federated learning can address privacy concerns by allowing devices to learn while keeping their data local.

2.4.4 Deep Learning (DL)

DL is a subfield of machine learning based on artificial neural networks in which multiple processing layers extract progressively higher-level features from data. The term "deep" in deep learning signifies the number of layers that the data undergoes through its transformation process. When neural networks have many layers, they are called Deep Neural Networks (DNNs). They are more complex and powerful, as they can process data in more ways and create more nuanced understandings of the input.

The increase in data and computing power powers deep learning algorithms and has numerous applications, including automatic speech recognition, natural language processing, sound recognition, machine translation, image processing, material inspection, social network filtering, bioinformatics, drug design, and board game programs, where deep learning systems have demonstrated superior performance.

Deep Learning models can learn intricate patterns in high-dimensional data, making them well-suited for the complexity and variability of D2D networks. By processing historical network data, these models can predict optimal configurations, like resource allocation and power control, which can be used to maximize the sum rate.

Several deep learning techniques are being applied for sum-rate maximization in D2D networks:

- **Deep Neural Networks (DNNs):** DNNs, with their ability to model complex

non-linear relationships, can be used to predict the sum rate based on network parameters. Given the predicted sum rate, optimization techniques can then be used to find the parameter settings that maximize the sum rate.

- **Convolutional Neural Networks (CNNs):** CNNs, known for their excellence in processing grid-like data (e.g., image and signal processing), can process spatial network data. They can assist in tasks like device clustering or interference prediction, contributing to sum-rate maximization.

- **Deep Reinforcement Learning (DRL):** DRL combines the power of deep learning and reinforcement learning, using deep learning models to predict the Q-values in reinforcement learning. DRL can adapt to dynamic network conditions and continuously optimize the sum rate.

Despite the promising potential of deep learning for sum-rate maximization in D2D networks, certain challenges exist. Training deep learning models requires a substantial amount of labeled data and computational resources, and the black-box nature of these models can lead to interpretability issues. However, advances in machine learning research and the growing computational power of wireless devices are expected to mitigate these challenges. Transfer learning and data augmentation can help train models with fewer data points, while model compression and hardware acceleration can alleviate computational concerns. Additionally, developments in explainable AI can enhance model interpretability.

A list of works that have been done on the sum rate optimization problem for D2D communication using deep learning methods is displayed here, starting with the authors names and then the focus and coverage of the research.

1. **Sun, Haoran and Chen, Xiangyi and Shi, Qingjiang and Hong, Mingyi and Fu, Xiao and Sidiropoulos, Nicholas D.** [28]
   The paper presents a new approach to addressing the challenges posed by numerical optimization in key Signal Processing (SP) applications such as communications, radar, filter design, and speech and image analytics. The complexity of optimization algorithms often creates a disconnect between theory

and real-time processing. To mitigate this, the authors propose using a DNN to approximate the unknown nonlinear mapping between the input and output of an SP algorithm. Initially, the paper identifies a subset of optimization algorithms that a fully connected DNN can effectively approximate. The authors then apply this approach to the widely used interference management algorithm, Weighted Minimum Mean Square Error (WMMSE), demonstrating its effectiveness. Tests conducted with synthetically generated wireless channel data and actual Digital Subscriber Line (DSL) channel data revealed that a relatively small network could achieve high approximation accuracy and could significantly speed up computation times. However, the main problem is that it depends on the quality and variety of the training data, and any difference can lead to less-than-optimal approximations.

2. **Lee, Woongsup and Kim, Minhoe and Cho, Dong-Ho** [29]

   This letter introduces Deep Power Control (DPC), the first framework for controlling transmit power based on a CNN. In DPC, the strategy for adjusting transmit power to optimize either Spectral Efficiency (SE) or Energy Efficiency (EE) is learned using a CNN. Unlike traditional power control schemes that require a substantial number of calculations, DPC allows for determining users' transmit power with significantly fewer computations, making real-time processing possible. It also presents a variant of DPC that can be implemented in a decentralized fashion using local channel state information, drastically reducing the signaling overhead. Simulation results indicate that DPC can achieve nearly the same or even superior SE and EE as traditional power control methods but with considerably less computation time. However, a significant limitation is the dependency on extensive training and accurate data. Additionally, CNNs can be computationally intensive, potentially posing challenges for real-time applications in dynamic environments.

3. **Ron, Dara and Lee, Jung-Ryun** [30]

   This paper addresses interference arising from concurrent frequency band usage by Device-to-Device User Equipment (DUE) and Cellular User Equipment (CUE) in scenarios where cellular uplinks underlay D2D communication. This

interference can be mitigated by optimizing the transmit power of the devices, which is typically modeled as a complex NP-hard combinatorial optimization problem with linear constraints. Conventional optimization methods are generally unable to solve this problem effectively. The authors introduce a DRL algorithm to optimize the transmit power for both DUEs and CUEs within the context of D2D communication underlaid by uplink cellular networks. This algorithm effectively resolves the optimization problem by combining optimal decision-making with efficient deep network training. The proposed algorithm offers a solution near the global optimum with significantly lower computational complexity than an exhaustive search, such as fixed-power, WMMSE, gradient descent, and Newton-Raphson approaches. However, a key limitation is the need for substantial training in varied scenarios. DRL can be sensitive to its training environment, and suboptimal training can lead to inefficient power allocation decisions in unencountered scenarios, potentially causing interference and subpar network performance.

2.4.5 Deep Unsupervised Learning (DUL)

Unsupervised deep learning or DUL, as an emerging field, has shown promise in addressing the shortcomings of traditional methods. The potential of DUL methods for understanding the wireless environment's complexities without needing labeled data sets has made them effective. Several studies have specifically looked into applying DUL to D2D networks. These methods are adept at handling the dynamism and nonlinearity of wireless environments. Their study showed a significant improvement in the sum rate compared to traditional methods.

A listing of research studies on the sum rate optimization problem for D2D communication utilizing deep unsupervised learning is provided below, starting with the authors names and then the focus and coverage of the research.

1. **Liang, Fei and Shen, Cong and Yu, Wei and Wu, Feng** [31]

   A new power control strategy based on DNNs is proposed to maximize the sum

rate of a multi-user interference channel, addressing a non-convex optimization problem. The solution, PCNet, is a specifically designed neural network trained via unsupervised learning. PCNet(+) enhances its generalization by using noise power as an input. A further improvement, ePCNet(+), uses multiple independently trained PCNets, outperforming single PCNet(+) and other existing solutions, as per simulation results. Nevertheless, it offers a flexible boundary, necessitates the adjustment of a penalty parameter, and frequently does not guarantee the best or most feasible solution.

2. **Lea, Benjamin and Shome, Debaditya and Waqar, Omer and Tomal, Jabed** [32]

   The study involves a system model integrating energy-harvesting drones with D2D networks. The aim is to develop an optimal power transmission vector, maximizing the D2D network's sum rate while satisfying the drones' energy needs. Given the need for real-time solutions, a hybrid approach is proposed, combining deep unsupervised learning with a comprehensive power scheme via a deep neural network. The hybrid method outperforms non-data-driven methods by up to 91% in sum rate, achieving efficient solutions within the channel coherence time. However, it also suffers from the issue of a "soft" boundary, requires tuning a penalty factor, and often does ensure neither optimality nor feasibility.

3. **Kim, Donghyeon and Jung, Haejoon and Lee, In-Ho** [33]

   Managing transmit power is crucial for effectively reducing interference and increasing the sum rate in overlay D2D communication systems. However, such power control for optimizing the sum rate is an NP-hard problem typically addressed using iterative algorithms like the WMMSE method, which is complex and time-consuming. The authors introduce an unsupervised learning-based deep learning power control scheme that considers partial and outdated CSI to address these challenges. This scheme uses a DNN to formulate an optimization problem to maximize spectral efficiency while considering user fairness and energy efficiency constraints. Moreover, the authors propose a method for reporting CSI based on the channel-to-interference power ratio that significantly

reduces the feedback overhead. Simulation results indicate improved spectral efficiency, energy efficiency, and fairness performance across varying topographical sizes and channel correlation coefficients. However, the major limitation is the potential for suboptimal decision-making. Relying on incomplete or outdated CSI can lead to inaccurate power control decisions, resulting in increased interference and a compromised sum rate, negating the very goals of the scheme.

2.5 Conclusion

The reviewed literature suggests that conventional methods struggle with large-scale optimization due to computational intensity [22]-[27]. DNNs can efficiently handle these problems due to their function approximation abilities [34]. However, ensuring feasible solutions with NNs in constrained optimization, especially with intertwined constraints, is challenging. One strategy for constraints in deep learning is using a penalty term in the loss function, especially for D2D networks [31]. While this offers flexibility, it gives only a "soft" boundary, requiring an additional penalty factor to balance optimality and feasibility, often failing to achieve either. Projection-based methods [11] provide "hard" boundaries but increase computational complexity and often do not yield optimal solutions. Some recent methods ensure feasibility using iterative processes like the double description method and the gradient-descent-based DC3 algorithm [14]. However, these iterative steps contradict the primary advantage of using deep learning over traditional iterative algorithms. This implies that more extensive research and validation are needed to conclusively establish these methods' practicality and effectiveness. This research, therefore, introduces a new DUL-based framework designed to consistently produce feasible solutions for optimization problems with non-homogeneous linear inequality constraints combined with box constraints. The study further showcases its efficacy in optimizing the sum rate in D2D networks.

In the next chapter, the methodology of the study will be discussed in detail, laying the foundation for the experimental setup and analysis of the results.

Chapter 3

Methodology

This chapter delineates the methodology implemented in this research study. It begins by introducing the system model, research design and discussing the selection of the DUL model. This is followed by explaining the proposed network model, the data generation methods, and the evaluation metrics used to assess the model's performance.

3.1 The System Model

For a standard K-user single-antenna interference channel model, as shown in Figure 3.1, where all transmitter-receiver pairs are considered to share the same narrowband spectrum and to be synchronized, the discrete-time baseband signal received by the i-th receiver is:

$$y_i = h_{i,i}x_i + \sum_{j \in K/\{i\}} h_{j,i}x_j + n_i \tag{3.1}$$

Here, x_i = the signal transmitted by the i-th transmitter; $K = \{1, 2, \ldots, K\}$ = the set of transmitter-receiver pairs; $K/\{i\}$ = the set of transmitter-receiver pairs excluding the i-th one; $h_{i,i}$ = the direct link channel for the i-th user; $h_{j,i}$ = the cross-link channel for the j-th transmitter and the i-th receiver; n_i = the i-th receiver noise and $n_i \sim N(0, \sigma_i^2)$. Also, $x_i \in C$, $h_{i,i} \in C$ and $h_{j,i} \in C$ when C = the set of

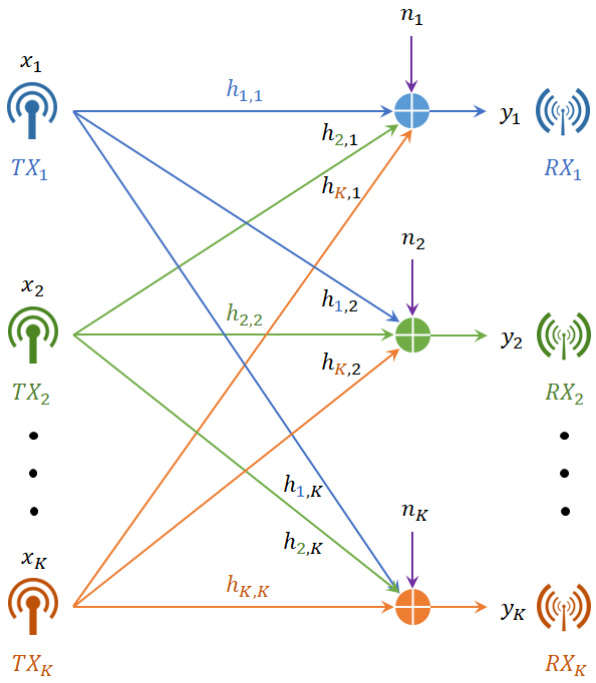

Figure 3.1: The K-user interference channel

complex numbers. Moreover, $h_{j,i}$ follows the Circularly Symmetric Complex Gaussian (CSCG) distributions with zero means and unit variances.

It is relevant to point out that this model has been thoroughly explored in studies [26, 35, 36, 28, 31].

Equation (3.1) can be written in a matrix form as:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_K \end{pmatrix}_{K\times 1} = \begin{pmatrix} h_{1,1} & h_{2,1} & \dots & h_{K,1} \\ h_{1,2} & h_{2,2} & \dots & h_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ h_{1,K} & h_{2,K} & \dots & h_{K,K} \end{pmatrix}_{K\times K} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix}_{K\times 1} + \begin{pmatrix} n_1 \\ n_2 \\ \vdots \\ n_K \end{pmatrix}_{K\times 1} \tag{3.2}$$

For a given vector $\mathbf{P} = (P_1, P_2, \dots, P_K)^T$ = the joint power profile of all users, P_i = the transmit power for i-th user when $(0 \leq P_i \leq P_{max})$, and the channel realization $\{h_{ij}\}_{i,j\epsilon K}$, the achieveable rate of the i-th receiver under Gaussian codebooks is:

$$R_i(\mathbf{P}) = \log_2 \left(1 + \frac{P_i |h_{i,i}|^2}{\sigma_i^2 + \sum_{j\in K/\{i\}} P_j |h_{j,i}|^2} \right) \tag{3.3}$$

Then the SINR is:

$$SINR_i(\mathbf{P}) = \frac{P_i|h_{i,i}|^2}{\sigma_i^2 + \sum_{j \in K/\{i\}} P_j|h_{j,i}|^2} \tag{3.4}$$

3.1.1 Problem Formulation

Power control in interference management aims to identify the ideal power profile $\mathbf{P}$ for all users to enhance system performance, i.e., SINR. It needs to be done while adhering to certain specified restrictions, as detailed below:

$$\begin{aligned} &\underset{\mathbf{P}}{maximize} \sum_{i=1}^{K} R_i(\mathbf{P}) \\ &\text{subject to } SINR_i(\mathbf{P}) \geq r_{i,min} \\ &\text{and } (0 \leq P_i \leq P_{max}) \end{aligned} \tag{3.5}$$

Here, $r_{i,min}$ = the minimum required SINR of the i-th receiver and $r_{min} = (r_{1,min}, r_{2,min}, \ldots, r_{K,min})$.

3.1.2 Constraint Elimination

Typically, DNNs parameters are unbounded and can take on random values in the entire real space. However, for wireless communications optimization problems, the optimization variables are susceptible to numerous constraints. Eliminating constraints and transforming constrained optimization issues into unconstrained problems is a natural application of the proposed DNNs with constrained variables.

From Equation (3.5):

$$SINR_i(\mathbf{P}) \geq r_{i,min} \tag{3.6}$$

Now, from Equation (3.4):

$$\frac{P_i|h_{i,i}|^2}{\sigma_i^2 + \sum_{j \in K/\{i\}} P_j|h_{j,i}|^2} \geq r_{i,min} \tag{3.7}$$

This Equation (3.7) can be reorganised as:

$$P_i|h_{i,i}|^2 - r_{i,min} \sum_{j \in K/\{i\}} P_j|h_{j,i}|^2 \geq \left(r_{i,min} \times \sigma_i^2\right) \tag{3.8}$$

So, the equation in matrix form becomes:

$$\begin{pmatrix} |h_{1,1}|^2 & -r_{1,min}|h_{2,1}|^2 & \dots & -r_{1,min}|h_{K,1}|^2 \\ -r_{2,min}|h_{1,2}|^2 & |h_{2,2}|^2 & \dots & -r_{2,min}|h_{K,2}|^2 \\ \vdots & \vdots & \ddots & \vdots \\ -r_{K,min}|h_{1,K}|^2 & -r_{K,min}|h_{2,K}|^2 & \dots & |h_{K,K}|^2 \end{pmatrix}_{K \times K} \times \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_K \end{pmatrix}_{K \times 1} \geq \begin{pmatrix} r_{1,min} \times \sigma_1^2 \\ r_{2,min} \times \sigma_2^2 \\ \vdots \\ r_{K,min} \times \sigma_K^2 \end{pmatrix}_{K \times 1} \tag{3.9}$$

Equation (3.9) can be expressed as:

$$A_{K \times K} \times \mathbf{P}_{K \times 1} \geq \mathbf{b}_{K \times 1} \tag{3.10}$$

Equation (3.10) satisfies conditions of linear inequality constraint and the transform is:

$$\mathbf{P} = A^{-1}(\mathbf{b} + \nu) \tag{3.11}$$

where, A^{-1} = the pseudo inverse of A and $\nu = e^{\nu'} > 0$ = the introduced set of slack variables.

3.1.3 Formulation for The Optimization Problem

For an optimization problem:

$$\begin{aligned} &\text{P1}: \min_{\mathbf{x}} \quad f(\mathbf{x}) \\ &\text{subject to } A\mathbf{x} \geq \mathbf{b} \\ &\text{and } \mathbf{0} \leq \mathbf{x} \leq \mathbf{c} \end{aligned} \tag{3.12}$$

Here, $\mathbf{x}$, $\mathbf{b}$, and $\mathbf{c}$ are K-dimensional real vectors, with non-negative entries; $f(\mathbf{x})$ is a real-valued (not necessarily convex) objective function; and A is a $K \times K$ real monotone matrix.

The non-homogeneous constraint given in Equation (3.12) can be rewritten as:

$$A\mathbf{x} = \mathbf{b} + \mu \tag{3.13}$$

Here $\mu \geq \mathbf{0}$ is a K-dimensional real vector. As a consequence,

$$\mathbf{x} = \widehat{\mathbf{x}} + A^{-1}\mu \tag{3.14}$$

Where,

$$\widehat{\mathbf{x}} = A^{-1}\mathbf{b} \tag{3.15}$$

It is worth mentioning that A is monotone, and, as such, all entries of A^{-1} are non-negative [37]. Consequently, $A^{-1}\mu \geq \mathbf{0}$, and $\mathbf{x} \geq \widehat{\mathbf{x}}$. Therefore, the problem P1 is infeasible if $\widehat{\mathbf{x}} \not\leq \mathbf{c}$. Furthermore, $\mathbf{b} \geq \mathbf{0}$ implies $\widehat{\mathbf{x}} \geq \mathbf{0}$. Hence, we can have $\mathbf{x} \geq \mathbf{0}$. As such, the lower-limit constraint in Equation (3.12) is satisfied implicitly.

In the following, the optimization problem P1 is transformed into an equivalent problem in terms of μ. To this end, it is observed that the P1 constraint of Equation (3.12) is equivalent to $\mu \geq \mathbf{0}$. Moreover, since the lower-limit constraint of Equation

(3.12) is always satisfied implicitly, it can be expressed in terms of μ as:

$$A^{-1}\mu \leq \mathbf{c} - \widehat{\mathbf{x}} \tag{3.16}$$

Consequently, P1 can be reformulated in terms of μ as follows:

$$\begin{aligned} &\text{P2}: \min_{\mu} \quad f(\widehat{\mathbf{x}} + A^{-1}\mu) \\ &\text{subject to } \mu \geq \mathbf{0} \\ &\text{and } A^{-1}\mu \leq \mathbf{c} - \widehat{\mathbf{x}} \end{aligned} \tag{3.17}$$

Although it is quite straightforward to satisfy constraint of P2 of Equation (3.17) in DNNs by employing ReLU activation function, meeting constraint of Equation (3.12) is non-trivial. In what follows, a method that guarantees satisfaction of both constraints of P2 simultaneously by the DNNs is proposed. To this end, it can be written $A^{-1} = \begin{bmatrix} \mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_K \end{bmatrix}$, where $\mathbf{a}_k$ is the k-th column of A^{-1}, and $k = 1, 2, \cdots, K$. Denoting the k-th element of μ by μ_k, Equation (3.17) can be re-written as:

$$\sum_{k=1}^{K} \mu_k \mathbf{a}_k \leq \mathbf{c} - \widehat{\mathbf{x}} \tag{3.18}$$

Equivalently,

$$\mu_k \mathbf{a}_k \leq \mathbf{c} - \widehat{\mathbf{x}} - \sum_{l=1, l\neq k}^{K} \mu_l \mathbf{a}_l \tag{3.19}$$

Noting that $\mu_k \mathbf{a}_k \geq 0$ for $1 \leq k \leq K$, it comes:

$$\mu_k \mathbf{a}_k \leq \mathbf{c} - \widehat{\mathbf{x}} \tag{3.20}$$

and

$$0 \leq \mu_k \leq \beta_k \triangleq \min_{l} \frac{c_l - \widehat{x}_l}{a_{k,l}} \tag{3.21}$$

Where c_l, $\widehat{x}_l$, and $a_{k,l}$ are the l-th elements of $\mathbf{c}$, $\widehat{\mathbf{x}}$, and $\mathbf{a}_k$, respectively, and $k, l = 1, 2, \cdots, K$.

Here, the lower limit on μ_k comes directly from Equation (3.17). To point out that Equation (3.21) is a box constraint and can be easily implemented using a sigmoid activation function in DNNs. It is important to mention here that Equation (3.21) must be satisfied for Equation (3.17) to hold. However, the converse is not true in general. Nonetheless, Equation (3.21) provides a tight boundary box of the convex polytope represented by Equation (3.17).

Since Equation (3.21) does not imply Equation (3.17) in all cases, it is proposed to scale μ by $\alpha \geq 0$, if Equation (3.17) is not satisfied. To this end, it is defined $\nu = \alpha\mu$, where $\alpha = 1$ if Equation (3.17) is satisfied. Otherwise, α is computed using Equation (3.17) as:

$$\alpha = \begin{cases} 1 & \text{if Equation (3.17) is satisfied;} \\ \min\limits_{l} \dfrac{c_l - \widehat{x}_l}{[A^{-1}\mu]_l} & \text{otherwise;} \end{cases} \tag{3.22}$$

and $[A^{-1}\mu]_l$ is the l-th element of $A^{-1}\mu$ with $l = 1, 2, \cdots, K$. By employing Equation (3.22) it is ensured that $A^{-1}\nu \leq \mathbf{c} - \widehat{\mathbf{x}}$, thus 100% constraint satisfaction is ensured. For noting that $\min\limits_{l} \dfrac{c_l - \widehat{x}_l}{[A^{-1}\mu]_l}$ cannot be less than unity when Equation (3.17) is satisfied, it can be re-writen α in a more compact form as:

$$\alpha = \min\left\{1, \min_{l} \frac{c_l - \widehat{x}_l}{[A^{-1}\mu]_l}\right\} \tag{3.23}$$

Regarding the Equation (3.5), α can be expressed as:

$$\alpha = \min\left\{1, \min_{1 \leq l \leq K} \frac{P_{max} - \widehat{P}_l}{[A^{-1}\mu]_l}\right\} \tag{3.24}$$

3.2 Research Design

The research was executed following a quantitative approach to develop a prediction model that can estimate the optimal sum rate for D2D networks using DUL techniques. The study involved designing a deep learning model, training it on a

comprehensive dataset, and subsequently evaluating its performance.

3.2.1 Generating Feasible Datasets for the Transmission Channel Parameters

The data used for training, validation, and testing the model were generated from CSCG distributions with zero means and unit variances for various D2D network simulations. The variables included maximum transmission power, minimum SINR for each receiver, and the number of devices.

Matrix B defines the necessary and sufficient criteria for evaluating the viability of Equation (3.5) as [26, 38]:

$$B_{i,j} = \begin{cases} 0, & i = j \\ \frac{\gamma_{i,min}|h_{j,i}|^2}{|h_{i,i}|^2}, & i \neq j \end{cases} \tag{3.25}$$

Here, $B_{i,j}$ = the (i,j)-th element of B and $\gamma_{i,min} = r_{i,min}$ = the minimum SINR of the i-th receiver that is required to satisfy as constraint.

It is possible to locate a workable power allocation $\widehat{\mathbf{P}}$ in the following way if the maximum eigenvalue of B is less than 1:

$$\widehat{\mathbf{P}} = (\mathbf{I} - B)^{-1}\mathbf{u} \tag{3.26}$$

where, $\mathbf{I}$ is an $K \times K$ identity matrix and $\mathbf{u}$ is a $K\times$ 1 column vector with the i-th element u_i as:

$$u_i = \frac{\gamma_{i,min}\sigma_i^2}{|h_{i,i}|^2} \tag{3.27}$$

If every element in $\widehat{\mathbf{P}}$ falls between 0 and P_{max}, then the power profile $\widehat{\mathbf{P}}$ is a viable solution to the problem posed by equation Equation (3.5). However, there might be better options than this.

3.2.2 Proposed DNN Model

The choice of model for this study was influenced by the need for a system that could learn complex distributions from the data without requiring labeled examples to exploit a fully connected deep neural network to address the power control problem.

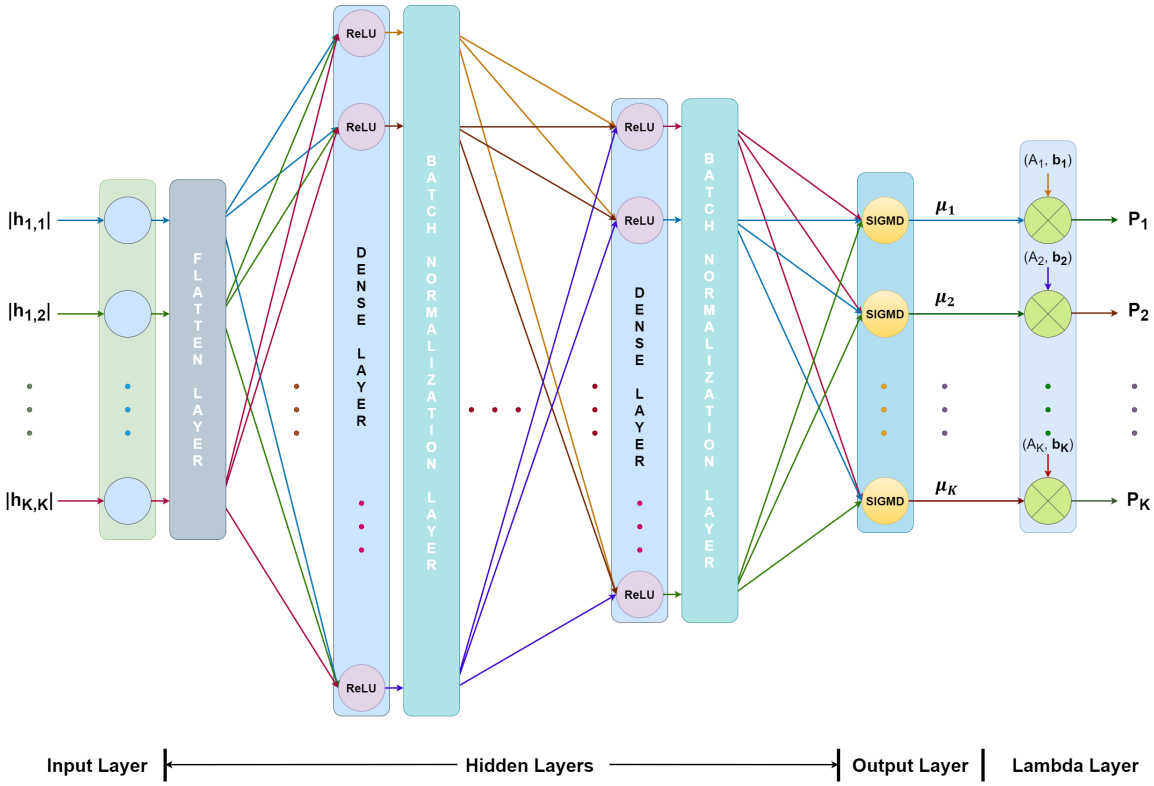

Figure 3.2: The architecture of the proposed DNN model

- **DNN Architecture**

Using Equation (3.17) and Equation (3.23), the DNN architecture as shown in Figure 3.2 is proposed to solve the optimization problem of Equation (3.5). The proposed network model comprises several layers: input, hidden (flattened, dense, batch normalization), and output.

The input layer of a neural network receives the feature vector, which is the feasible concatenation of the channel coefficients' magnitudes, $|h_{i,j}|$. The input layer has K^2 neurons corresponding to the elements of A or any bijective function related to A. Next are the densely interconnected hidden layers. A ReLU activation function is used

to calculate the output of each node in the concealed layer, which is then followed by a Batch Normalization (BN) layer. The addition of the BN layer expedites the training procedure. Then, the output layer is activated using the sigmoid function. The output layer has K neurons corresponding to μ with scaled sigmoid activation function corresponding to Equation (3.21). Finally, there is a **Lambda** layer to calculate the power profile $\mathbf{P}$ that satisfies both the SINR and power constraints of the optimization problem Equation (3.5) corresponding to Equation (3.22) using $\widehat{\mathbf{P}}$, A^{-1} and ν. The loss function is $f(\widehat{\mathbf{P}} + A^{-1}\nu)$, where $\nu = \alpha\mu$, and α is given in Equation (3.22).

$$\mathcal{L}\text{oss}_{\mathbf{DNN}} = \frac{1}{|\Psi|}\sum_{\psi\epsilon\Psi} -R_{\psi}\left(\mathbf{P}\right) \tag{3.28}$$

Here, ψ represents the mini-batch of size $|\Psi|$. The objective of the DNN is to minimize the loss function and maximize the sum rate $R(\mathbf{P})$, which would help the model recognize the correlations and dependencies between different network parameters and their effect on the sum rate. It is essential to mention that this training framework is known as UL because the objective function is directly incorporated into the loss function, eliminating the need for label data.

3.2.3 Evaluation Metrics

The performance of this proposed DUL-based DNN model was compared mainly based on two metrics: Constraint Violation Probability (CVP) or Hit Rate (HR), HR = 1 - CVP, and the Average Sum Rate (ASR) in Bit/Second/Hertz. However, the complexity, the model's robustness to channel noise, and the ability to handle different network conditions were also assessed. In problem (3.5), the CVP refers to the likelihood that a given solution violates the two constraints: transmit power $\mathbf{P}$ and SINR of the problem. The ASR refers to the cumulative data rate of multiple concurrent transmissions in the network, averaged over channel realizations.

The following chapter presents the results obtained from these experiments and interprets the findings.

Chapter 4

Discussion

This chapter outlines the experimental setup and the process of model validation and testing, presents the analysis results and discusses the findings from the study carried out to explore the application of DUL for achieving an optimum sum rate for D2D networks. The primary focus was understanding how effectively deep learning techniques can improve the sum rate, which is a critical factor in determining the overall network performance.

4.1 Setup, Training and Testing The DNN Model

4.1.1 Setup Specification

The simulations for this study were done on Google Colaboratory, also known as Colab, a research project created by Google. It provides a free virtual machine with pre-installed Python libraries, including TensorFlow, and offers access to GPUs and TPUs for machine learning tasks. The "Runtime" setting for Colab was set to:

- **Runtime type:** Python3
- **Hardware accelerator:** TPU
- **Shape:** High-RAM (Random-Access Memory)

4.1.2 Baseline Scheme

The performance of the proposed DUL approach is measured against PCNet, a method previously proposed in referenced literature [31]. This benchmark is used for a fair comparison. One key aspect that the proposed DNN-based method and PCNet share is their ability to circumvent intensive computational steps, such as projection procedures or iterative algorithms, which can often be resource-heavy. In addition, it employs a soft-loss function to train the NN, incorporating a penalty term, denoted by λ, into the sum rate.

A separate PCNet needs to be trained for a specific background noise power since it only considers channel coefficients as input. This requires multiple PCNet models for different noise powers in real-world applications, limiting its generalization. PCNet+, a modified version of PCNet, is introduced in [31] to improve generalization by including noise power, σ^2, alongside channel coefficients as input.The proposed DUL approach is also modified to compare the performance with the PCNet+ model.

4.1.3 Primary Parameters

Throughout the study, the symmetric interference channel model with i.i.d. the Rayleigh fading channel model with unity mean channel gain was considered for all channels. Five transmitter-receiver antenna pairs were taken to evaluate the efficacy of this proposed model with the PCNet model, i.e., when $K = 5$. Each receiver's noise output was set to the same variance or σ^2 level. The maximum transmit power was set to $\mathbf{P_{max}} = 1.0$ Watt for all transmitter and was considered the noise level $EsN0$ as:

$$EsN0 = 10 \log_{10} \frac{\mathbf{P_{max}}}{\sigma^2} \tag{4.1}$$

For performance analysis, the results under seven typical values of $EsN0$ = (0 dB, 10 dB, 20 dB, 30 dB, 40 dB, 50 dB, 60 dB) are presented. Different values of K, i.e., K = [5, 6, 7, 8], were also considered with $EsN0$ = (0 dB, 10 dB, 20 dB, 30 dB, 40 dB) to compare the performance with the PCNet+ model.

4.1.4 Datasets of Feasible Transmission Channel Parameters

In wireless communications, the SINR is an important parameter that quantifies cellular connection quality. It reflects the balance between the desired signal and the interference and noise levels.

- **Training with a given background noise power:** In this particular scenario, five different SINR cases were considered with $\mathbf{SINR_{min}} = 0.5$. Combining this with seven individual $EsN0$, 35 (1 K value $\times$ 5 SINR cases $\times$ 7 $EsN0$) distinct scenarios were taken. For each scenario, a dataset of 250,000 data points, each of which was a five-by-five matrix related to the transmission channel parameters, has been generated, resulting in an impressive total of 8,750,000 (35 $\times$ 250,000) data points. The total size of these datasets is 11.1 GB.
- **Training with enhanced generalization capacity:** For this scenario, four different K, i.e., $K = [5, 6, 7, 8]$ with the most stringent SINR case of $\mathbf{SINR_{min}} = 0.2$ values and five individual $EsN0$, 20 (4 K values $\times$ 1 SINR case $\times$ 5 $EsN0$) different scenarios were taken. For each scenario, a dataset of 250,000 data points resulted in a total of 5,000,000 (20 $\times$ 250,000). The total size of these datasets is 11 GB.

All the datasets can be accessed via the link provided in Appendix A. These comprehensive datasets provide a wide variety of scenarios, that cater to numerous situations in wireless communications, enabling robust analysis and modeling.

Table 4.1, referenced in the text, compares the ratio between feasible and random datasets for the transmission channel parameters for five different SINR cases with seven individual noise levels ($EsN0$) in dB, when $K = 5$. A feasible dataset represents a likely scenario that meets both the constraints $SINR_i(\mathbf{P})$ and P_i of Equation (3.5). In contrast, a random dataset is generated without considering specific conditions or restrictions. The ratio between these two dataset types helps to understand the balance between real-world applicability (feasible datasets) and broad variability (random datasets), which can significantly impact the model's performance and ability to generalize to new, unseen data.

Table 4.1: Count ratios of feasible vs. random datasets for the channel parameters with 5 SINR cases for $K = 5$, e.g., Case 3 : $\mathbf{SINR_{min}}$= [0.5, 0.5, 0.5, 0.0, 0.0]

$\mathbf{SINR_{min}}$	**0 dB**	**10 dB**	**20 dB**	**30 dB**	**40 dB**	**50 dB**	**60 dB**
Case 1	60.65%	95.13%	99.50%	99.95%	99.99%	100.00%	100.00%
Case 2	22.51%	63.05%	70.79%	71.69%	71.70%	71.70%	71.80%
Case 3	4.73%	25.76%	31.50%	32.06%	32.19%	32.21%	32.27%
Case 4	0.55%	5.91%	7.92%	8.14%	8.15%	8.16%	8.22%
Case 5	0.03%	0.74%	1.05%	1.10%	1.11%	1.13%	1.13%

In analyzing the performances, the dataset was divided into three distinct sections: training, testing, and validation. This division is essential for ensuring the model have a balanced ability to learn from data (training), evaluate its performance during the learning process (validation), and verify its final accuracy and robustness on unseen data (testing). An 80%:10%:10% split for the training, validation, and testing data was utilized.

4.1.5 Tuning Hyperparameters

Hyperparameter tuning in DNNs is a central facet of model optimization and instrumental in realizing peak performance. Consequently, an extensive preliminary analysis was conducted to tune key hyperparameters systematically. These include the learning rate, batch size, number of hidden layers and neurons, epochs, and optimization algorithms. The Adam Optimizer was employed for network training in alignment with conventional practice. However, an array of other hyperparameters were explored and tested. This selection was driven by the necessity to strike a balance among critical factors: computational execution time, effective training of the model, and the achievement of an optimal average sum rate.

This report presents several training outcomes predicated on $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.5) and EsN0 set at 0 dB. Figures 4.1 through 4.4 demonstrate the implications of varying the learning rate from 0.1 to 0.0001. The training trajectories

corresponding to mini-batch sizes of 10,000 and 100 are depicted in Figures 4.5 and 4.6, respectively. As represented in Figure 4.7, an increased number of epochs results in a flattened tail in the learning curve. Moreover, Figures 4.8 and 4.9 display the training curves resulting from the addition of one [(K × K), (2 × K × K), (K × K)] and two [(K × K), (K × K), (K × K), (K × K)] more dense layers, respectively from [(2 × K × K), (K × K)] set of neurons. Finally, Table 4.2 provides a detailed account of the execution time and the average sum rate (measured in Bit/Second/Hertz) for all the hyperparameter combinations explored in this study. It should be noted that there is a 0% CVP, meaning the HR is consistently 100% across all scenarios.

The trials demonstrated the significant influence of various hyperparameters on the performance and efficiency of the training process in DNNs. If the learning rate is too large, the model may overshoot the optimal solution; if it is too small, the learning process can become excessively slow. Batch size, the number of training examples used in one iteration, impacts both the training process's speed and the model's quality. Smaller batch sizes can offer a regularizing effect and better generalization, but the learning process may become slower and less stable. Larger batch sizes can make the training faster, but they require more memory, and the model may converge to a less optimal solution.

The epoch size determines the number of complete passes the learning algorithm makes over the entire training dataset. This parameter's optimal value is crucial for the convergence and performance of the model. Too few epochs can lead to underfitting of the model, as it might not have learned all the intricate patterns in the data. Conversely, too many epochs can lead to overfitting, as the model may start to memorize the training data instead of learning to generalize from it. Furthermore, the number of epochs directly affects the computational cost, as more epochs require more time to compute. Therefore, choosing an appropriate epoch size is a delicate balance between model performance and computational efficiency.

Determining the optimal number of hidden dense layers and neurons per layer is challenging. More layers and neurons can increase the model's capacity to learn complex patterns, but they can also make the model prone to overfitting and increase the computational cost. Conversely, too few can lead to underfitting.

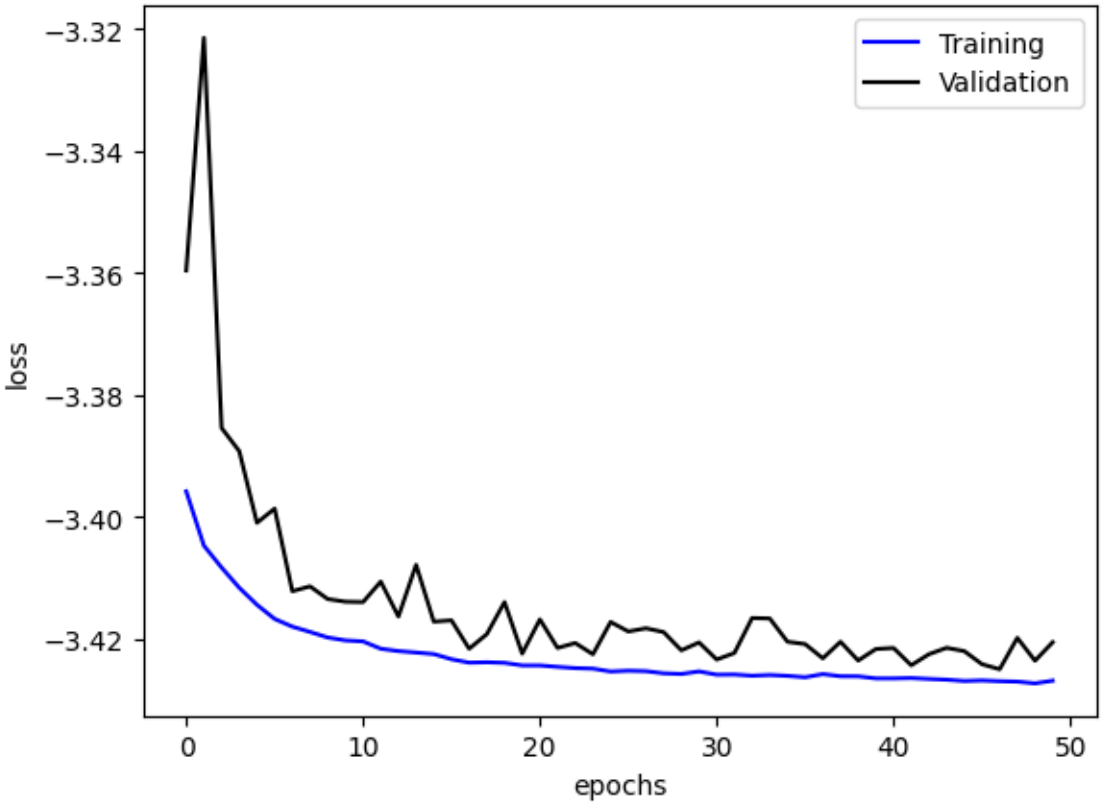

Figure 4.1: Training with Learning Rate = 0.1

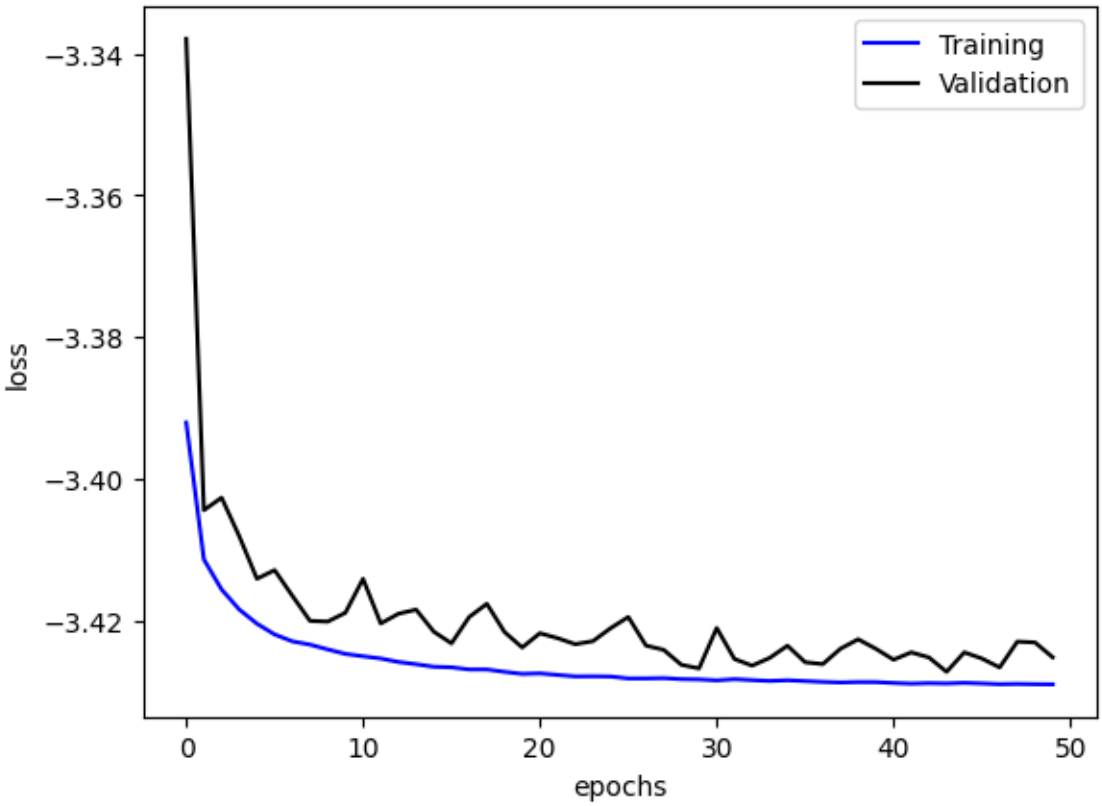

Figure 4.2: Training with Learning Rate = 0.01

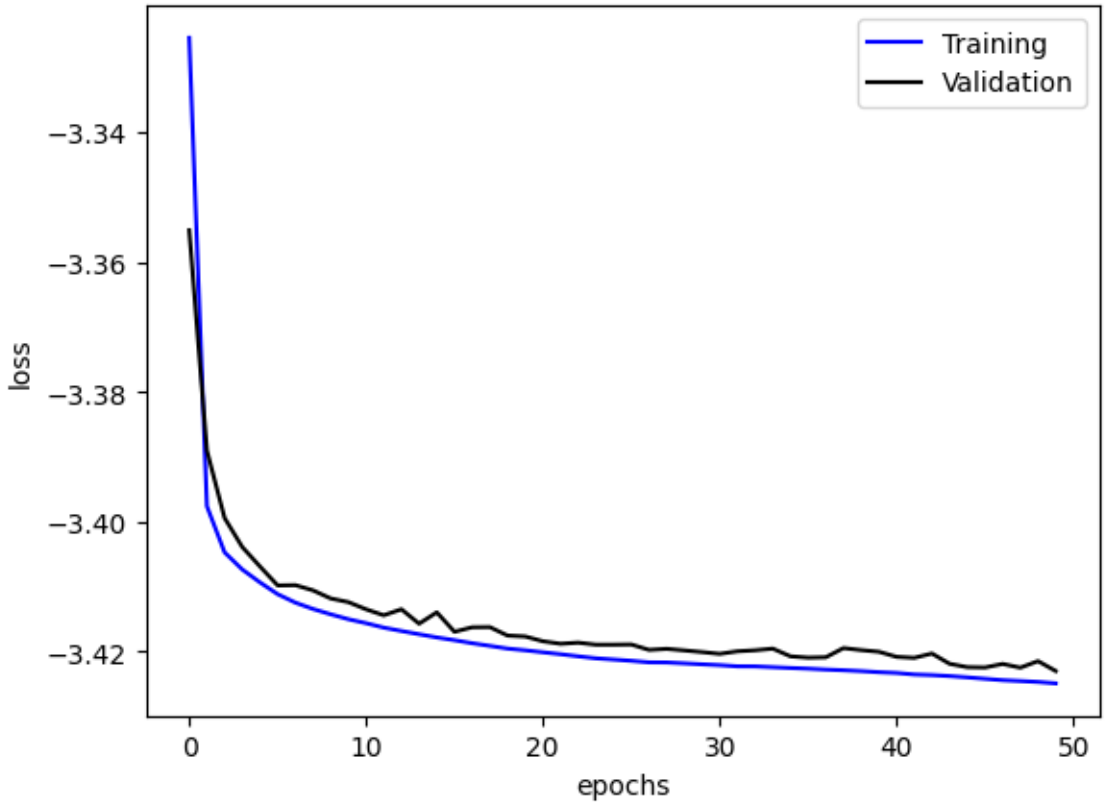

Figure 4.3: Training with Learning Rate = 0.001

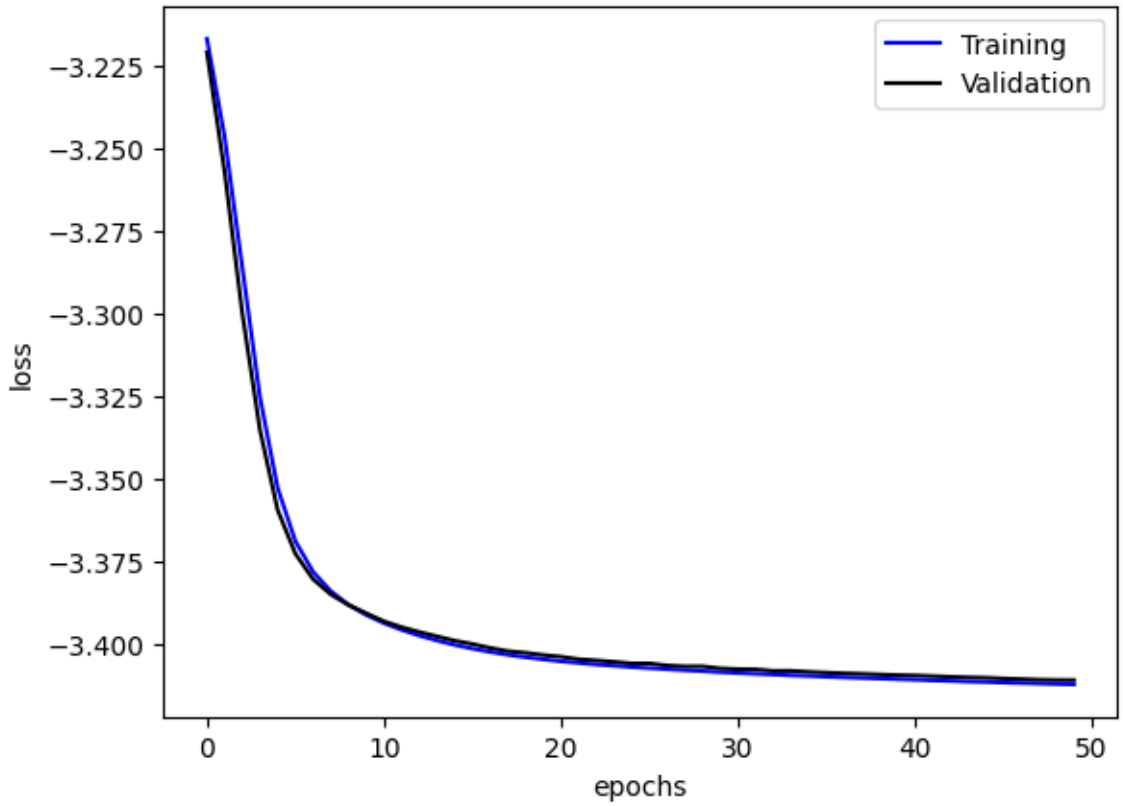

Figure 4.4: Training with Learning Rate = 0.0001

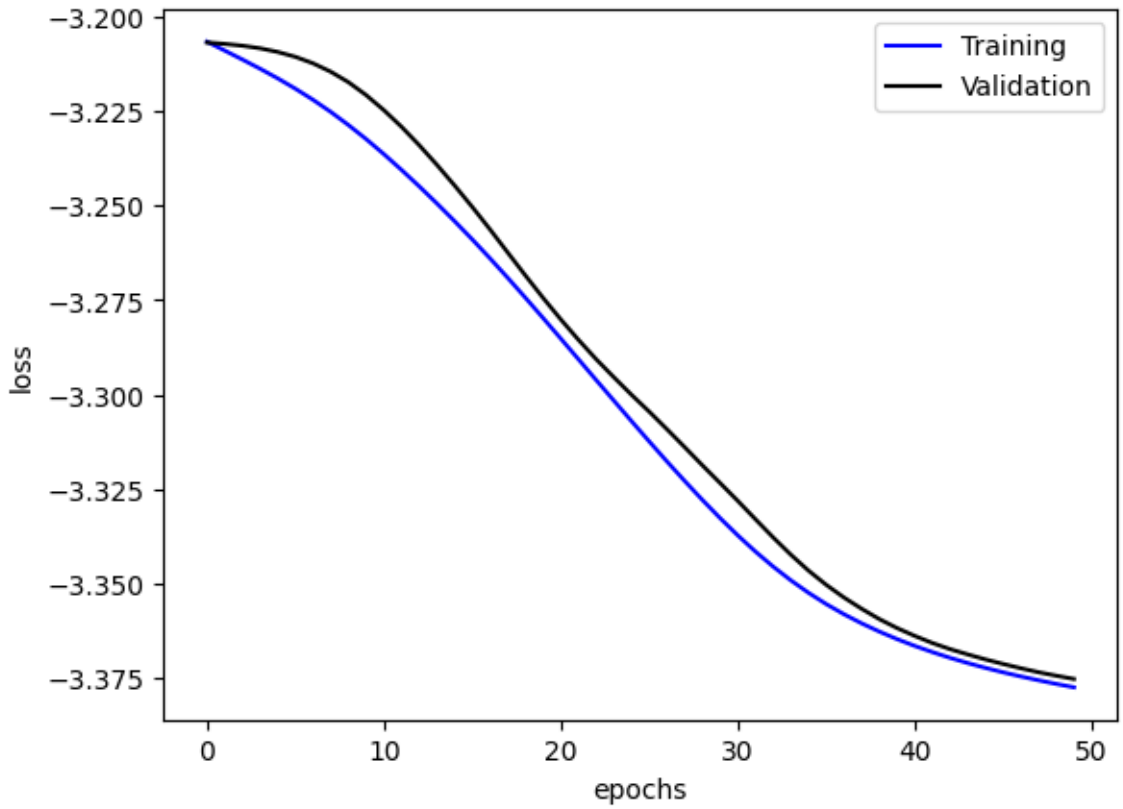

Figure 4.5: Training with Mini-Batch Size = 10,000

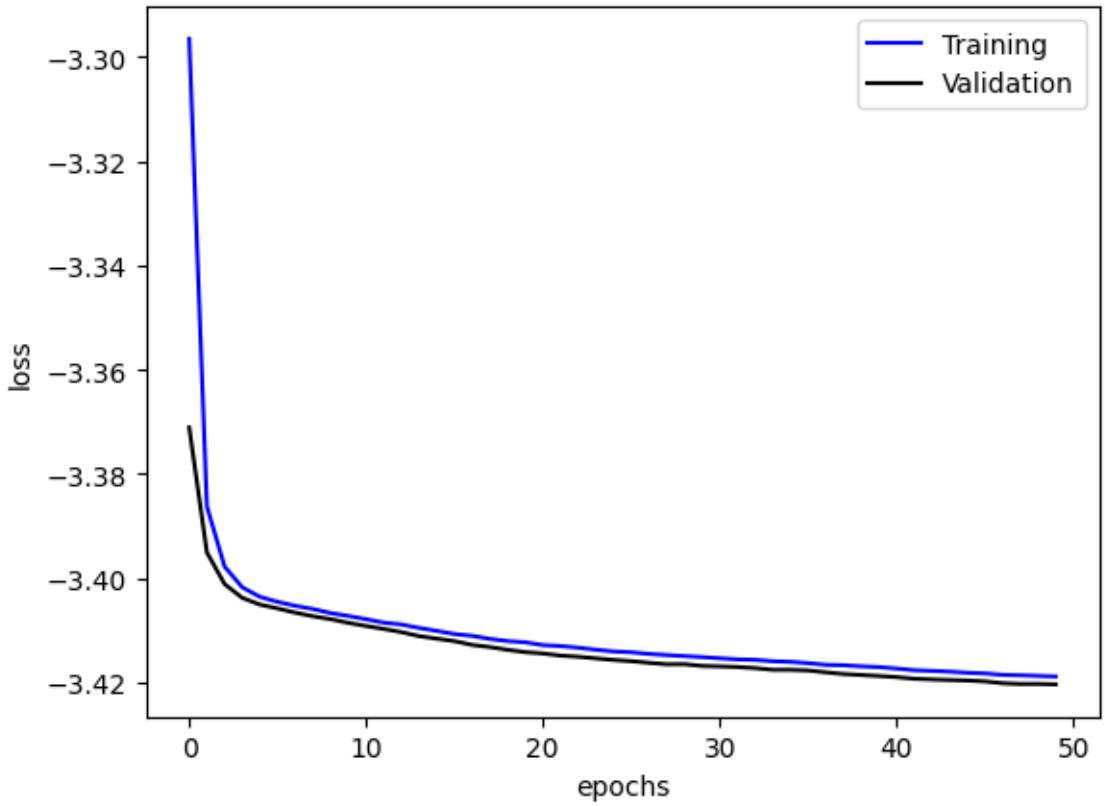

Figure 4.6: Training with Mini-Batch Size = 100

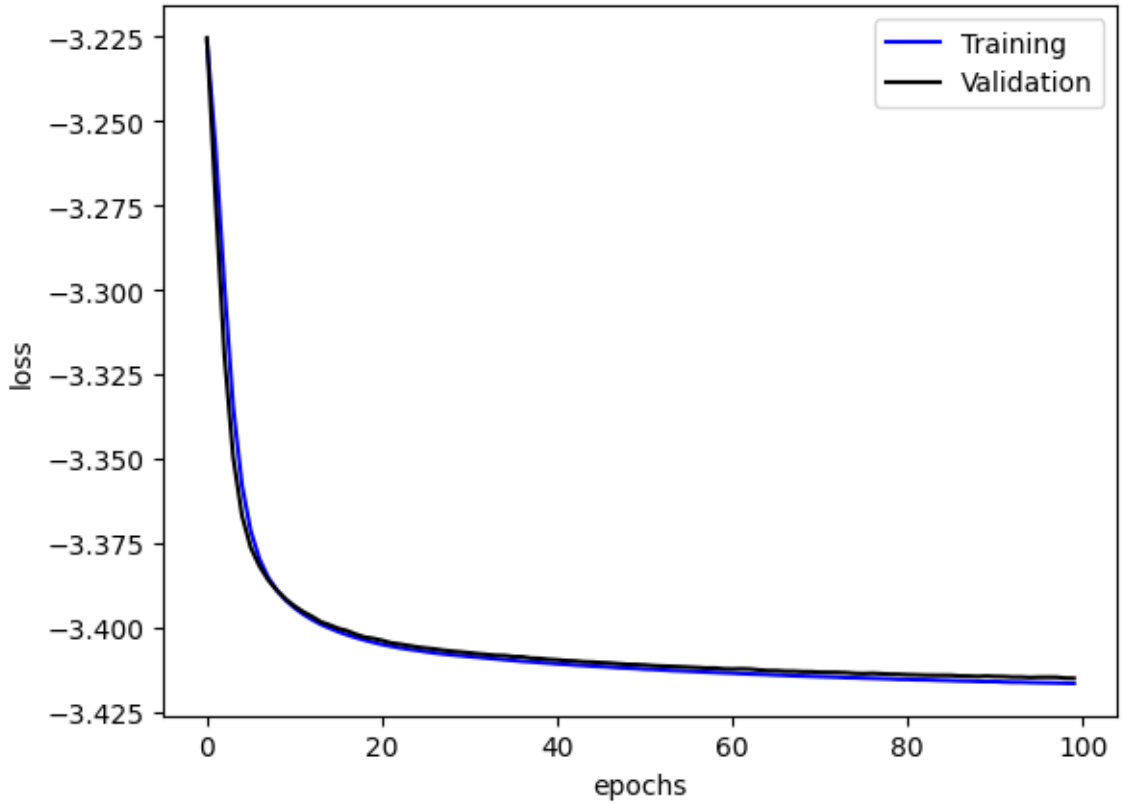

Figure 4.7: Training with epoch = 100

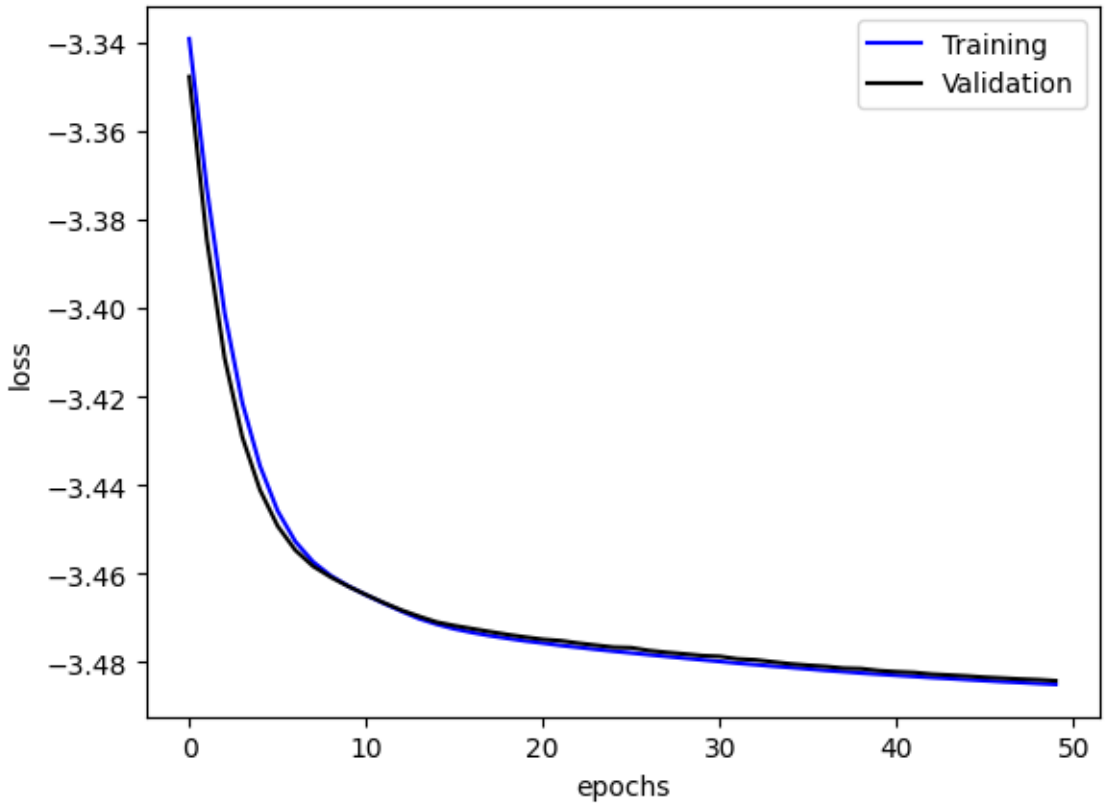

Figure 4.8: Training with three dense layers and [25, 50, 25] set of neurons

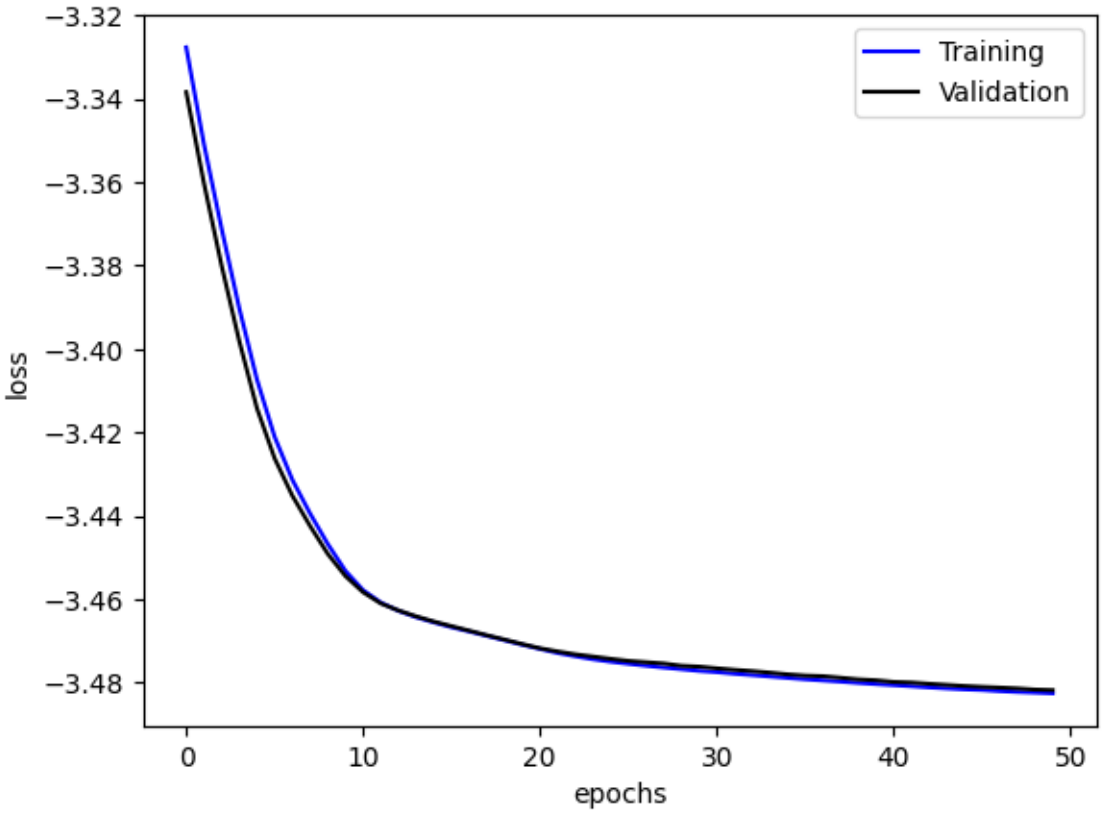

Figure 4.9: Training with four dense layers and [25, 25, 25, 25] set of neurons

Table 4.2: Results for different hyperparameters for EsN0 = 0 dB and $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.5)

Layer	Epoch	Batch Size	Learning Rate	Training Time	Avgerage Sum Rate (Bit /Second /Hertz)
2	50	1,000	0.1	55.952s	3.492
2	50	1,000	0.01	55.889s	3.498
2	50	1,000	0.001	55.859s	3.490
2	50	1,000	0.0001	56.463s	3.483
2	50	10,000	0.0001	19.64s	3.455
2	50	100	0.0001	294.999s	3.490
2	100	1,000	0.0001	109.658s	3.486
3	50	1,000	0.0001	66.399s	3.487
4	50	1,000	0.0001	71.527s	3.486

4.2 Results of the Analysis

Certain parameters were held constant to ensure a balanced and equitable comparison between the proposed DUL method and PCNet/PCNet+. Specifically, the architecture of the neural networks, in terms of the number of hidden layers and neurons per layer, was maintained identically for both approaches. The performance was measured with two dense layers of [(2 × K × K), (K × K)] number of neurons, respectively. The NNs were trained using a mini-batch gradient descent algorithm with a batch size of 1,000 realizations. Adam Optimizer was used with a learning rate of 0.0001 and a total of 50 epochs.

In addition, for a fair comparison of the ASR, the infeasible output power vectors were scaled for the benchmark schemes, i.e., PCNet/PCNet+. As detailed in subsection 4.1.5, the proposed model consistently yields a feasible power vector. Conversely, the NNs used in the benchmark schemes do not always produce power vectors adhering to the constraints. So, a heuristic method is used, where $\widehat{\mathbf{P}}$ is scaled when the NN gives a power vector that is not feasible, as shown in Equation (15) of [31].

The DNN input layer contains K^2 nodes for training with a given background noise power because it only takes channel coefficients as the input. However, the input layer contains $(K^2 + 1)$ inputs for training its NNs with enhanced generalization capacity to take the noise power σ^2 as another input to the network, in addition to the channel coefficients.

4.2.1 Training with A Given Background Noise Power

Comparison plots on ASR (4.10, 4.12, 4.14, 4.16 and 4.18) and HR (4.11, 4.13, 4.15, 4.17 and 4.19) between the proposed model and PCNet are presented in the next few pages for the five cases of minimum SINR requirements. Respective data tables are also provided for reference. Tables 4.3, 4.5, 4.7, 4.9, and 4.11 are for ASR for both models. However, Tables 4.4, 4.6, 4.8, 4.10 and 4.12 are only for HR of PCNet for different λ values. HR for proposed model is always 100%.

4.2.1.1 Results for Case 1 : $\mathbf{SINR_{min}}$= [0.5, 0.0, 0.0, 0.0, 0.0]

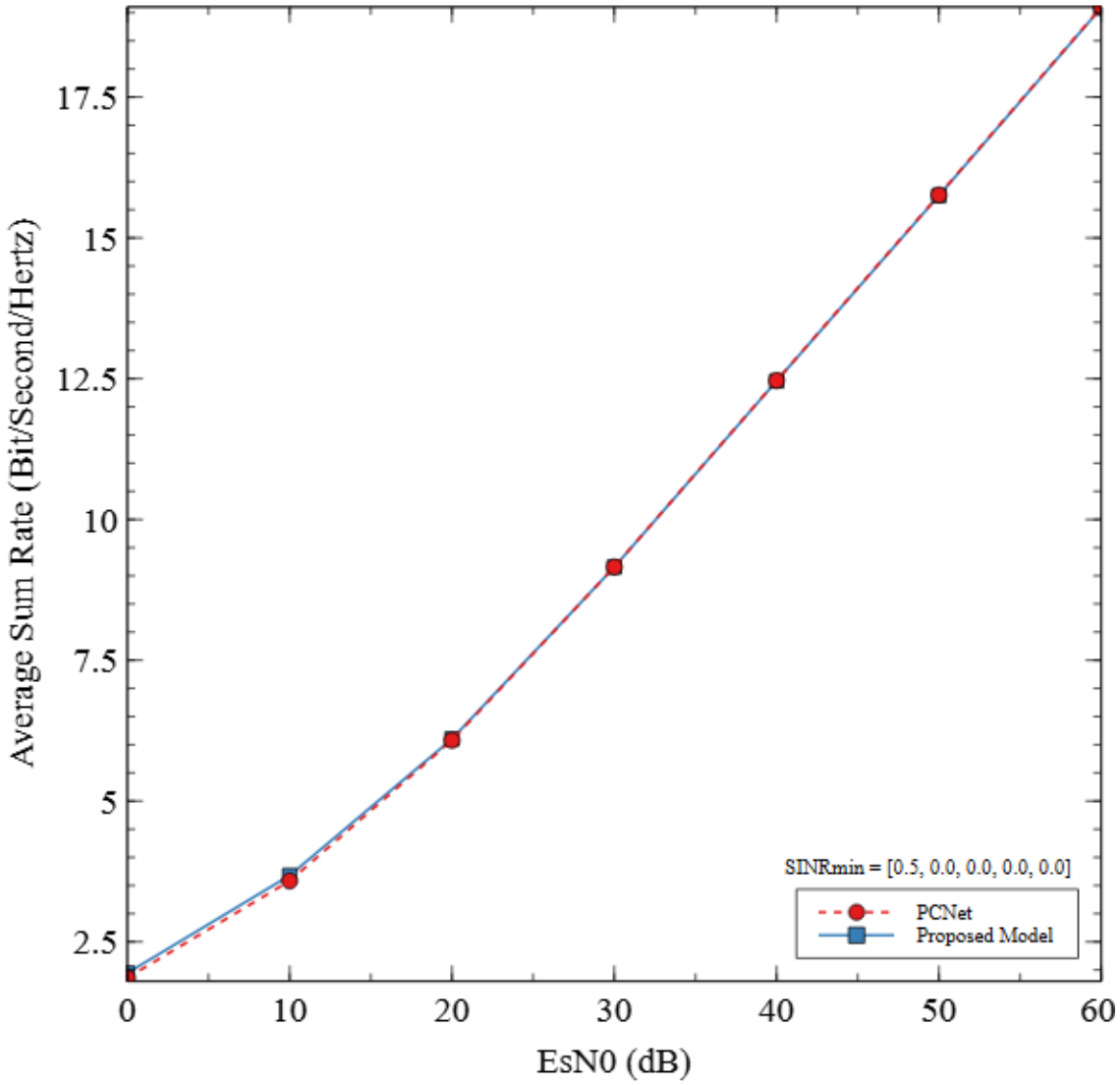

Figure 4.10: Average Sum Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.0, 0.0, 0.0, 0.0]

Table 4.3: Average Sum Rates (Bit/Second/Hertz) for different EsN0 (dB) for $\mathbf{SINR_{min}}$ = (0.5, 0.0, 0.0, 0.0, 0.0)

Model	0 dB	10 dB	20 dB	30 dB	40 dB	50 dB	60 dB
PCNet	1.860	3.580	6.078	9.156	12.466	15.761	19.096
Proposed	1.945	3.678	6.102	9.154	12.459	15.751	19.085

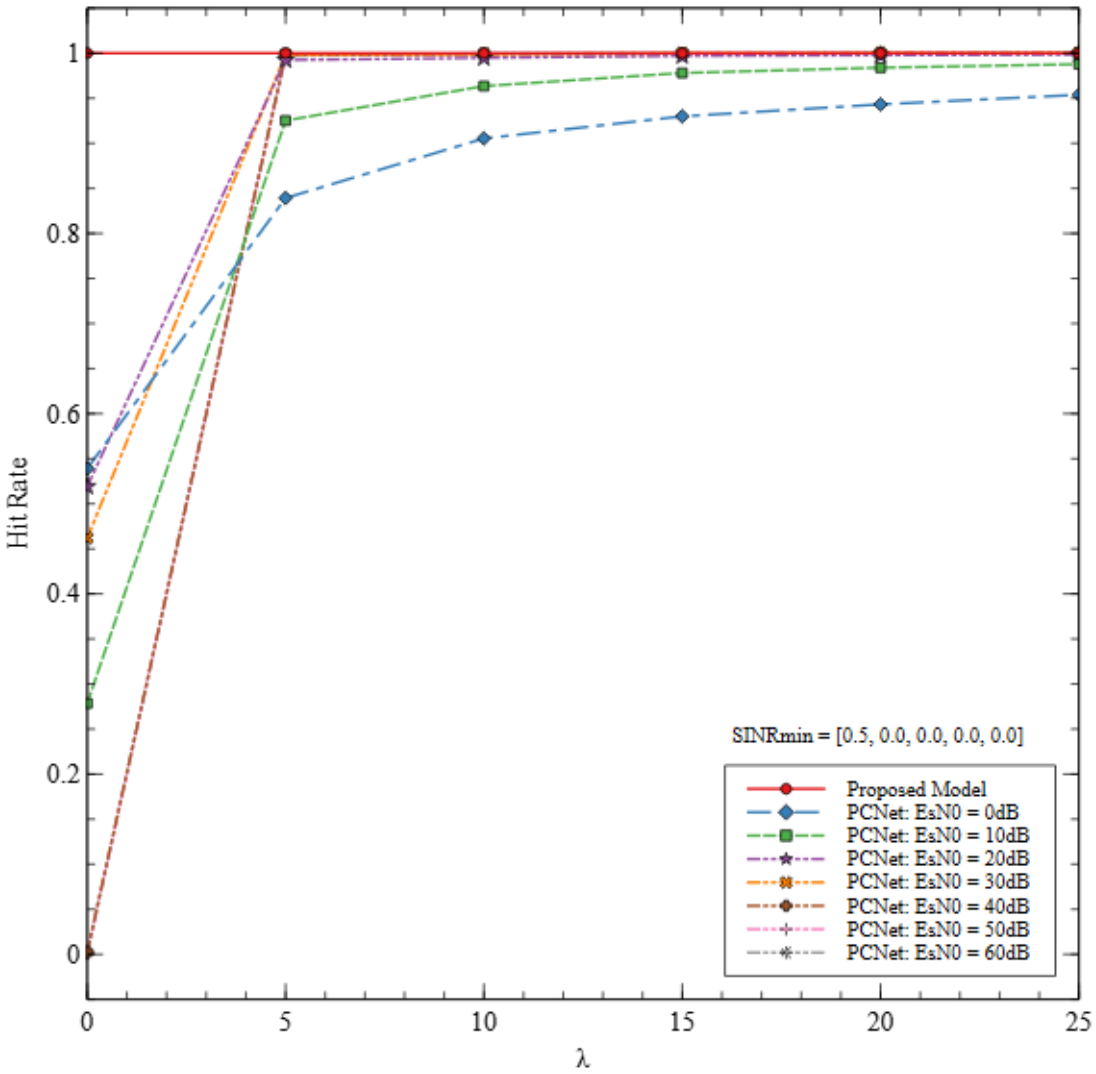

Figure 4.11: Hit Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.0, 0.0, 0.0, 0.0]

Table 4.4: Hit Rates for PCNet for $\mathbf{SINR_{min}}$ = (0.5, 0.0, 0.0, 0.0, 0.0)

EsN0	$\lambda = 0$	$\lambda = 5$	$\lambda = 10$	$\lambda = 15$	$\lambda = 20$	$\lambda = 25$
0 dB	53.92%	83.92%	90.54%	92.99%	94.32%	95.38%
10 dB	27.84%	92.52%	96.35%	97.80%	98.39%	98.78%
20 dB	51.91%	99.24%	99.48%	99.70%	99.77%	99.84%
30 dB	46.20%	99.84%	99.90%	100.00%	100.00%	100.00%
40 dB	0.25%	99.84%	99.87%	99.90%	99.99%	100.00%
50 dB	0.11%	99.82%	99.94%	99.98%	100.00%	100.00%
60 dB	0.19%	99.69%	99.97%	100.00%	100.00%	100.00%

4.2.1.2 Results for Case 2 : $\mathbf{SINR_{min}}$= [0.5, 0.5, 0.0, 0.0, 0.0]

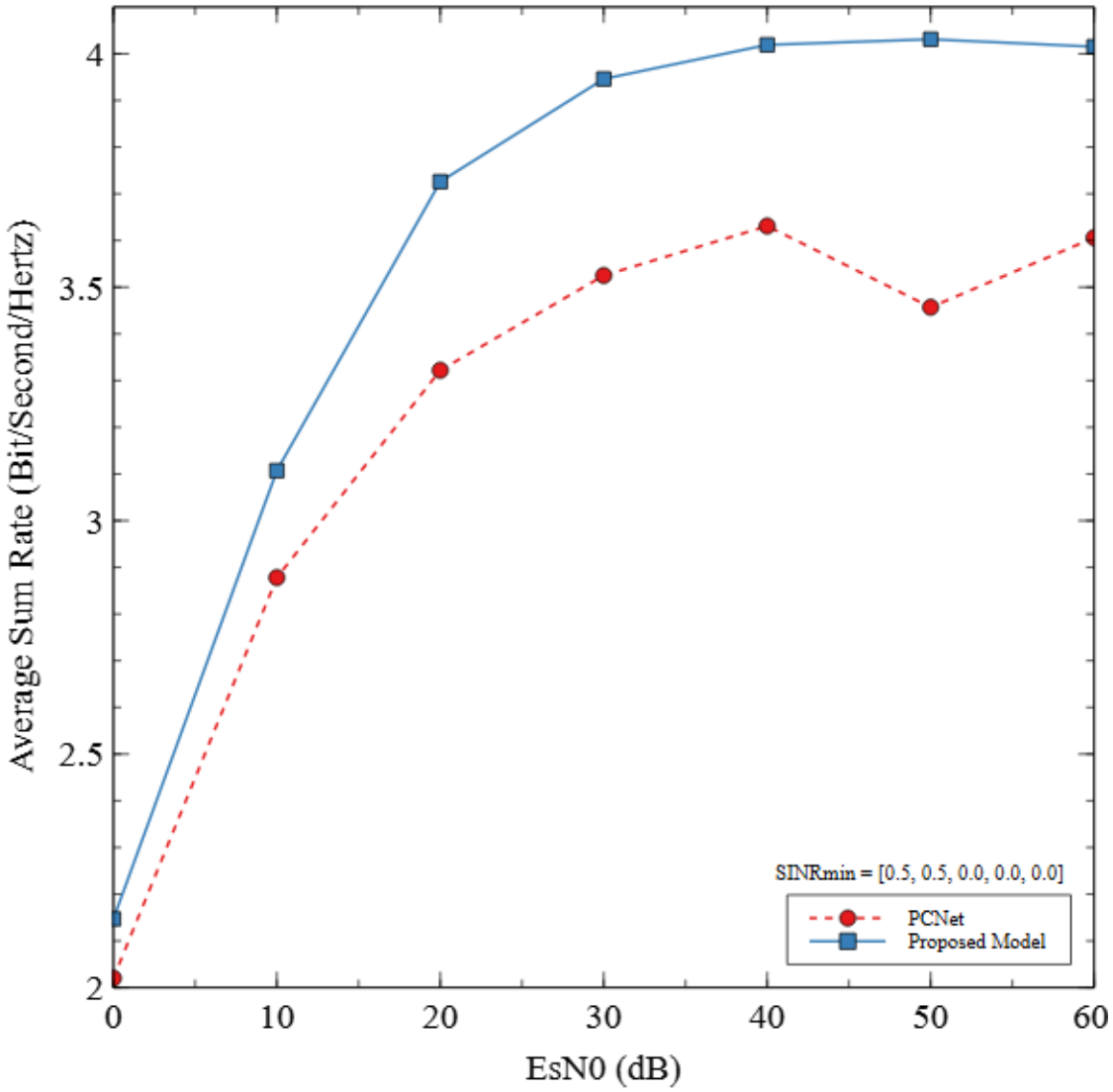

Figure 4.12: Average Sum Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.0, 0.0, 0.0]

Table 4.5: Average Sum Rates (Bit/Second/Hertz) for different EsN0 (dB) for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.0, 0.0, 0.0)

Model	0 dB	10 dB	20 dB	30 dB	40 dB	50 dB	60 dB
PCNet	2.020	2.878	3.322	3.525	3.631	3.457	3.606
Proposed	2.147	3.107	3.726	3.946	4.019	4.031	4.015

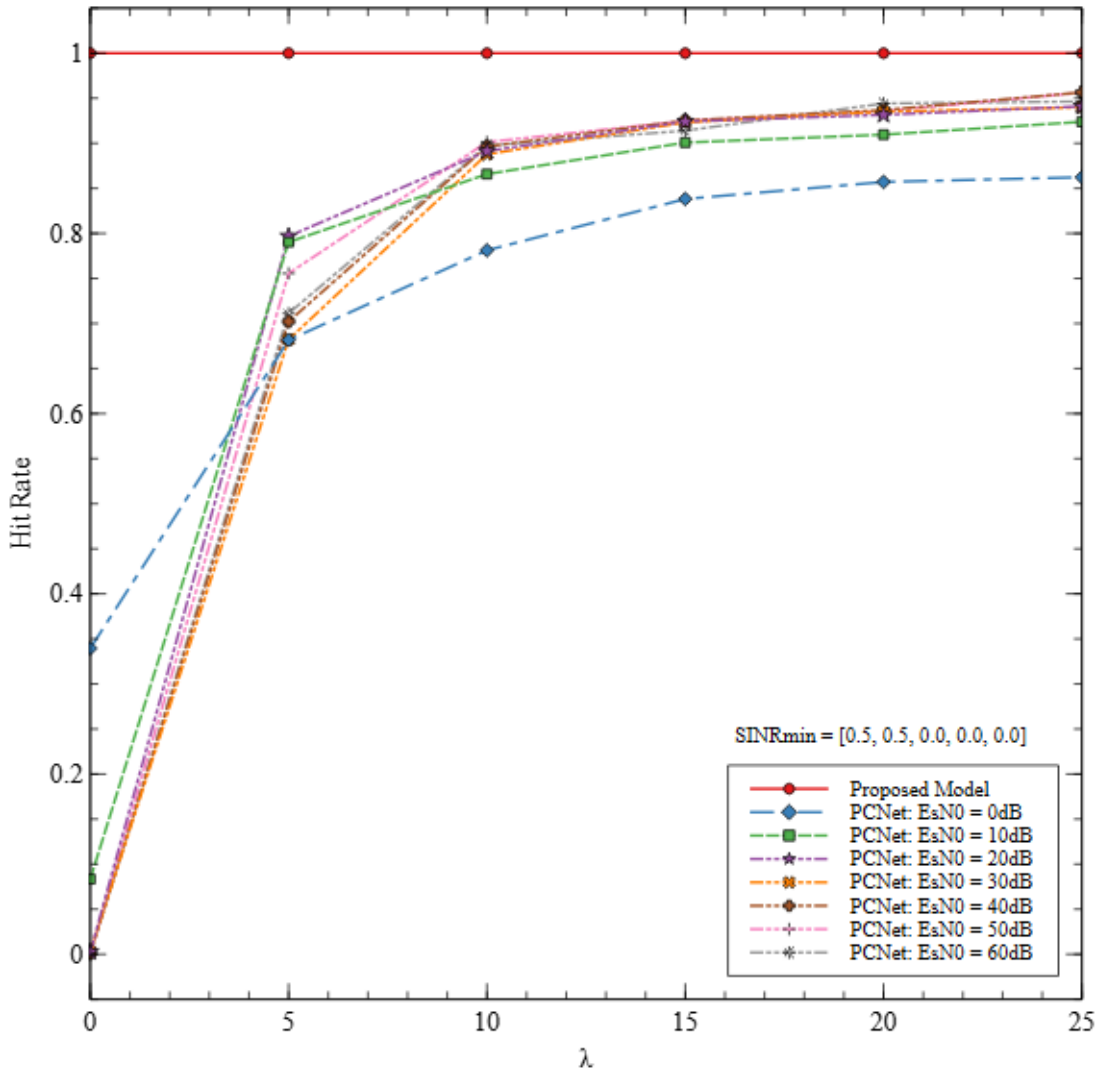

Figure 4.13: Hit Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.0, 0.0, 0.0]

Table 4.6: Hit Rates for PCNet for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.0, 0.0, 0.0)

EsN0	$\lambda = 0$	$\lambda = 5$	$\lambda = 10$	$\lambda = 15$	$\lambda = 20$	$\lambda = 25$
0 dB	33.95%	68.16%	78.12%	83.82%	85.71%	86.23%
10 dB	8.34%	79.03%	86.58%	90.08%	90.96%	92.38%
20 dB	0.26%	79.68%	89.13%	92.41%	93.16%	94.10%
30 dB	0.32%	68.22%	88.78%	92.32%	93.50%	93.99%
40 dB	0.11%	70.22%	89.64%	92.53%	93.69%	95.65%
50 dB	0.00%	75.56%	90.13%	92.48%	93.56%	95.62%
60 dB	0.16%	71.12%	89.75%	91.40%	94.42%	94.64%

4.2.1.3 Results for Case 3 : $\mathbf{SINR_{min}}$= [0.5, 0.5, 0.5, 0.0, 0.0]

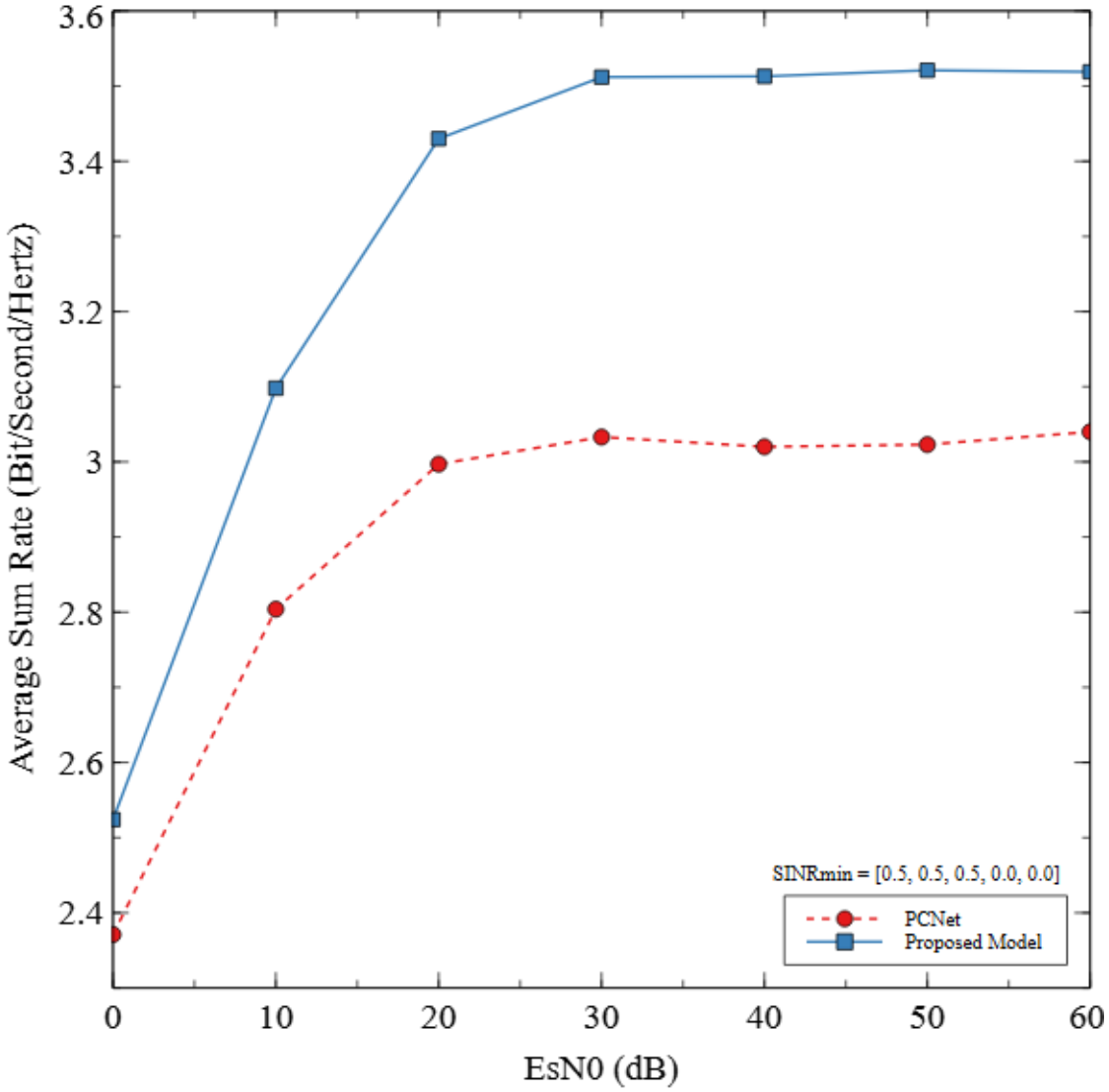

Figure 4.14: Average Sum Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.5, 0.0, 0.0]

Table 4.7: Average Sum Rates (Bit/Second/Hertz) for different EsN0 (dB) for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.0, 0.0)

Model	0 dB	10 dB	20 dB	30 dB	40 dB	50 dB	60 dB
PCNet	2.371	2.804	2.997	3.033	3.020	3.023	3.040
Proposed	2.524	3.098	3.430	3.512	3.513	3.521	3.519

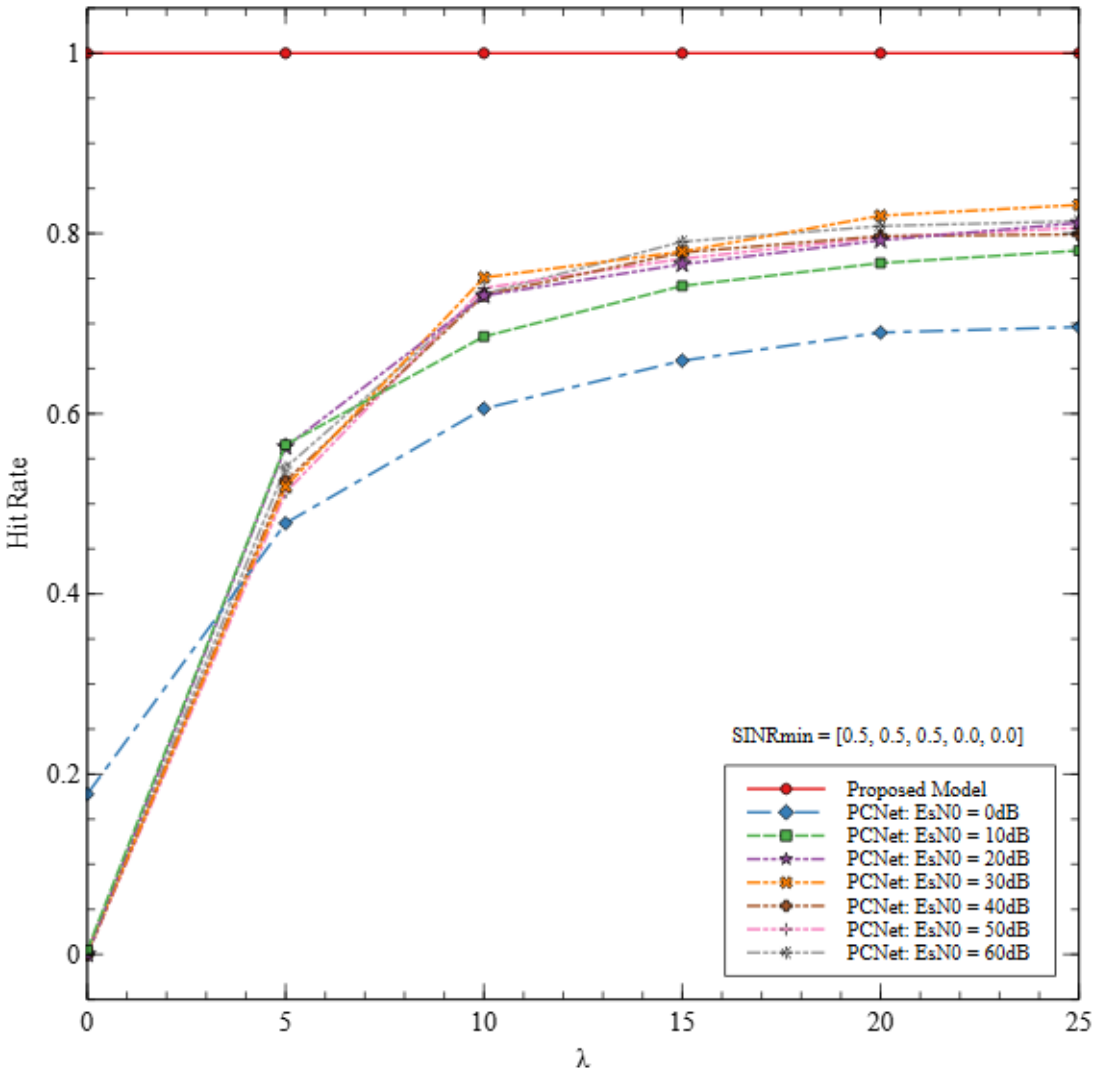

Figure 4.15: Hit Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.5, 0.0, 0.0]

Table 4.8: Hit Rates for PCNet for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.0, 0.0)

EsN0	$\lambda = 0$	$\lambda = 5$	$\lambda = 10$	$\lambda = 15$	$\lambda = 20$	$\lambda = 25$
0 dB	17.78%	47.84%	60.56%	65.88%	69.00%	69.62%
10 dB	0.47%	56.54%	68.56%	74.18%	76.71%	78.09%
20 dB	0.01%	56.30%	73.10%	76.58%	79.23%	81.17%
30 dB	0.24%	51.90%	75.10%	77.99%	81.97%	83.17%
40 dB	0.00%	52.51%	73.15%	77.89%	79.70%	79.89%
50 dB	0.00%	51.26%	73.94%	77.19%	79.57%	80.62%
60 dB	0.00%	54.01%	73.28%	79.08%	80.82%	81.36%

4.2.1.4 Results for Case 4 : $\mathbf{SINR_{min}}$= [0.5, 0.5, 0.5, 0.5, 0.0]

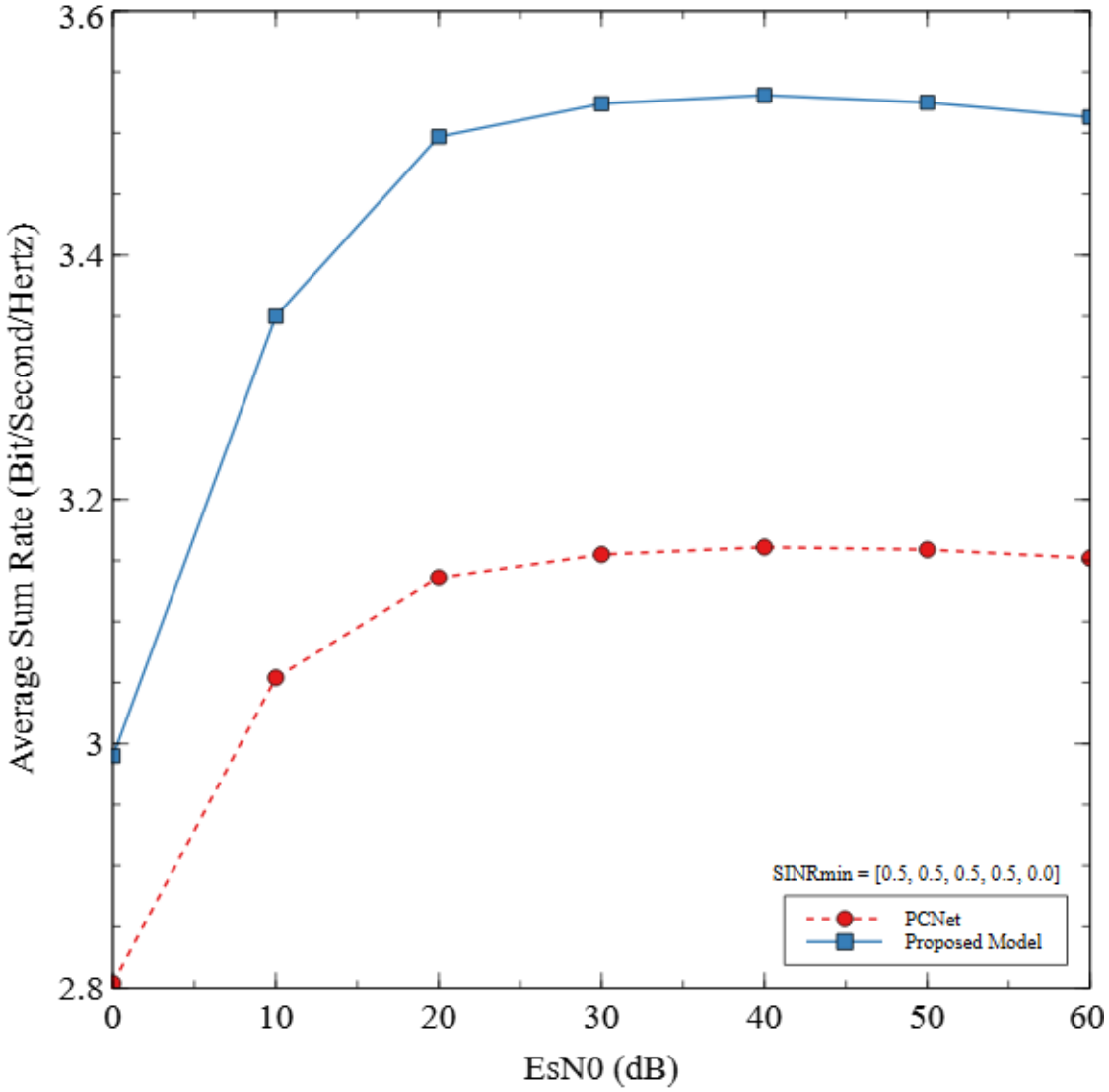

Figure 4.16: Average Sum Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.5, 0.5, 0.0]

Table 4.9: Average Sum Rates (Bit/Second/Hertz) for different EsN0 (dB) for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.0)

Model	0 dB	10 dB	20 dB	30 dB	40 dB	50 dB	60 dB
PCNet	2.804	3.054	3.136	3.155	3.161	3.159	3.152
Proposed	2.990	3.350	3.497	3.524	3.531	3.525	3.513

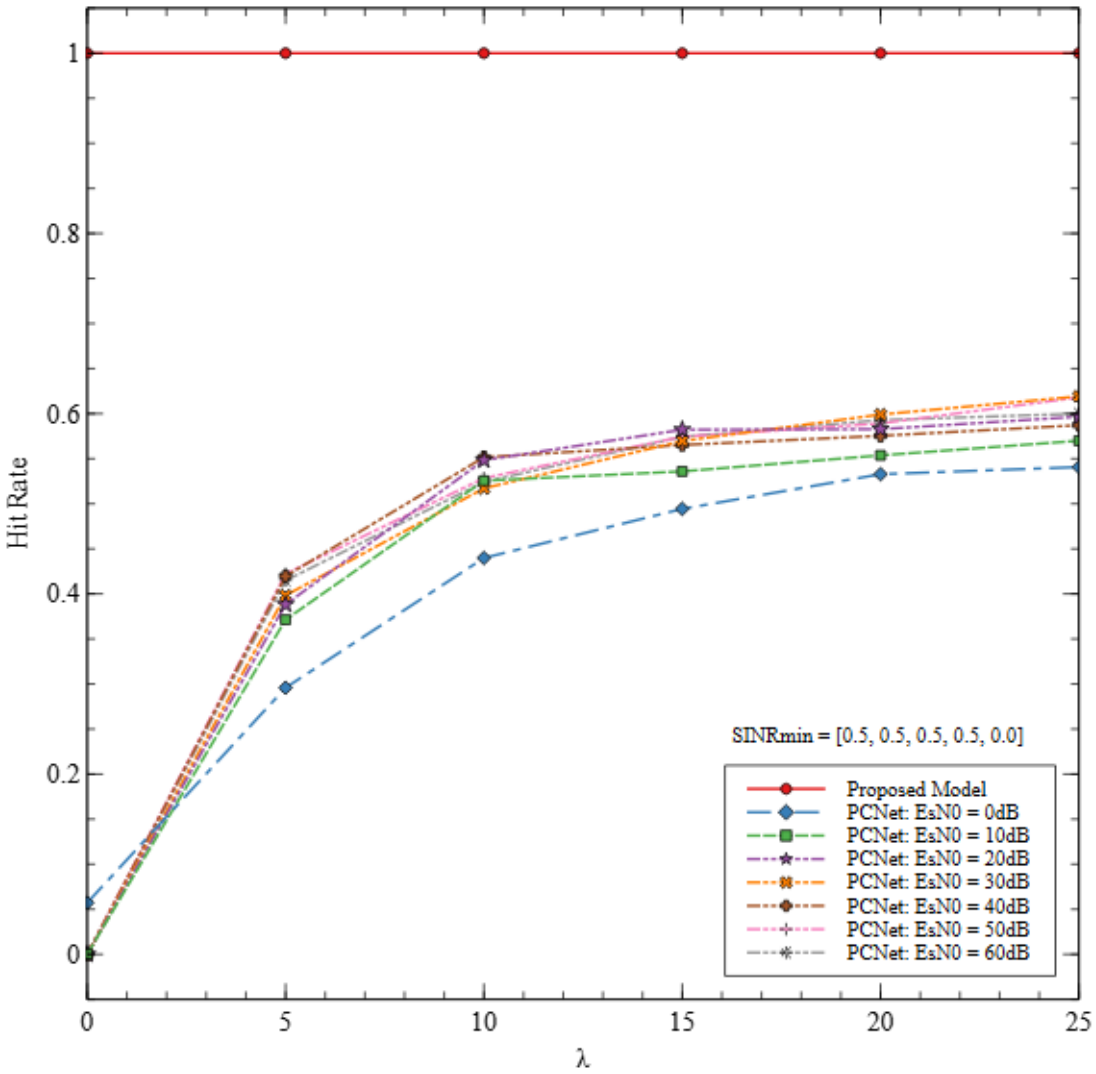

Figure 4.17: Hit Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.5, 0.5, 0.0]

Table 4.10: Hit Rates for PCNet for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.0)

EsN0	$\lambda = 0$	$\lambda = 5$	$\lambda = 10$	$\lambda = 15$	$\lambda = 20$	$\lambda = 25$
0 dB	5.72%	29.58%	43.98%	49.43%	53.28%	54.07%
10 dB	0.02%	37.13%	52.56%	53.59%	55.37%	56.99%
20 dB	0.00%	38.82%	54.83%	58.22%	58.28%	59.66%
30 dB	0.00%	39.82%	51.71%	56.96%	59.90%	61.92%
40 dB	0.00%	41.96%	55.16%	56.52%	57.74%	58.72%
50 dB	0.00%	42.23%	52.84%	57.44%	58.91%	61.80%
60 dB	0.00%	41.48%	52.41%	57.44%	59.28%	60.00%

4.2.1.5 Results for Case 5 : $\mathbf{SINR_{min}}$= [0.5, 0.5, 0.5, 0.5, 0.5]

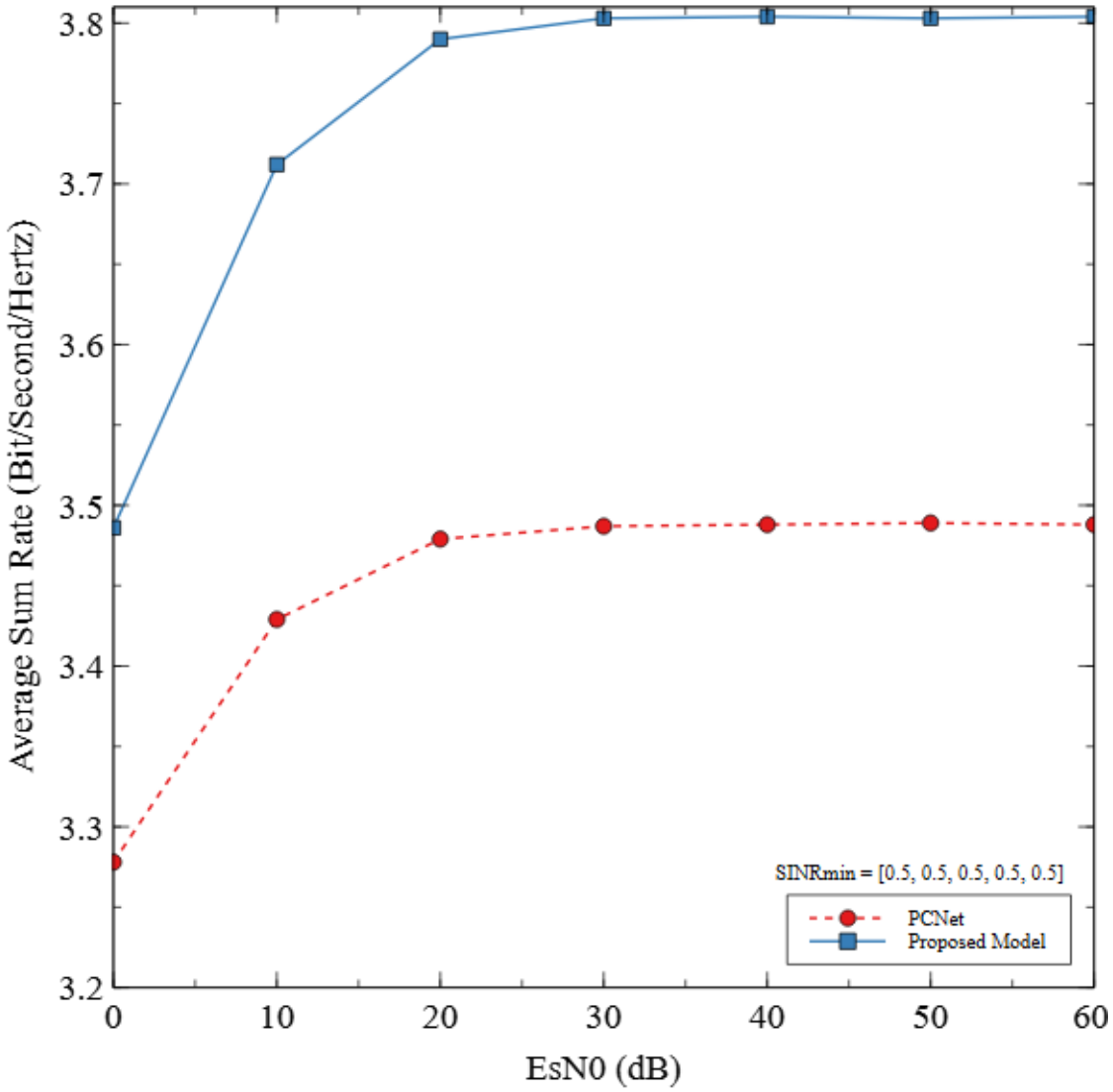

Figure 4.18: Average Sum Rate Plot for $\mathbf{SINR_{min}}$ = [0.5, 0.5, 0.5, 0.5, 0.5]

Table 4.11: Average Sum Rates (Bit/Second/Hertz) for different EsN0 (dB) for $\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.5)

Model	0 dB	10 dB	20 dB	30 dB	40 dB	50 dB	60 dB
PCNet	3.278	3.429	3.479	3.487	3.488	3.489	3.488
Proposed	3.486	3.712	3.790	3.803	3.804	3.803	3.804

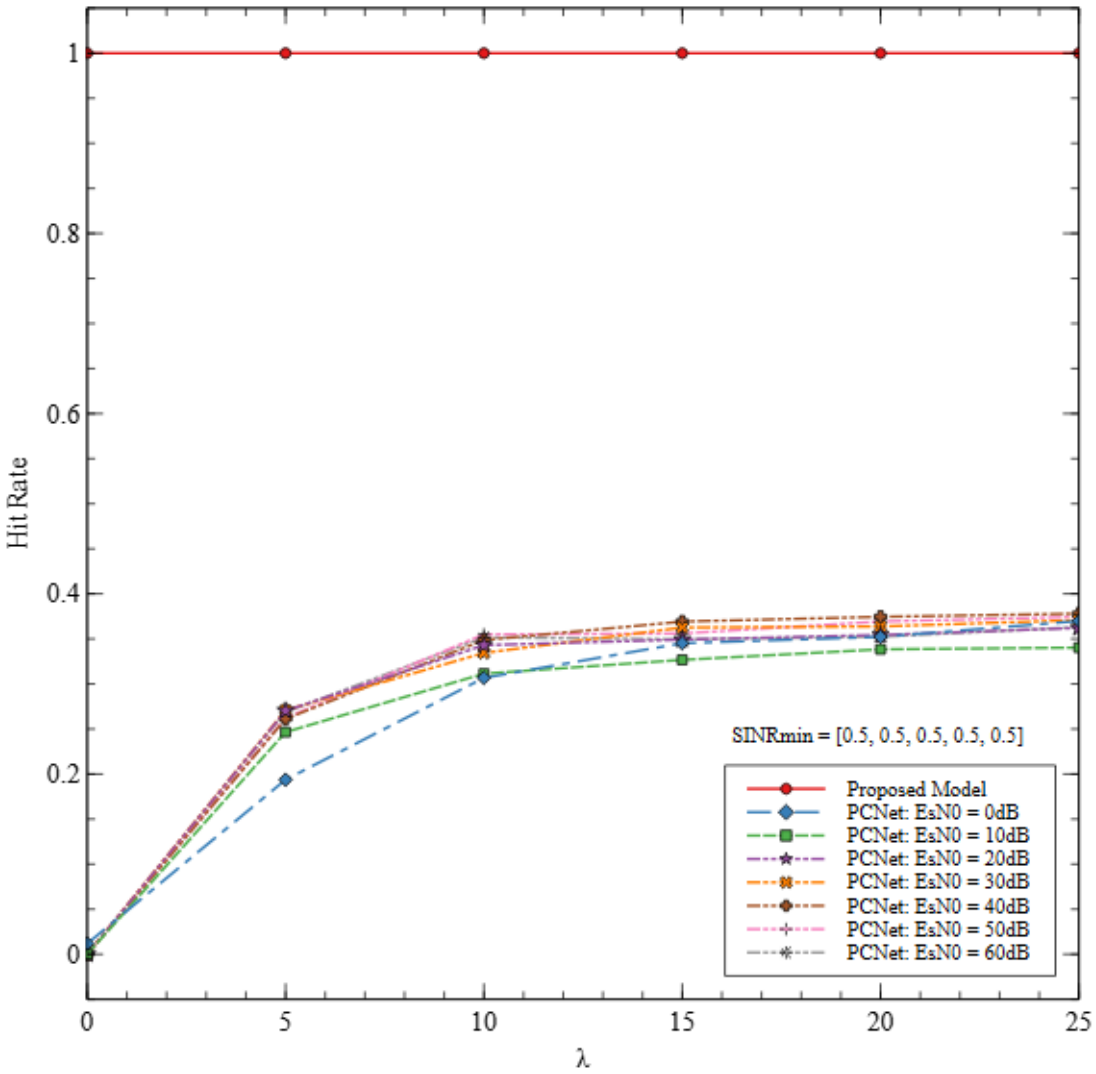

Figure 4.19: Hit Rate Plot for $\mathbf{SINR_{min}} = [0.5, 0.5, 0.5, 0.5, 0.5]$

Table 4.12: Hit Rates for PCNet for $\mathbf{SINR_{min}} = (0.5, 0.5, 0.5, 0.5, 0.5)$

EsN0	$\lambda = 0$	$\lambda = 5$	$\lambda = 10$	$\lambda = 15$	$\lambda = 20$	$\lambda = 25$
0 dB	1.18%	19.37%	30.67%	34.52%	35.22%	36.99%
10 dB	0.00%	24.65%	31.14%	32.67%	33.84%	34.03%
20 dB	0.00%	27.02%	34.30%	34.92%	35.43%	36.22%
30 dB	0.00%	27.08%	33.44%	36.26%	36.38%	37.09%
40 dB	0.00%	26.11%	34.87%	36.92%	37.45%	37.80%
50 dB	0.00%	26.22%	35.49%	35.60%	36.93%	37.44%
60 dB	0.00%	26.88%	35.17%	34.95%	35.24%	36.31%

Figures 4.10 to 4.19 show the proposed DUL model consistently outperforming the PCNet model across all QoS (i.e., $SINR_{min}$ requirements) and the entire $EsN0$ range for HR and ASR, except in Case 1 of $SINR_{min}$, where they perform at par for ASR. The proposed model adhered to all constraints, recording no violations. In contrast, PCNet faced challenges, especially with stringent QoS demands.

Training times for the NNs were 46.408 seconds for PCNet and 49.717 seconds for the proposed model. The additional time for the proposed model is due to calculations at the **Lambda** layer to ensure the power profile $\mathbf{P}$ meets both SINR and power constraints as outlined in Equation (3.5), related to Equation (3.22) using $\widehat{\mathbf{P}}$, A^{-1} and ν.

PCNet's training involves a soft-loss function that adds a penalty term to the sum rate. A high penalty value enhances QoS adherence but could lower the average sum rate. Conversely, a low penalty value might elevate the sum rate but risk QoS breaches. Fine-tuning this penalty factor, which acts like an extra hyperparameter, can be tedious. For these comparisons, PCNet's penalty was optimized for maximum sum rates.

The proposed DUL-based DNN method is based on an equivalent constrained optimization problem, utilizing the properties of a monotone matrix. This model adeptly manages polytope constraints, leveraging a monotone matrix's unique characteristic. Notably, the DUL approach guarantees solution feasibility without iterations, projections, or additional hyperparameter tuning. Unlike PCNet, the proposed algorithm seeks power profile values both at and within the boundaries of the box polytope constraint, leading to better performance under stringent QoS compared to loose QoS.

4.2.2 Training with Enhanced Generalization Capacity

Comparison plots on ASR (4.20 and 4.21) between the proposed model and PCNet+ are presented in the next two pages for different numbers of K with $EsN0 = 0$ dB and $SINR_{min} = 0.2$ for all receiver antennas. Respective data tables are also provided for reference. Tables 4.13 and 4.14 are for ASR for both models.

4.2.2.1 ASR for $EsN0 = 0$ dB with $\mathbf{SINR_{min}}$= 0.2 for all Receivers

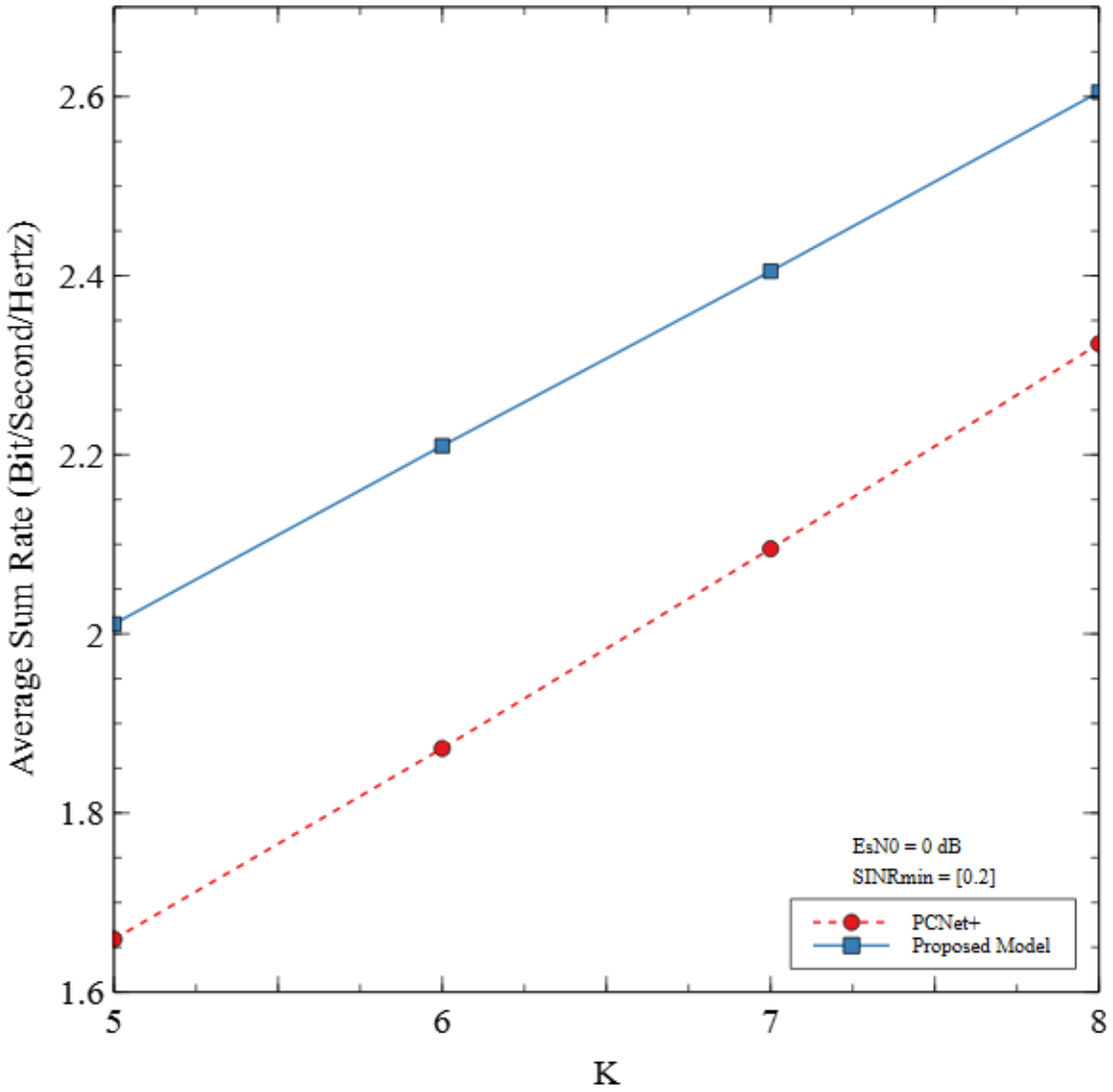

Figure 4.20: Average Sum Rate Plot for different numbers of K with EsN0 = 0 dB and $\mathbf{SINR_{min}}$ = 0.2 for all receiver antennas

Table 4.13: Average Sum Rates (Bit/Second/Hertz) for different numbers of K with EsN0 = 0 dB and $\mathbf{SINR_{min}}$ = 0.2 for all receiver antennas

Model	K = 5	K = 6	K = 7	K = 8
PCNet+	1.659	1.872	2.095	2.324
Proposed	2.011	2.210	2.405	2.605

4.2.2.2 ASR for $EsN0$ = 20 dB with $\mathbf{SINR_{min}}$= 0.2 for all Receivers

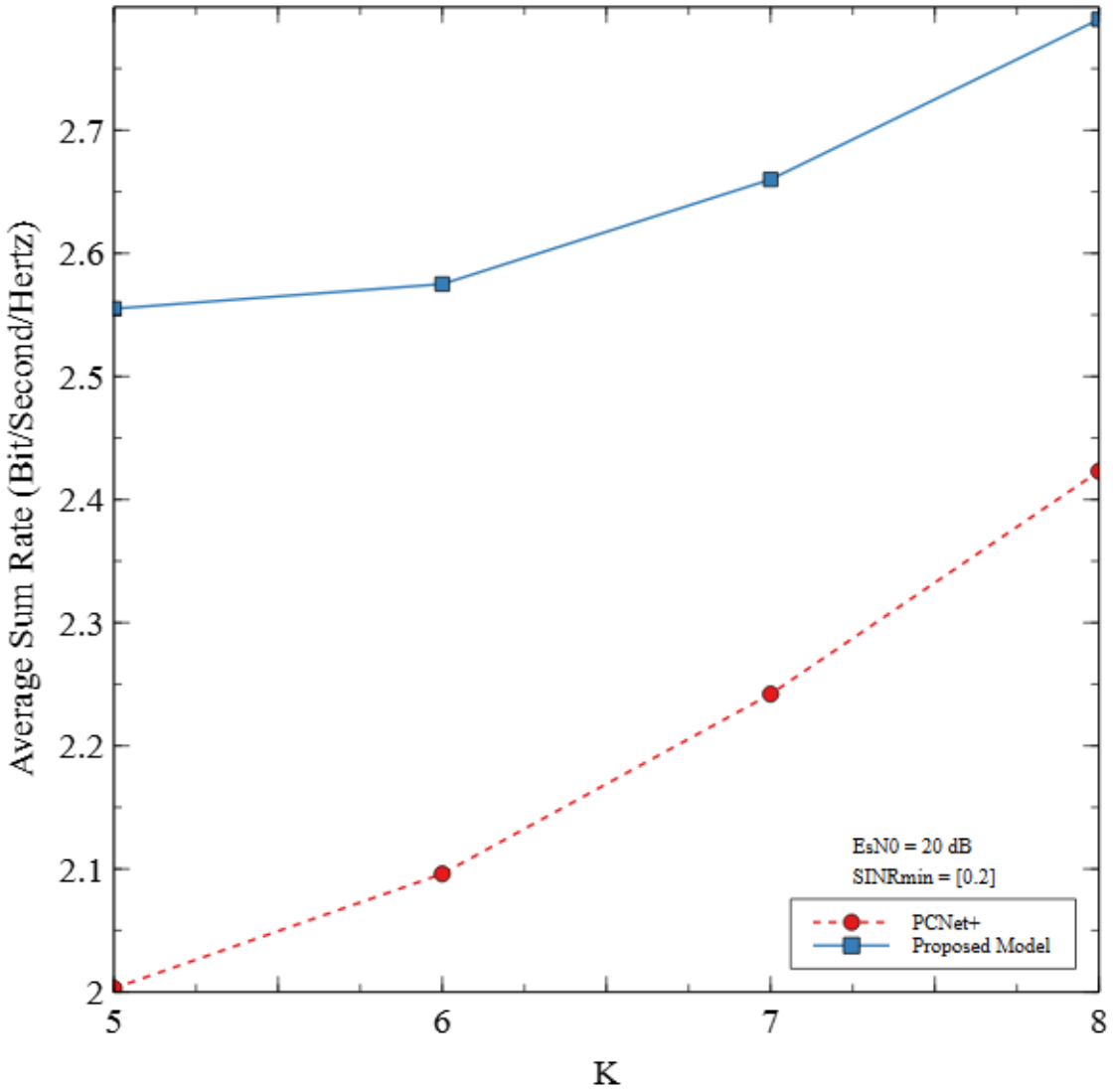

Figure 4.21: Average Sum Rate Plot for different numbers of K with EsN0 = 20 dB and $\mathbf{SINR_{min}}$ = 0.2 for all receiver antennas

Table 4.14: Average Sum Rates (Bit/Second/Hertz) for different numbers of K with EsN0 = 20 dB and $\mathbf{SINR_{min}}$ = 0.2 for all receiver antennas

Model	K = 5	K = 6	K = 7	K = 8
PCNet+	2.003	2.096	2.242	2.423
Proposed	2.555	2.575	2.660	2.790

Training individual NNs for each $EsN0$ typically yields better performance than a single adaptive network. While improved generality may reduce performance, it is still desirable. The figures 4.20 and 4.21 reveal that the DUL-based DNN model surpasses the PCNet+ model in ASR for both $EsN0$ scenarios, consistent with enhanced generalization capacity. Training NNs took 346.42 and 401.95 seconds for PCNet+ and the proposed model, respectively.

Appendix B is attached for comprehensive details regarding the simulation results. All Python code in Colab is included in Appendix C.

4.3 Summary

The performance of the PCNet/PCNet+ and the proposed DUL-based DNN model was mainly compared based on constraint adherence or hit rate valuations, and average sum rates. The proposed model distinguishes itself by ensuring complete satisfaction of all constraints and a better average sum rate whenever the problem is feasible. PCNet/PCNet+, in contrast, relies on a penalty factor λ as a mechanism to augment its hit rate, operating under the principle that the larger the penalty for constraint violations, the higher the consequent hit rate, and vice versa. However, PCNet/PCNet+ encounters difficulties maintaining high hit rates when QoS requirements become increasingly stringent. Additionally, it is noteworthy that changes in noise level $EsN0$ exert minimal impact on the hit rates of PCNet/PCNet+. Contrarily, the proposed DUL-based scheme exhibits exceptional performance, consistently achieving a 100% hit rate, irrespective of the QoS constraints and $EsN0$ noise levels. In all scenarios and across the complete $EsN0$ noise level range, the proposed DUL-based scheme surpasses PCNet/PCNet+'s performance, except for the slightly extended training time needed to train its NNs. However, this superior performance is achieved without extensive hyperparameter tuning, unlike PCNet/PCNet+, which requires careful selection of the penalty factor λ. Moreover, the DUL-based scheme eliminates the need for feasibility checks on its output or the application of heuristic solutions. This streamlined approach leads to a reduction in computational overhead, making the DUL-based scheme a more efficient solution when compared to PCNet/PCNet+.

Chapter 5

Conclusion

5.1 Overview

This thesis embarks on an in-depth exploration of DUL methodologies, emphasizing their ability to augment the sum rate in D2D networks. At the heart of this exploration lies a substantial contribution, wherein an inventive DUL-based optimizer is put forth. This innovative model takes strides in efficiently tackling the challenge of loss function minimization, grappling simultaneously with box and polytope constraints tied to a monotone matrix.

The strength of this proposed model is rooted in its robustness and ease of implementation. Leveraging Sigmoid activation functions within the output layer, the model encapsulates a unique configuration that can deftly handle complex computational tasks. Notably, this design enables the model to fulfill both box and polytope constraints concurrently while driving the optimization of the loss function.

With its pioneering approach, the model showcases the potential of DUL methodologies in enhancing network efficiency, setting the stage for further research and advancement in this field. The study lays out a solid foundation for future endeavors seeking to unlock the untapped potential of DUL in optimizing D2D networks, signifying a critical step forward in wireless communication networks.

5.2 Key Findings

The research revealed a noteworthy improvement in the sum rate when deploying the DUL model compared to conventional D2D communication methods. It has been shown that the model can effectively learn complex data patterns and make informed, real-time decisions regarding power control to transmit, resulting in more efficient use of the available spectrum.

The model's performance remained consistent across varying network channel parameters, showcasing its adaptability. Deep learning techniques can support more dynamic and responsive management of D2D communications, resulting in better quality of service for users and enhanced network capacity.

During meticulous analysis, a salient feature of the proposed DUL-based optimizer emerges: it reliably ensures an exceptional 100% constraint satisfaction rate. This level of consistency sets it apart from its contemporaries, especially compared to existing DUL-based methodologies such as PCNet.

Additionally, the proposed methodology demonstrates a clear superiority in performance when juxtaposed with PCNet in terms of the average sum rate. The prowess of this method does not end there; it extends into the realm of simplicity and efficiency. Unlike many other methods, the proposed approach successfully eliminates the need for additional hyperparameters or heuristic solutions, often deemed necessary in methods akin to PCNet.

This streamlined operation not only makes the proposed approach more accessible but also improves its efficiency. The proposed methodology's exemplary performance, simplicity, and high efficiency underscore the potential of DUL as a robust tool for optimizing D2D networks. It signifies a remarkable advancement in the field, opening up new possibilities for future research and development in D2D network optimization.

The superiority of this approach has been verified through comprehensive study, meticulous data analysis, and simulation, underscoring the considerable potential of machine-learning techniques in managing D2D networks and enhancing their perfor-

mance. The thesis underscores the potential for these innovative techniques to manage D2D networks more efficiently and significantly improve their performance, thereby providing a substantial contribution to the field.

5.3 Implications

Employing DUL-based DNNs for D2D networks holds considerable promise. The potential ramifications of this transition are manifold:

1. It paves the way for an enhanced user experience by optimizing communication processes and reducing latency.
2. It could lead to more efficient resource management by utilizing machine learning algorithms to optimally allocate network resources, thus minimizing waste and maximizing network efficiency.
3. Adopting DUL can support the growth and scalability of D2D networks, making them more adaptable and resilient in the face of the rapidly evolving demands of today's digital landscape.

The outcomes of this research hint at a paradigm shift in how D2D networks are managed, steering away from the conventional and static processes currently employed for resource allocation and interference management. Traditional methods often rely on pre-defined processes that may not cater to network conditions' dynamic and unpredictable nature. In contrast, the new paradigm suggests using flexible and adaptive models powered by machine learning. These models can learn from past experiences and continually evolve their strategies, dynamically adapting to changing network conditions.

It would enable a more proactive and responsive approach where networks can anticipate changes and adapt their strategies in real-time. It signifies transforming from a reactive model to a proactive, predictive network management model. This shift could dramatically improve the efficiency, reliability, and user experience of D2D

networks, making them more suitable for a world increasingly reliant on robust and efficient digital communication platforms.

5.4 Limitations and Future Research

Despite the promising findings, the study also identified potential challenges, notably computational complexity with real-world application in 5G and beyond networks.

The current model operates under the assumption of ideal channel estimation. It, however, tends to be unrealistic in real-world scenarios where numerous variables can influence the estimation process. Therefore, future research endeavors should focus on enhancing the model's robustness against potential errors in channel estimation. It would pave the way for a more accurate and reliable model that can withstand practical challenges.

The proposed model demonstrates a commendable ability for generalization, particularly in the context of background noise power. However, integrating other system parameters like the number of users and the distribution of channel coefficients presents a significant challenge. Developing a model that can generalize these system parameters effectively and accurately is a crucial research direction that warrants exploration.

The current framework of the proposed scheme is centralized, meaning all processing and decision-making happen in one central unit. It may present limitations when scaling or when the network demands a more distributed approach. Consequently, another significant and intriguing line of inquiry lies in developing a distributed version of this scheme. While this is a challenging venture, its success could significantly enhance the scalability and efficiency of the system, thereby making it more adaptable to diverse network conditions and requirements.

Additionally, the process of selecting appropriate hyperparameters for the model is an area that requires further investigation. Developing strategies for hyperparameter selection that can optimize the model's performance will be essential for future

research.

These concerns present an avenue for future research and must be addressed to fully realize the benefits of DUL in real-world D2D networks. Further research might also investigate the potential of other deep learning architectures or ensemble models that could further enhance sum-rate optimization.

5.5 Final Words

This research study has shed light on the remarkable potential of DUL as a potent tool for optimizing the sum rate in D2D networks. Undeniably, the current landscape presents many challenges and limitations that need to be addressed. However, the prospective advantages of successfully implementing DUL in this context indicate that this field is ripe for further exploration and progressive development.

As the realm of D2D communication technologies continues to evolve, so should the methodologies used to enhance their performance. The employment of DUL in this sphere represents an exciting frontier in network optimization. The benefits of harnessing DUL stretch far beyond traditional techniques, promising superior network performance and more effective utilization of D2D networks. By dynamically learning and adapting to network conditions, DUL can revolutionize how we manage and optimize these networks.

This promising trajectory of DUL-based optimization strategies for D2D networks paints an exciting future, suggesting a shift towards more adaptive and efficient network management. As we continue to explore the nuances of this technology and build on the foundation provided by this study, the D2D communication field stands to make significant strides in network optimization, performance enhancement, and resource utilization.

Bibliography

[1] J. Liu, N. Kato, J. Ma, and N. Kadowaki, "Device-to-Device Communication in LTE-Advanced Networks: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 1923–1940, 2015.

[2] S.-Y. Lien, C.-C. Chien, F.-M. Tseng, and T.-C. Ho, "3GPP device-to-device communications for beyond 4G cellular networks," *IEEE Communications Magazine*, vol. 54, no. 3, pp. 29–35, 2016.

[3] M. S. M. Gismalla, A. I. Azmi, M. R. B. Salim, M. F. L. Abdullah, F. Iqbal, W. A. Mabrouk, M. B. Othman, A. Y. I. Ashyap, and A. S. M. Supa'at, "Survey on Device to Device (D2D) Communication for 5GB/6G Networks: Concept, Applications, Challenges, and Future Directions," *IEEE Access*, vol. 10, pp. 30 792–30 821, 2022.

[4] X. Shen, "Device-to-device Communication in 5G Cellular Networks," *IEEE Network*, vol. 29, no. 2, pp. 2–3, 2015.

[5] C. Liu and B. Natarajan, "Power-Aware Maximization of Ergodic Capacity in D2D Underlay Networks," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 3, pp. 2727–2739, 2017.

[6] A. Mehmood, O. Waqar, and M. M. Ur Rahman, "Throughput maximization of an IRS-assisted wireless powered network with interference: A deep unsupervised learning approach," *Physical Communication*, vol. 51, p. 101558, 2022, doi: https://doi.org/10.1016/j.phycom.2021.101558.

[7] A. Kaushik, M. Alizadeh, O. Waqar, and H. Tabassum, "Deep Unsupervised Learning for Generalized Assignment Problems: A Case-Study of User-

Association in Wireless Networks," in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2021, pp. 1–6.

[8] H. Song, M. Zhang, J. Gao, and C. Zhong, "Unsupervised Learning-Based Joint Active and Passive Beamforming Design for Reconfigurable Intelligent Surfaces Aided Wireless Networks," *IEEE Communications Letters*, vol. 25, no. 3, pp. 892–896, 2021.

[9] J. Gao, C. Zhong, X. Chen, H. Lin, and Z. Zhang, "Unsupervised Learning for Passive Beamforming," *IEEE Communications Letters*, vol. 24, no. 5, pp. 1052–1056, 2020.

[10] H. Huang, W. Xia, J. Xiong, J. Yang, G. Zheng, and X. Zhu, "Unsupervised Learning-Based Fast Beamforming Design for Downlink MIMO," *IEEE Access*, vol. 7, pp. 7599–7605, 2019.

[11] Y. Li, S. Han, and C. Yang, "Multicell Power Control Under Rate Constraints With Deep Learning," *IEEE Transactions on Wireless Communications*, vol. 20, pp. 7813–7825, 12 2021.

[12] T. Frerix, M. Niesner, and D. Cremers, "Homogeneous Linear Inequality Constraints for Neural Network Activations," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2020, pp. 3229–3234. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPRW50498.2020.00382

[13] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004.

[14] P. Donti, D. Rolnick, and J. Kolter, "DC3: A learning method for optimization with hard constraints," *International Conference on Learning Representations 2021*, 04 2021.

[15] M. Haroon, Z. Abbas, F. Muhammad, and G. Abbas, "Coverage Analysis of Cell Edge Users in Heterogeneous Wireless Networks using Stienen's Model and RFA Scheme," *International Journal of Communication Systems*, 07 2020.

[16] C. Sudhamani, M. Roslee, J. J. Tiang, and A. U. Rehman, "A Survey on 5G Coverage Improvement Techniques: Issues and Future Challenges," *Sensors*, vol. 23, no. 4, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/4/2356

[17] Y.-D. Lin and Y.-C. Hsu, "Multihop cellular: a new architecture for wireless communications," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, vol. 3, 2000, pp. 1273–1282 vol.3.

[18] X. Wu, S. Tavildar, S. Shakkottai, T. Richardson, J. Li, R. Laroia, and A. Jovicic, "FlashLinq: A Synchronous Distributed Scheduler for Peer-to-Peer Ad Hoc Networks," *IEEE/ACM Transactions on Networking*, vol. 21, no. 4, pp. 1215–1228, 2013.

[19] J. Iqbal, M. A. Iqbal, A. Ahmad, M. Khan, A. Qamar, and K. Han, "Comparison of Spectral Efficiency Techniques in Device-to-Device Communication for 5G," *IEEE Access*, vol. 7, pp. 57 440–57 449, 2019.

[20] A. Asadi, Q. Wang, and V. Mancuso, "A Survey on Device-to-Device Communication in Cellular Networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1801–1819, 2014.

[21] S. Xu, H. Wang, and T. Chen, "Effective Interference Cancellation Mechanisms for D2D Communication in Multi-Cell Cellular Networks," in *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)*, 2012, pp. 1–5.

[22] N. Naderializadeh and A. S. Avestimehr, "ITLinQ: A new approach for spectrum sharing in device-to-device communication systems," in *2014 IEEE International Symposium on Information Theory*, 2014, pp. 1573–1577.

[23] F. Hussain, M. Y. Hassan, M. S. Hossen, and S. Choudhury, "An optimal resource allocation algorithm for D2D communication underlaying cellular networks," in *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2017, pp. 867–872.

[24] S. Lin, L. Fu, K. Li, and Y. Li, "Sum-Rate Optimization for Device-to-Device Communications over Rayleigh Fading Channel," in *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, 2017, pp. 1–6.

[25] M. Chiang, C. W. Tan, D. P. Palomar, D. O'neill, and D. Julian, "Power Control By Geometric Programming," *IEEE Transactions on Wireless Communications*, vol. 6, no. 7, pp. 2640–2651, 2007.

[26] L. P. Qian, Y. J. Zhang, and J. Huang, "MAPEL: Achieving global optimality for a non-convex wireless power control problem," *IEEE Transactions on Wireless Communications*, vol. 8, no. 3, pp. 1553–1563, 2009.

[27] L. Liu, R. Zhang, and K.-C. Chua, "Achieving Global Optimality for Weighted Sum-Rate Maximization in the K-User Gaussian Interference Channel with Multiple Antennas," *IEEE Transactions on Wireless Communications*, vol. 11, no. 5, pp. 1933–1945, 2012.

[28] H. Sun, X. Chen, Q. Shi, M. Hong, X. Fu, and N. D. Sidiropoulos, "Learning to Optimize: Training Deep Neural Networks for Interference Management," *IEEE Transactions on Signal Processing*, vol. 66, no. 20, pp. 5438–5453, 2018.

[29] W. Lee, M. Kim, and D.-H. Cho, "Deep Power Control: Transmit Power Control Scheme Based on Convolutional Neural Network," *IEEE Communications Letters*, vol. 22, no. 6, pp. 1276–1279, 2018.

[30] D. Ron and J.-R. Lee, "DRL-Based Sum-Rate Maximization in D2D Communication Underlaid Uplink Cellular Networks," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 10, pp. 11 121–11 126, 2021.

[31] F. Liang, C. Shen, W. Yu, and F. Wu, "Towards Optimal Power Control via Ensembling Deep Neural Networks," *IEEE Transactions on Communications*, vol. 68, no. 3, pp. 1760–1776, 2020.

[32] B. Lea, D. Shome, O. Waqar, and J. Tomal, "Sum rate maximization of D2D networks with energy constrained UAVs through deep unsupervised learning," in *2021 IEEE 12th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2021, pp. 0453–0459.

[33] D. Kim, H. Jung, and I.-H. Lee, "Deep Learning-Based Power Control Scheme With Partial Channel Information in Overlay Device-to-Device Communication Systems," *IEEE Access*, vol. 9, pp. 122 125–122 137, 2021.

[34] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208

[35] Q. Shi, M. Razaviyayn, Z.-Q. Luo, and C. He, "An Iteratively Weighted MMSE Approach to Distributed Sum-Utility Maximization for a MIMO Interfering Broadcast Channel," *IEEE Transactions on Signal Processing*, vol. 59, no. 9, pp. 4331–4340, 2011.

[36] C. S. Chen, K. W. Shum, and C. W. Sung, "Round-robin power control for the weighted sum rate maximisation of wireless networks over multiple interfering links," *European Transactions on Telecommunications*, vol. 22, no. 8, pp. 458–470, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.1496

[37] O. L. Mangasarian, "Characterizations of real matrices of monotone kind," *SIAM Review*, vol. 10, no. 4, pp. 439–441, 1968. [Online]. Available: https://doi.org/10.1137/1010095

[38] M. Chiang, P. Hande, T. Lan, and C. W. Tan, *Power Control in Wireless Cellular Networks*. New Foundations and Trends, 2008, vol. 2, no. 4.

Appendix A

Feasible Datasets for the Transmission Channel Parameters

In total, 55 feasible different datasets have been generated to analyze the model's performance, which totals 22.1 GB.

File Type: **.csv**

Link for datasets:

https://1drv.ms/f/s!ApktXr3Tu2RWgtQPgQM8qhtPgzYmgw?e=KNdtfu

There are three groups of datasets, as follows:

1. **Dataset Group A:** 35 datasets for K = 5, EsN0 = [0 dB, 10 dB, 20 dB, 30 dB, 40 dB, 50 dB, 60 dB], and five scenarios of $\mathbf{SINR_{min}} = 0.5$. The total size of these datasets is 11.1 GB.
2. **Dataset Group B:** 20 datasets in total for EsN0 = [0 dB, 10 dB, 20 dB, 30 dB, 40 dB] for each K = [5, 6, 7, 8], with all $\mathbf{SINR_{min}} = 0.2$, e.g., for K = 5, $\mathbf{SINR_{min}} = [0.2, 0.2, 0.2, 0.2, 0.2]$. The total size of these datasets is 11 GB.
3. **Dataset Group C:** 25 datasets out of 35 datasets from Group A, i.e., K = 5, and five scenarios of $\mathbf{SINR_{min}} = 0.5$ but EsN0 = [0 dB, 10 dB, 20 dB, 30 dB, 40 dB]

Appendix B

Simulation Results

Tables B1 to B14 with Results

Notes:

- "Basic" stands for basic model for finding $\widehat{\mathbf{P}}$ through Equation (3.26)
- "Model A" determines $\mathbf{P}$ using $\widehat{\mathbf{P}} + \mathbf{A}^{-1}\mu$, where $0 \leq \mu_k \leq \min_{1 \leq l \leq K} \frac{P_{\max} - \widehat{P}_l}{[\sum_{k=1}^{K} \mathbf{a}_k]_l}$ and normalizes it such that the maximum of powers is $P_{\max}$.
- Hit Rate (HR) = 1 - Constraint Violation Probability (CVP)
- CVP for Basic Model = CVP for Model A = CVP for Proposed Model = 0.00%
- Parameters: Epochs = 50; Batch Size = 1,000 and Learning Rate = 0.0001
- PCNet+, a modified version of PCNet, is introduced in [31] to improve generalization by including noise power, σ^2, alongside channel coefficients as input.

Table B.1: PCNet CVP% and Average Sum Rate (in Bit/Second/Hertz) from all four Models for K = 5 and **$\mathbf{SINR_{min}}$ = (0.5, 0.0, 0.0, 0.0, 0.0)**

EsN0	Basic	λ	CVP	PCNet	Model A	Proposed
0 dB	0.585	$\lambda = 0$	46.08%	1.654	1.821	1.945
		$\lambda = 5$	16.08%	1.845		
		$\lambda = 10$	9.46%	1.860		
		$\lambda = 15$	7.01%	1.859		
		$\lambda = 20$	5.68%	1.851		
		$\lambda = 25$	4.62%	1.849		
10 dB	0.585	$\lambda = 0$	72.16%	3.188	3.623	3.678
		$\lambda = 5$	7.48%	3.580		
		$\lambda = 10$	3.65%	3.567		
		$\lambda = 15$	2.20%	3.552		
		$\lambda = 20$	1.61%	3.542		
		$\lambda = 25$	1.22%	3.529		
20 dB	0.585	$\lambda = 0$	48.09%	5.924	6.006	6.102
		$\lambda = 5$	0.76%	6.078		
		$\lambda = 10$	0.52%	6.072		
		$\lambda = 15$	0.30%	6.068		
		$\lambda = 20$	0.23%	6.053		
		$\lambda = 25$	0.16%	6.049		
30 dB	0.585	$\lambda = 0$	53.80%	9.139	9.150	9.154
		$\lambda = 5$	0.16%	9.156		
		$\lambda = 10$	0.10%	9.153		
		$\lambda = 15$	0.00%	9.144		
		$\lambda = 20$	0.00%	9.137		
		$\lambda = 25$	0.00%	9.127		
40 dB	0.585	$\lambda = 0$	99.75%	12.443	12.458	12.459
		$\lambda = 5$	0.16%	12.466		
		$\lambda = 10$	0.13%	12.462		
		$\lambda = 15$	0.10%	12.443		
		$\lambda = 20$	0.01%	12.435		
		$\lambda = 25$	0.00%	12.415		
50 dB	0.585	$\lambda = 0$	99.89%	15.749	15.746	15.751
		$\lambda = 5$	0.18%	15.761		
		$\lambda = 10$	0.06%	15.748		
		$\lambda = 15$	0.02%	15.731		
		$\lambda = 20$	0.00%	15.725		
		$\lambda = 25$	0.00%	15.705		
60 dB	0.585	$\lambda = 0$	99.81%	19.068	19.094	19.085
		$\lambda = 5$	0.31%	19.096		
		$\lambda = 10$	0.03%	19.029		
		$\lambda = 15$	0.00%	19.027		
		$\lambda = 20$	0.00%	18.980		
		$\lambda = 25$	0.00%	18.975		

Table B.2: PCNet CVP% and Average Sum Rate (in Bit/Second/Hertz) from all four Models for K = 5 and **$\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.0, 0.0, 0.0)**

EsN0	Basic	λ	CVP	PCNet	Model A	Proposed
0 dB	1.170	$\lambda = 0$	66.05%	1.863	2.050	2.147
		$\lambda = 5$	31.84%	2.008		
		$\lambda = 10$	21.88%	2.020		
		$\lambda = 15$	16.18%	2.018		
		$\lambda = 20$	14.29%	2.011		
		$\lambda = 25$	13.77%	2.000		
10 dB	1.170	$\lambda = 0$	91.66%	2.573	3.068	3.107
		$\lambda = 5$	20.97%	2.878		
		$\lambda = 10$	13.42%	2.872		
		$\lambda = 15$	9.92%	2.855		
		$\lambda = 20$	9.04%	2.855		
		$\lambda = 25$	7.62%	2.845		
20 dB	1.170	$\lambda = 0$	99.74%	3.105	3.690	3.726
		$\lambda = 5$	20.32%	3.322		
		$\lambda = 10$	10.87%	3.313		
		$\lambda = 15$	7.59%	3.294		
		$\lambda = 20$	6.84%	3.289		
		$\lambda = 25$	5.90%	3.269		
30 dB	1.170	$\lambda = 0$	99.68%	3.273	3.936	3.946
		$\lambda = 5$	31.78%	3.525		
		$\lambda = 10$	11.22%	3.439		
		$\lambda = 15$	7.68%	3.422		
		$\lambda = 20$	6.50%	3.391		
		$\lambda = 25$	6.01%	3.369		
40 dB	1.170	$\lambda = 0$	99.89%	3.311	4.016	4.019
		$\lambda = 5$	29.78%	3.631		
		$\lambda = 10$	10.36%	3.472		
		$\lambda = 15$	7.47%	3.436		
		$\lambda = 20$	6.31%	3.405		
		$\lambda = 25$	4.35%	3.390		
50 dB	1.170	$\lambda = 0$	100.00%	3.310	4.022	4.031
		$\lambda = 5$	24.44%	3.457		
		$\lambda = 10$	9.87%	3.455		
		$\lambda = 15$	7.52%	3.442		
		$\lambda = 20$	6.44%	3.397		
		$\lambda = 25$	4.38%	3.383		
60 dB	1.170	$\lambda = 0$	99.84%	3.300	4.015	4.015
		$\lambda = 5$	28.88%	3.606		
		$\lambda = 10$	10.25%	3.452		
		$\lambda = 15$	8.60%	3.446		
		$\lambda = 20$	5.58%	3.397		
		$\lambda = 25$	5.36%	3.390		

Table B.3: PCNet CVP% and Average Sum Rate (in Bit/Second/Hertz) from all four Models for K = 5 and **$\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.0, 0.0)**

EsN0	Basic	λ	CVP	PCNet	Model A	Proposed
0 dB	1.755	$\lambda = 0$	82.22%	2.245	2.443	2.524
		$\lambda = 5$	52.16%	2.353		
		$\lambda = 10$	39.44%	2.371		
		$\lambda = 15$	34.12%	2.368		
		$\lambda = 20$	31.00%	2.354		
		$\lambda = 25$	30.38%	2.317		
10 dB	1.755	$\lambda = 0$	99.53%	2.615	3.051	3.098
		$\lambda = 5$	43.46%	2.804		
		$\lambda = 10$	31.44%	2.791		
		$\lambda = 15$	25.82%	2.756		
		$\lambda = 20$	23.29%	2.739		
		$\lambda = 25$	21.91%	2.734		
20 dB	1.755	$\lambda = 0$	99.99%	2.807	3.412	3.430
		$\lambda = 5$	43.70%	2.997		
		$\lambda = 10$	26.90%	2.968		
		$\lambda = 15$	23.42%	2.941		
		$\lambda = 20$	20.77%	2.904		
		$\lambda = 25$	18.83%	2.886		
30 dB	1.755	$\lambda = 0$	99.76%	2.856	3.510	3.512
		$\lambda = 5$	48.10%	3.033		
		$\lambda = 10$	24.90%	3.026		
		$\lambda = 15$	22.01%	2.999		
		$\lambda = 20$	18.03%	2.955		
		$\lambda = 25$	16.83%	2.170		
40 dB	1.755	$\lambda = 0$	100.00%	2.840	3.500	3.513
		$\lambda = 5$	47.49%	3.020		
		$\lambda = 10$	26.85%	3.016		
		$\lambda = 15$	22.11%	2.974		
		$\lambda = 20$	20.30%	2.933		
		$\lambda = 25$	20.11%	2.920		
50 dB	1.755	$\lambda = 0$	100.00%	2.845	3.497	3.521
		$\lambda = 5$	48.74%	3.023		
		$\lambda = 10$	26.06%	2.990		
		$\lambda = 15$	22.81%	2.966		
		$\lambda = 20$	20.43%	2.949		
		$\lambda = 25$	19.38%	2.932		
60 dB	1.755	$\lambda = 0$	100.00%	2.842	3.506	3.519
		$\lambda = 5$	45.99%	3.040		
		$\lambda = 10$	26.72%	2.997		
		$\lambda = 15$	20.92%	2.976		
		$\lambda = 20$	19.18%	2.948		
		$\lambda = 25$	18.64%	2.921		

Table B.4: PCNet CVP% and Average Sum Rate (in Bit/Second/Hertz) from all four Models for K = 5 and **$\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.0)**

EsN0	Basic	λ	CVP	PCNet	Model A	Proposed
0 dB	2.340	$\lambda = 0$	94.28%	2.705	2.914	2.990
		$\lambda = 5$	70.42%	2.791		
		$\lambda = 10$	56.02%	2.804		
		$\lambda = 15$	50.57%	2.796		
		$\lambda = 20$	46.72%	2.785		
		$\lambda = 25$	45.93%	2.768		
10 dB	2.340	$\lambda = 0$	99.98%	2.948	3.313	3.350
		$\lambda = 5$	62.87%	3.054		
		$\lambda = 10$	47.44%	3.040		
		$\lambda = 15$	46.41%	3.019		
		$\lambda = 20$	44.63%	3.002		
		$\lambda = 25$	43.01%	2.996		
20 dB	2.340	$\lambda = 0$	100.00%	3.028	3.486	3.497
		$\lambda = 5$	61.18%	3.136		
		$\lambda = 10$	45.17%	3.116		
		$\lambda = 15$	41.78%	3.100		
		$\lambda = 20$	41.72%	3.085		
		$\lambda = 25$	40.34%	3.080		
30 dB	2.340	$\lambda = 0$	100.00%	3.036	3.516	3.524
		$\lambda = 5$	60.18%	3.155		
		$\lambda = 10$	48.29%	3.125		
		$\lambda = 15$	43.04%	3.115		
		$\lambda = 20$	40.10%	3.102		
		$\lambda = 25$	38.08%	3.100		
40 dB	2.340	$\lambda = 0$	100.00%	3.040	3.527	3.531
		$\lambda = 5$	58.04%	3.161		
		$\lambda = 10$	44.84%	3.151		
		$\lambda = 15$	43.48%	3.123		
		$\lambda = 20$	42.46%	3.107		
		$\lambda = 25$	41.28%	3.104		
50 dB	2.340	$\lambda = 0$	100.00%	3.036	3.521	3.525
		$\lambda = 5$	57.77%	3.159		
		$\lambda = 10$	47.16%	3.127		
		$\lambda = 15$	42.56%	3.112		
		$\lambda = 20$	41.09%	3.106		
		$\lambda = 25$	38.20%	3.101		
60 dB	2.340	$\lambda = 0$	100.00%	3.028	3.509	3.513
		$\lambda = 5$	58.52%	3.152		
		$\lambda = 10$	47.49%	3.119		
		$\lambda = 15$	42.56%	3.108		
		$\lambda = 20$	40.72%	3.098		
		$\lambda = 25$	40.00%	3.092		

Table B.5: PCNet CVP% and Average Sum Rate (in Bit/Second/Hertz) from all four Models for K = 5 and **$\mathbf{SINR_{min}}$ = (0.5, 0.5, 0.5, 0.5, 0.5)**

EsN0	Basic	λ	CVP	PCNet	Model A	Proposed
0 dB	2.925	$\lambda = 0$	98.82%	3.215	3.415	3.486
		$\lambda = 5$	80.63%	3.274		
		$\lambda = 10$	69.33%	3.278		
		$\lambda = 15$	65.48%	3.263		
		$\lambda = 20$	64.78%	3.258		
		$\lambda = 25$	63.01%	3.248		
10 dB	2.925	$\lambda = 0$	100.00%	3.376	3.677	3.712
		$\lambda = 5$	75.35%	3.429		
		$\lambda = 10$	68.86%	3.416		
		$\lambda = 15$	67.33%	3.407		
		$\lambda = 20$	66.16%	3.403		
		$\lambda = 25$	65.97%	3.398		
20 dB	2.925	$\lambda = 0$	100.00%	3.418	3.781	3.790
		$\lambda = 5$	72.98%	3.479		
		$\lambda = 10$	65.70%	3.465		
		$\lambda = 15$	65.08%	3.459		
		$\lambda = 20$	64.57%	3.456		
		$\lambda = 25$	63.78%	3.454		
30 dB	2.925	$\lambda = 0$	100.00%	3.420	3.801	3.803
		$\lambda = 5$	72.92%	3.487		
		$\lambda = 10$	66.56%	3.473		
		$\lambda = 15$	63.74%	3.469		
		$\lambda = 20$	63.62%	3.466		
		$\lambda = 25$	62.91%	3.463		
40 dB	2.925	$\lambda = 0$	100.00%	3.422	3.802	3.804
		$\lambda = 5$	73.89%	3.488		
		$\lambda = 10$	65.13%	3.476		
		$\lambda = 15$	63.08%	3.471		
		$\lambda = 20$	62.55%	3.468		
		$\lambda = 25$	62.20%	3.466		
50 dB	2.925	$\lambda = 0$	100.00%	3.424	3.803	3.803
		$\lambda = 5$	73.78%	3.489		
		$\lambda = 10$	64.51%	3.478		
		$\lambda = 15$	64.40%	3.472		
		$\lambda = 20$	63.07%	3.468		
		$\lambda = 25$	62.56%	3.467		
60 dB	2.925	$\lambda = 0$	100.00%	3.424	3.805	3.804
		$\lambda = 5$	73.12%	3.488		
		$\lambda = 10$	64.83%	3.478		
		$\lambda = 15$	65.05%	3.470		
		$\lambda = 20$	64.76%	3.468		
		$\lambda = 25$	63.69%	3.466		

Table B.6: PCNet+ CVP% and Average Sum Rate (in Bit/Second/Hertz) from the two Models for K = 5 and **$\mathbf{SINR_{min}}$ = (0.2, 0.2, 0.2, 0.2, 0.2)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	93.28%	1.659	2.011
10 dB	$\lambda = 5$	78.74%	1.954	2.445
20 dB	$\lambda = 5$	78.07%	2.003	2.555
30 dB	$\lambda = 5$	78.01%	2.007	2.568
40 dB	$\lambda = 5$	77.75%	2.003	2.568

Table B.7: PCNet+ CVP% and Average Sum Rate (in Bit/Second/Hertz) from the two Models for K = 6 and **$\mathbf{SINR_{min}}$ = (0.2, 0.2, 0.2, 0.2, 0.2, 0.2)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	95.81%	1.872	2.210
10 dB	$\lambda = 5$	48.65%	2.858	3.415
20 dB	$\lambda = 5$	84.67%	2.096	2.575
30 dB	$\lambda = 5$	84.68%	2.098	2.585
40 dB	$\lambda = 5$	84.66%	2.103	2.593

Table B.8: PCNet+ CVP% and Average Sum Rate (in Bit/Second/Hertz) from the two Models for K = 7 and **$\mathbf{SINR_{min}}$ = (0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	95.49%	2.095	2.405
10 dB	$\lambda = 5$	88.84%	2.223	2.608
20 dB	$\lambda = 5$	89.18%	2.242	2.660
30 dB	$\lambda = 5$	89.54%	2.243	2.669
40 dB	$\lambda = 5$	89.66%	2.239	2.662

Table B.9: PCNet+ CVP% and Average Sum Rate (in Bit/Second/Hertz) from the two Models for K = 8 and **$\mathbf{SINR_{min}}$ = (0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	97.05%	2.324	2.608
10 dB	$\lambda = 5$	93.24%	2.414	2.761
20 dB	$\lambda = 5$	93.57%	2.423	2.790
30 dB	$\lambda = 5$	93.42%	2.426	2.798
40 dB	$\lambda = 5$	93.31%	2.425	2.792

Table B.10: CVP% and Average Sum Rate (in Bit/Second/Hertz) for the Models for K = 5 and $\mathbf{SINR_{min}}$ **= (0.5, 0.0, 0.0, 0.0, 0.0)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	0.13%	1.261	1.234
10 dB	$\lambda = 5$	0.07%	3.078	3.039
20 dB	$\lambda = 5$	0.08%	5.971	5.932
30 dB	$\lambda = 5$	0.05%	9.162	8.693
40 dB	$\lambda = 5$	0.06%	12.437	12.466

Table B.11: CVP% and Average Sum Rate (in Bit/Second/Hertz) for the Models for K = 5 and $\mathbf{SINR_{min}}$ **= (0.5, 0.5, 0.0, 0.0, 0.0)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	33.02%	1.977	2.002
10 dB	$\lambda = 5$	28.90%	2.873	3.046
20 dB	$\lambda = 5$	20.46%	3.338	3.704
30 dB	$\lambda = 5$	19.30%	3.458	3.913
40 dB	$\lambda = 5$	19.32%	3.489	3.980

Table B.12: CVP% and Average Sum Rate (in Bit/Second/Hertz) for the Models for K = 5 and $\mathbf{SINR_{min}}$ **= (0.5, 0.5, 0.5, 0.0, 0.0)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	73.93%	2.236	2.422
10 dB	$\lambda = 5$	49.71%	2.798	3.086
20 dB	$\lambda = 5$	50.74%	2.968	3.413
30 dB	$\lambda = 5$	51.59%	2.998	3.484
40 dB	$\lambda = 5$	52.40%	2.985	3.477

Table B.13: CVP% and Average Sum Rate (in Bit/Second/Hertz) for the Models for K = 5 and $\mathbf{SINR_{min}}$ **= (0.5, 0.5, 0.5, 0.5, 0.0)**

EsN0	λ	PCNet+ CVP	PCNet+	Proposed
0 dB	$\lambda = 5$	95.31%	2.687	2.936
10 dB	$\lambda = 5$	67.54%	3.047	3.352
20 dB	$\lambda = 5$	68.10%	3.117	3.494
30 dB	$\lambda = 5$	68.16%	3.123	3.509
40 dB	$\lambda = 5$	68.03%	3.125	3.518

Table B.14: CVP% and Average Sum Rate (in Bit/Second/Hertz) for the Models for K = 5 and $\mathbf{SINR_{min}}$ **= (0.5, 0.5, 0.5, 0.5, 0.5)**

EsN0	λ	**PCNet+ CVP**	**PCNet+**	**Proposed**
0 dB	$\lambda = 5$	87.93%	3.246	3.453
10 dB	$\lambda = 5$	89.91%	3.402	3.713
20 dB	$\lambda = 5$	89.56%	3.441	3.783
30 dB	$\lambda = 5$	89.65%	3.444	3.793
40 dB	$\lambda = 5$	89.40%	3.447	3.794

Appendix C

Codes on Google Colaboratory

List of code files (File Type: .ipynb):

1. Codes for generating feasible datasets for the transmission channel parameters
   (a) Codes to calculate the average sum rate for the basic model
2. Codes for analyzing the PCNet model
   (a) For training with a given background noise power
   (b) PCNet+ model: For enhanced generalization capacity
3. Codes for analyzing the Proposed Model
   (a) For training with a given background noise power
   (b) For enhanced generalization capacity
4. Codes for analyzing the Model A
   (a) Codes to calculate the average sum rate for the basic model

Notes:

- "Basic" model is for finding feasible power allocation $\widehat{\mathbf{P}}$ through Equation (3.26)
- "Model A" determines $\mathbf{P}$ using $\widehat{\mathbf{P}} + \mathbf{A}^{-1}\mu$, where $0 \leq \mu_k \leq \min\limits_{1 \leq l \leq K} \dfrac{P_{\max} - \widehat{P}_l}{[\sum_{k=1}^{K} \mathbf{a}_k]_l}$ and normalizes it such that the maximum of powers is $P_{\max}$.

C.1 Codes for generating feasible datasets for the transmission channel parameters

```
import numpy as np

## Number of transmitter-receiver pairs
K = 5

## Variances for noise signals
sigma_sqr_noise = np.array([1e-0, 1e-0, 1e-0, 1e-0, 1e-0], dtype =
    float)

## Minimum rate for the achievable SINR of multiple concurrent
## transmissions
SINR_P_min = np.array([0.5, 0.5, 0.5, 0.5, 0.5], dtype = float)

## Maximum transmit power
p_max = 1.0
```

```
## Function to generate the Circularly Symmetric Complex Gaussian (
    CSCG) distributions

def complex_gaussian(d_mean = 0, d_var = 1, n = 1000):
  # Draw random samples from a normal (Gaussian) distribution.
  # Parameters:
  #   loc = Mean or center of the distribution.
  #   scale = Standard deviation or spread or width of the
    distribution. Must be non-negative.
  #   size = int or tuple of ints, optional
  return np.random.normal(loc = d_mean, scale = np.sqrt(2*d_var)/2,
    size = (n, 2)).view(np.complex128)


## Function to generate the channel-coefficient matrix H
def generate_H(K, sigma_sqr_h, sample_size):
  hij = []
  for i in range(K):  # Total rows, i.e., total receivers or users
    hj =[]
```

```
    for j in range(K):  # Total columns, i.e., total transmitters
      h = complex_gaussian(d_mean = 0, d_var = sigma_sqr_h, n =
   sample_size)
      hj.append(h)
    hj = np.concatenate(hj, 1)
    hij.append(hj)
  hij = np.stack(hij, 1)
  return hij
```

```
## Create matrix H
H_size = int(1e6)
sigma_sqr_h = 1
# np.random.seed(0)
H = generate_H(K, sigma_sqr_h, H_size)
print(H.shape)
# print(H)
```

```
import numba as nb

## Function to compute the square of the absolute value of an array
## of complex numbers
@nb.vectorize([nb.float64(nb.complex128),nb.float32(nb.complex64)])
def cmplx_abs_sqr(cmplx_var):
  return cmplx_var.real**2 + cmplx_var.imag**2
```

```
## Function to generate the matrix B
def generate_B(H_size, K, SINR_P_min, H):
  Bij_list = []
  H_abs_sqr = cmplx_abs_sqr(H)
  for k in range(H_size):
    for i in range(K):  # Total rows
      Bj_list =[]
      for j in range(K):  # Total columns
        if i==j:
          B = 0
        else:
          B_temp = np.multiply(SINR_P_min[i], H_abs_sqr[k,i,j])
          B = np.divide(B_temp, H_abs_sqr[k,i,i])
        Bj_list.append(B)
      Bij_list.append(Bj_list)
```

```
  Bij_array = np.array(Bij_list)
  Bij = Bij_array.reshape((H_size, K, K)) # H_size X row X column
  return Bij
```

```
## Create matrix B
B = generate_B(H_size, K, SINR_P_min, H)
print(B.shape)
# print(B)
```

```
## Function to generate the vector u
def generate_u(H_size, K, SINR_P_min, sigma_sqr_noise, H):
  ui_list = []
  H_abs_sqr = cmplx_abs_sqr(H)
  for k in range(H_size):
    for i in range(K):  # Total rows, i.e., total transmitters
      u_temp = np.multiply(SINR_P_min[i], sigma_sqr_noise[i])
      u = np.divide(u_temp, H_abs_sqr[k,i,i])
      ui_list.append(u)
  ui_array = np.array(ui_list)
  ui = ui_array.reshape((H_size, K, 1)) # H_size X row X column
  return ui
```

```
## Create vector u
u = generate_u(H_size, K, SINR_P_min, sigma_sqr_noise, H)
print(u.shape)
# print(u)
```

```
## Finding indexes of H matrix with the hij set that satisfy
## constraint on the maximum transmit power p_max

count_var = 0
indx_F_H = []
indx_temp_F_H = []
p_hat_temp_list = []

for k in range(H_size):
  eigen_value, eigen_vector = np.linalg.eig(B[k])
  # print(eigen_value)
  if max(abs(eigen_value)) < 1:
    subtr = np.identity(K) - B[k,:,:]
    invr = np.linalg.inv(subtr)
```

```
    u_temp = u[k]
    p_temp = np.matmul(invr, u_temp)
    p_hat_temp_list.append(p_temp)
    indx_temp_F_H.append(k)
    count_var += 1

p_hat_temp_array = np.array(p_hat_temp_list)
p_hat_temp = p_hat_temp_array.reshape((count_var,K,1))
print(p_hat_temp.shape)
# print(p_hat_temp)


P = abs(p_hat_temp)
fcount = 0
p_hat_list = []
for n in range(count_var):
  P_max = np.amax(P[n])
  if P_max <= p_max:
    p = p_hat_temp[n]
    p_hat_list.append(p)
    indx_F_H.append(indx_temp_F_H[n])
    fcount += 1

p_hat_array = np.array(p_hat_list)
p_hat = p_hat_array.reshape((fcount,K,1))
# p_hat = p_hat_array.reshape((fcount,1,K))
print(p_hat.shape)
p_hat_size = p_hat.shape[0]
# print(p_hat)
```

```
## H matrix for a feasible power profile
F_H_size = len(indx_F_H)
F_H = np.empty((F_H_size, K, K), dtype = complex, order = 'C')

for i in range(F_H_size):
  j = indx_F_H[i]
  F_H[i] = H[j]

print(F_H.shape)
```

```
# print(F_H)
```

```
# ## Checking SINR_P for feasible H matrix
# F_H_abs_sqr = cmplx_abs_sqr(F_H)

# for k in range(F_H_size):
#   SINR_P_F_H_list = []
#   for i in range(K):
#     ph = 0
#     for j in range(K):
#       ph_j = np.multiply(p_hat[k,j], F_H_abs_sqr[k,i,j])
#       ph = ph + ph_j

#     numr = np.multiply(p_hat[k,i], F_H_abs_sqr[k,i,i])
#     dnumr = sigma_sqr_noise[i] + ph - numr
#     SINR_P_temp = np.divide(numr, dnumr)
#     SINR_P_F_H_list.append(SINR_P_temp)

#   SINR_P_F_H_array = np.array(SINR_P_F_H_list)
#   SINR_P_F_H = SINR_P_F_H_array.reshape((1, K))
#   print(SINR_P_F_H)
#   p_hat_t = p_hat[k].reshape((1, 1, K)) # H_size X row X column
#   print(p_hat_t)
```

```
## Saving 3D Numpy array to CSV file
# Saving feasible H matrix F_H
from numpy import savetxt

# Reshaping the array from 3D to 2D
F_H_2D = F_H.reshape(F_H.shape[0], -1)

# Saving reshaped array to file in "Files" of colab at left bar
# Can download the file in local drive
savetxt('F_H_2D.csv', F_H_2D, delimiter=',')
```

```
# ## Saving p_hat matrix to CSV file
# # from numpy import savetxt

# # Reshaping the array from 3D to 2D
# p_hat_2D = p_hat.reshape(p_hat.shape[0], -1)
```

```
6
7 # # Saving reshaped array to file in "Files" of colab at left bar
8 # # Can download the file in local drive
9 # savetxt('p_hat_2D.csv', p_hat_2D, delimiter=',')
```

C.1.1 Codes to calculate the average sum rate for the basic model

```
1 ## Function to split datasets for training, validation, and testing.
2 def split(np_array):
3   # data_size = np_array.shape[0]
4   # train_data_size = int(data_size * 0.8)
5   # valid_data_size = int(data_size * 0.1)
6   # test_data_size = int(data_size * 0.1)
7
8   train_data_size = int(200000)
9   valid_data_size = int(25000)
10  test_data_size = int(25000)
11
12  train_e_indx = train_data_size
13  valid_e_indx = train_e_indx + valid_data_size
14  test_e_indx = valid_e_indx + test_data_size
15  test_data_size_n = test_e_indx - valid_e_indx
16
17  row_count = np_array.shape[1]
18  column_count = np_array.shape[2]
19
20  train_data = np.empty((train_data_size, row_count, column_count),
     dtype = complex, order = 'C')
21  valid_data = np.empty((valid_data_size, row_count, column_count),
     dtype = complex, order = 'C')
22  test_data = np.empty((test_data_size_n, row_count, column_count),
     dtype = complex, order = 'C')
23
24  for i in range(train_e_indx):
25    train_data[i] = np_array[i]
26
```

```
  xv = 0
  for j in range(train_e_indx, valid_e_indx):
    valid_data[xv] = np_array[j]
    xv = xv + 1

  xt = 0
  for k in range(valid_e_indx, test_e_indx):
    test_data[xt] = np_array[k]
    xt = xt + 1

  # print(train_data.shape, valid_data.shape, test_data.shape)


  ## Training input will be the absolute value
  train_input = np.absolute(train_data)
  valid_input = np.absolute(valid_data)
  test_input = np.absolute(test_data)

  print(train_input.shape, valid_input.shape, test_input.shape)

  return [train_input, valid_input, test_input, test_data]
```

```
## Split F_H matrix
F_H_S = split(F_H)
train_input_F_H = F_H_S[0]
valid_input_F_H = F_H_S[1]
test_input_F_H = F_H_S[2]
test_data_F_H = F_H_S[3]
```

```
## Split p_hat vector
p_hat_S = split(p_hat)
train_input_p_hat = p_hat_S[0]
valid_input_p_hat = p_hat_S[1]
test_input_p_hat = p_hat_S[2]
test_data_p_hat = p_hat_S[3]
```

```
## Function to calculate the average sum rate
# Here, p_model is the output of DNN, and it is a 2D array.
import math

```

```
def average_sum_rate(hij, p_model, sigma_sqr_noise, K):
  R = 0
  hij_size = hij.shape[0]
  hij_abs_sqr = cmplx_abs_sqr(hij)

  for k in range(hij_size):
    for i in range(K):  # Total rows
      phn = 0
      for j in range(K):  # Total columns
        phn_j = np.multiply(p_model[k,j], hij_abs_sqr[k,i,j])
        phn = phn + phn_j

      numr_s = np.multiply(p_model[k,i], hij_abs_sqr[k,i,i])
      dnumr_s = sigma_sqr_noise[i] + phn - numr_s
      R_temp = math.log2(1 + np.divide(numr_s, dnumr_s))
      R = R + R_temp

  return (R/hij_size)
```

```
# DNN Sum Rate for test_data_F_H
output_P_hat = abs(test_data_p_hat)
sumrate_F_H = average_sum_rate(test_data_F_H, output_P_hat,
    sigma_sqr_noise, K)
print("Average Sum Rate for all H matrices: {:.3f} Bit/Second/Hertz".
    format(sumrate_F_H))
```

C.2 Codes for analyzing the PCNet model

C.2.1 For training with a given background noise power

```
import numpy as np

## Number of transmitter-receiver pairs
K = 5

## Variances for noise signals
sigma_sqr_noise = np.array([1e-0, 1e-0, 1e-0, 1e-0, 1e-0], dtype =
    float)

## Minimum rate for the achievable SINR of multiple concurrent
## transmissions
SINR_P_min = np.array([0.5, 0.5, 0.5, 0.5, 0.5], dtype = float)

## Maximum transmit power
p_max = 1.0
```

```
## Loading a CSV file (F_H_2D.csv) for feasible H matrices that was
## uploaded to Google Collab's session storage.
from numpy import loadtxt

## Reading an array from the file
F_H_2D_L = np.loadtxt('F_H_2D.csv', delimiter = ',', dtype = str)

## Reshaping the array from 2D to 3D
F_H_3D = F_H_2D_L.reshape(F_H_2D_L.shape[0], F_H_2D_L.shape[1] // K,
    K)
F_H_3D_size = F_H_3D.shape[0]
```

```
## Converting string data to complex data and removing the initial
## whitespace
F_H_list = []
for k in range(F_H_3D_size):
  for i in range(K):  # Total rows
    for j in range(K):  # Total columns
```

```
      F_H_temp = complex(F_H_3D[k][i][j].strip())
      F_H_list.append(F_H_temp)
F_H_array = np.array(F_H_list)
F_H = F_H_array.reshape((F_H_3D_size, K, K)) # H_size X row X
    column_count
print(F_H.shape)
F_H_size = F_H.shape[0]
# print(F_H)
```

```
import numba as nb

## Function to compute the square of the absolute value of an array
## of complex numbers
@nb.vectorize([nb.float64(nb.complex128),nb.float32(nb.complex64)])
def cmplx_abs_sqr(cmplx_var):
  return cmplx_var.real**2 + cmplx_var.imag**2
```

```
## Function to generate the matrix A (K x K)
def generate_A(F_H_size, K, SINR_P_min, F_H):
  Aij_list = []
  F_H_abs_sqr = cmplx_abs_sqr(F_H)

  for k in range(F_H_size):
    for i in range(K):  # Total rows
      Aj_list =[]
      for j in range(K): # Total columns
        if i==j:
          A = F_H_abs_sqr[k,i,j]
        else:
          A = np.multiply(-SINR_P_min[i], F_H_abs_sqr[k,i,j])
        Aj_list.append(A)
      Aij_list.append(Aj_list)
  Aij_array = np.array(Aij_list)
  Aij = Aij_array.reshape((F_H_size, K, K)) # H_size X row X column
  return Aij
```

```
## Create matrix A
A = generate_A(F_H_size, K, SINR_P_min, F_H)
print(A.shape)
# print(A)
```

```
## Function to generate the vector b (K x 1)
def generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H):
  bi_list = []
  for k in range(F_H_size):
    for i in range(K):  # Total rows, i.e., total transmitters
      b = np.multiply(SINR_P_min[i], sigma_sqr_noise[i])
      bi_list.append(b)
  bi_array = np.array(bi_list)
  bi = bi_array.reshape((F_H_size, K, 1)) # H_size X row X column
  return bi
```

```
## Create vector b
b = generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H)
print(b.shape)
# print(b)
```

```
## Create matrix A_inv, i.e., the pseudo inverse of matrix A
A_inv = np.linalg.pinv(A)
print(A_inv.shape)
# print(A_inv)
```

```
## Create a vector p_hat = (A_inv x b)
p_hat = np.matmul(A_inv, b)
print(p_hat.shape)
# print(p_hat)
```

```
## Function to split datasets for training, validation, and testing.

def split(np_array):
  # data_size = np_array.shape[0]
  # train_data_size = int(data_size * 0.8)
  # valid_data_size = int(data_size * 0.1)
  # test_data_size = int(data_size * 0.1)

  train_data_size = int(200000)
  valid_data_size = int(25000)
  test_data_size = int(25000)

  train_e_indx = train_data_size
  valid_e_indx = train_e_indx + valid_data_size
  # test_e_indx = valid_e_indx + test_data_size - 2
```

```
  test_e_indx = valid_e_indx + test_data_size
  test_data_size_n = test_e_indx - valid_e_indx

  row_count = np_array.shape[1]
  column_count = np_array.shape[2]

  train_data = np.empty((train_data_size, row_count, column_count),
   dtype = complex, order = 'C')
  valid_data = np.empty((valid_data_size, row_count, column_count),
   dtype = complex, order = 'C')
  test_data = np.empty((test_data_size_n, row_count, column_count),
   dtype = complex, order = 'C')

  for i in range(train_e_indx):
    train_data[i] = np_array[i]

  xv = 0
  for j in range(train_e_indx, valid_e_indx):
    valid_data[xv] = np_array[j]
    xv = xv + 1

  xt = 0
  for k in range(valid_e_indx, test_e_indx):
    test_data[xt] = np_array[k]
    xt = xt + 1

  # print(train_data.shape, valid_data.shape, test_data.shape)

  ## Training input will be the absolute value
  train_input = np.absolute(train_data)
  valid_input = np.absolute(valid_data)
  test_input = np.absolute(test_data)

  print(train_input.shape, valid_input.shape, test_input.shape)

  return [train_input, valid_input, test_input, test_data]
```

```
## Split F_H Matrix
F_H_S = split(F_H)
```

```
train_input_F_H = F_H_S[0]
valid_input_F_H = F_H_S[1]
test_input_F_H = F_H_S[2]
test_data_F_H = F_H_S[3]
```

```
## Split p_hat Matrix
p_hat_S = split(p_hat)
train_input_p_hat = p_hat_S[0]
valid_input_p_hat = p_hat_S[1]
test_input_p_hat = p_hat_S[2]
test_data_p_hat = p_hat_S[3]
```

```
## Define the DNN model - The Sequential model
import tensorflow as tf
from tensorflow import keras
## from tensorflow.keras import layers # shows warning
from keras.api._v2.keras import layers

model = keras.Sequential(name = "sequential_model")

model.add(keras.Input(shape = (K,K), name = "hij_inputs"))
model.add(layers.Flatten(name = "flatten_layer_hij"))

model.add(layers.Dense(units = 2*K*K, activation = 'relu',
    input_shape = (K*K,), name = "dense_layer_1"))
model.add(layers.BatchNormalization())

model.add(layers.Dense(units = K*K, activation = 'relu', input_shape
    = (2*K*K,), name = "dense_layer_2"))
model.add(layers.BatchNormalization())

model.add(layers.Dense(units = K, activation = 'sigmoid', input_shape
     = (K*K,), name = "P_hat"))

model.summary()
```

```
## Plot the model as a graph
keras.utils.plot_model(model, "Sequential_Model.png")
```

```
## Display the input and output shapes of each layer
```

```
keras.utils.plot_model(model, "Sequential_Model_with_shape_info.png",
    show_shapes=True)
```

```
## Convert sigma_sqr_noise from numpy array to tensor
sigma_sqr_noise_t = tf.convert_to_tensor(sigma_sqr_noise, dtype =
   float)
tf.print(sigma_sqr_noise_t)
```

```
## Convert SINR_P_min from numpy array to tensor
SINR_P_min_t = tf.convert_to_tensor(SINR_P_min, dtype = float)
tf.print(SINR_P_min_t)
```

```
## The customized loss function that penalizes the constraint
   violation
def custom_loss(y_true, y_pred):
  p = tf.math.multiply(p_max, y_pred)
  hij = tf.reshape(y_true[:,0:K*K], (-1,K,K))
  hij_abs_sqr = tf.math.square(tf.math.abs(hij))

  lambda_l = 0.0
  R_P = 0.0
  pnlty_f_CV = 0.0

  for i in range(K):  # Total rows
    ph = 0.0
    for j in range(K):  # Total columns
      ph_j = tf.math.multiply(p[:,j], hij_abs_sqr[:,i,j])
      ph = tf.math.add(ph, ph_j)

    numr = tf.math.multiply(p[:,i], hij_abs_sqr[:,i,i])
    dnumr = tf.math.add(sigma_sqr_noise_t[i], tf.math.subtract(ph,
   numr))
    SINR_i = tf.math.divide(numr, dnumr)
    R_P = tf.math.add(R_P, (tf.math.log(1 + SINR_i)/tf.math.log(2.0))
   )
    pnlty_f_CV = tf.math.add(pnlty_f_CV,
                             tf.nn.relu((tf.math.log(1 + SINR_P_min_t
   [i])/tf.math.log(2.0))
                                        - (tf.math.log(1 + SINR_i)/tf.
   math.log(2.0))))
```

```
24
25   loss = tf.math.add(-R_P, tf.math.multiply(lambda_l, pnlty_f_CV))
26   loss = tf.reduce_mean(loss) # batch mean
27   return loss
```

```
1 ## Build and compile the DNN model
2 ## Training and Testing
3 import matplotlib.pyplot as plt
4
5 optA = tf.keras.optimizers.Adam(learning_rate = 0.0001)
6 model.compile(optimizer = optA, loss = custom_loss)
7
8 history = model.fit(train_input_F_H, train_input_F_H, epochs = 50,
      validation_data = (valid_input_F_H, valid_input_F_H), batch_size
      = 1000)
9
10 plt.plot(history.epoch, history.history['loss'], color = "blue",
      label = "Training")
11 plt.plot(history.epoch, history.history['val_loss'], color="black",
      label = "Validation")
12 plt.xlabel("epochs")
13 plt.ylabel("loss")
14 plt.legend()
15 plt.show()
```

```
1 ## Constraint violation probability and
2 ## finding indexes of test_input_F_H matrix with the hij set that do
3 ## not satisfy constraint on the minimum SINR_P_min rate but satisfy
4 ## the maximum transmit power p_max
5
6 output_P_hat_temp = p_max * model.predict(test_input_F_H)
7 output_P_hat = output_P_hat_temp.reshape((output_P_hat_temp.shape[0],
       output_P_hat_temp.shape[1], 1)) # test_input_F_H_size X row X
      column
8 output_P_hat_size = output_P_hat.shape[0]
9 test_data_F_H_abs_sqr = cmplx_abs_sqr(test_data_F_H)
10
11 indx_n = []
12 count_v = 0
13
```

```
for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(output_P_hat[k,j], test_data_F_H_abs_sqr[k,i,j])
      ph = ph + ph_j

    numr = np.multiply(output_P_hat[k,i], test_data_F_H_abs_sqr[k,i,i])
    dnumr = sigma_sqr_noise[i] + ph - numr
    SINR_out = np.divide(numr, dnumr)
    if np.round(SINR_out, decimals = 3) < SINR_P_min[i]:
      indx_n.append(k)
      count_v = count_v + 1
      # print(SINR_out)
      break

violation_prb = (count_v / output_P_hat_size) * 100
print("Constraints Violation Probability: {:.2f}%".format(violation_prb))
# print(len(indx_n))
# print(indx_n)
```

```
## Function to calculate the average sum rate
# Here, p_model is the output of DNN, and it is a 2D array.
import math

def average_sum_rate(hij, p_model, sigma_sqr_noise, K):
  R = 0
  hij_size = hij.shape[0]
  hij_abs_sqr = cmplx_abs_sqr(hij)

  for k in range(hij_size):
    for i in range(K):  # Total rows
      phn = 0
      for j in range(K):  # Total columns
        phn_j = np.multiply(p_model[k,j], hij_abs_sqr[k,i,j])
        phn = phn + phn_j
```

```
16
17        numr_s = np.multiply(p_model[k,i], hij_abs_sqr[k,i,i])
18        dnumr_s = sigma_sqr_noise[i] + phn - numr_s
19        R_temp = math.log2(1 + np.divide(numr_s, dnumr_s))
20        R = R + R_temp
21
22    return (R/hij_size)
```

```
1 # Calculating the curated power vector p_tilda
2 # p_tilda = test_input_p_hat when SINR_P_min is not met
3 # p_tilda = output_P_hat when SINR_P_min is met
4
5 p_tilda = np.empty((output_P_hat_size, K, 1), dtype = float, order =
      'C')
6
7 i = 0
8 for j in range(output_P_hat_size):
9   if (i < len(indx_n)) and (j == indx_n[i]):
10    p_tilda[j] = (test_input_p_hat[j] * p_max) / np.amax(
      test_input_p_hat[j])
11    i = i + 1
12  else:
13    p_tilda[j] = output_P_hat[j]
14
15 print(p_tilda.shape)
16 # print(p_tilda)
```

```
1 ## Checking p_tilda, i.e., the power for test_data_F_H for negative
2 ## values and Hit Rate i.e. percentage for 0 <= p_tilda <= p_max
3 count_p_t = 0
4 count_n_t = 0
5
6 for n in range(output_P_hat_size):
7   P_max = np.amax(p_tilda[n])
8   if np.round(P_max, decimals = 3) <= 1:
9     count_p_t = count_p_t + 1
10
11  if np.any(p_tilda[n] < 0):
12    count_n_t = count_n_t + 1
13    print(n,'\n')
```

```
    print(p_tilda)

p_tilda_hit_rate = (count_p_t / output_P_hat_size) * 100
print("Hit Rate for Power p_tilda: {:.2f}%".format(p_tilda_hit_rate))
print("Negative power count: ", count_n_t)
```

```
## Constraint violation probability for p_tilda on the SINR_P_min
# indx_t = []
count_v_t = 0

for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(p_tilda[k,j], test_data_F_H_abs_sqr[k,i,j])
      ph = ph + ph_j

    numr = np.multiply(p_tilda[k,i], test_data_F_H_abs_sqr[k,i,i])
    dnumr = sigma_sqr_noise[i] + ph - numr
    SINR_out_t = np.divide(numr, dnumr)
    # if k == 24463:
    #   print(SINR_out_t)

    if np.round(SINR_out_t, decimals = 2) < SINR_P_min[i]:
      # indx_t.append(k)
      count_v_t = count_v_t + 1
      break

violation_prb_t = (count_v_t / output_P_hat_size) * 100
print("SINR_P_min Constraints Violation Probability for p_tilda: {:.2
   f}%".format(violation_prb_t))
```

```
## DNN Sum Rate for test_data_F_H
sumrate_s_F_H = average_sum_rate(test_data_F_H, p_tilda,
   sigma_sqr_noise, K)
print("Total Average Sum Rate for all H matrices: {:.3f} Bit/Second/
   Hertz".format(sumrate_s_F_H))
```

C.2.2 Codes for analyzing the PCNet+ model: For enhanced generalization capacity

```
import numpy as np

## Number of transmitter-receiver pairs
K = 8

## Minimum rate for the achievable SINR of multiple concurrent transmissions
SINR_P_min = np.array([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2])

## Maximum transmit power
p_max = 1.0
```

```
## Loading a NumPy array from a CSV file
# Loading F_H array from a CSV file
from numpy import loadtxt

## Reading an array from the file
# If we want to read a file from our local drive, we have to first upload it to Collab's session storage.
F_H_2D_L_0dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_0dB.csv', delimiter = ',', dtype = str)
F_H_2D_L_10dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_10dB.csv', delimiter = ',', dtype = str)
F_H_2D_L_20dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_20dB.csv', delimiter = ',', dtype = str)
F_H_2D_L_30dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_30dB.csv', delimiter = ',', dtype = str)
F_H_2D_L_40dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_40dB.csv', delimiter = ',', dtype = str)

# ## Reshaping the array from 2D to 3D
F_H_3D_0dB = F_H_2D_L_0dB.reshape(F_H_2D_L_0dB.shape[0], F_H_2D_L_0dB.shape[1] // K, K)
F_H_3D_10dB = F_H_2D_L_10dB.reshape(F_H_2D_L_10dB.shape[0], F_H_2D_L_10dB.shape[1] // K, K)
F_H_3D_20dB = F_H_2D_L_20dB.reshape(F_H_2D_L_20dB.shape[0],
```

```
    F_H_2D_L_20dB.shape[1] // K, K)
F_H_3D_30dB = F_H_2D_L_30dB.reshape(F_H_2D_L_30dB.shape[0],
    F_H_2D_L_30dB.shape[1] // K, K)
F_H_3D_40dB = F_H_2D_L_40dB.reshape(F_H_2D_L_40dB.shape[0],
    F_H_2D_L_40dB.shape[1] // K, K)

F_H_3D_0dB_size = F_H_3D_0dB.shape[0]
F_H_3D_10dB_size = F_H_3D_10dB.shape[0]
F_H_3D_20dB_size = F_H_3D_20dB.shape[0]
F_H_3D_30dB_size = F_H_3D_30dB.shape[0]
F_H_3D_40dB_size = F_H_3D_40dB.shape[0]
```

```
## Function to convert string data to complex data and to remove the
    initial whitespace
def cnvrt_2_cmplx_data(F_H_3D_size, F_H_3D):
  F_H_list = []
  for k in range(F_H_3D_size):
    for i in range(K):  # Total rows
      for j in range(K):  # Total columns
        F_H_temp = complex(F_H_3D[k][i][j].strip())
        F_H_list.append(F_H_temp)
  F_H_array = np.array(F_H_list)
  F_H = F_H_array.reshape((F_H_3D_size, K, K)) # H_size X row X
    column_count
  return F_H
```

```
## Converting string data to complex data and removing the initial
    whitespace
F_H_0dB = cnvrt_2_cmplx_data(F_H_3D_0dB_size, F_H_3D_0dB)
F_H_10dB = cnvrt_2_cmplx_data(F_H_3D_10dB_size, F_H_3D_10dB)
F_H_20dB = cnvrt_2_cmplx_data(F_H_3D_20dB_size, F_H_3D_20dB)
F_H_30dB = cnvrt_2_cmplx_data(F_H_3D_30dB_size, F_H_3D_30dB)
F_H_40dB = cnvrt_2_cmplx_data(F_H_3D_40dB_size, F_H_3D_40dB)

print(F_H_0dB.shape)
print(F_H_10dB.shape)
print(F_H_20dB.shape)
print(F_H_30dB.shape)
print(F_H_40dB.shape)

```

```
F_H_0dB_size = F_H_0dB.shape[0]
F_H_10dB_size = F_H_10dB.shape[0]
F_H_20dB_size = F_H_20dB.shape[0]
F_H_30dB_size = F_H_30dB.shape[0]
F_H_40dB_size = F_H_40dB.shape[0]

# print(F_H_0dB)
# print(F_H_10dB)
# print(F_H_20dB)
# print(F_H_30dB)
# print(F_H_40dB)
```

```
import numba as nb

## Function to compute the square of the absolute value of an array
    of complex numbers
@nb.vectorize([nb.float64(nb.complex128),nb.float32(nb.complex64)])
def cmplx_abs_sqr(cmplx_var):
  return cmplx_var.real**2 + cmplx_var.imag**2
```

```
## Function to generate the matrix A (K x K)
def generate_A(F_H_size, K, SINR_P_min, F_H):
  Aij_list = []
  F_H_abs_sqr = cmplx_abs_sqr(F_H)

  for k in range(F_H_size):
    for i in range(K):  # Total rows
      Aj_list =[]
      for j in range(K): # Total columns
        if i==j:
          A = F_H_abs_sqr[k,i,j]
        else:
          A = np.multiply(-SINR_P_min[i], F_H_abs_sqr[k,i,j])
        Aj_list.append(A)
      Aij_list.append(Aj_list)
  Aij_array = np.array(Aij_list)
  Aij = Aij_array.reshape((F_H_size, K, K)) # H_size X row X column
  return Aij
```

```
## Create matrix A
```

```
A_0dB = generate_A(F_H_0dB_size, K, SINR_P_min, F_H_0dB)
A_10dB = generate_A(F_H_10dB_size, K, SINR_P_min, F_H_10dB)
A_20dB = generate_A(F_H_20dB_size, K, SINR_P_min, F_H_20dB)
A_30dB = generate_A(F_H_30dB_size, K, SINR_P_min, F_H_30dB)
A_40dB = generate_A(F_H_40dB_size, K, SINR_P_min, F_H_40dB)

print(A_0dB.shape)
print(A_10dB.shape)
print(A_20dB.shape)
print(A_30dB.shape)
print(A_40dB.shape)

# print(A_0dB)
# print(A_10dB)
# print(A_20dB)
# print(A_30dB)
# print(A_40dB)
```

```
## Variances for noise signals
sigma_sqr_noise_0dB = np.array([1e-0, 1e-0, 1e-0, 1e-0, 1e-0, 1e-0, 1e-0, 1e-0], dtype = float)
sigma_sqr_noise_10dB = np.array([1e-1, 1e-1, 1e-1, 1e-1, 1e-1, 1e-1, 1e-1, 1e-1], dtype = float)
sigma_sqr_noise_20dB = np.array([1e-2, 1e-2, 1e-2, 1e-2, 1e-2, 1e-2, 1e-2, 1e-2], dtype = float)
sigma_sqr_noise_30dB = np.array([1e-3, 1e-3, 1e-3, 1e-3, 1e-3, 1e-3, 1e-3, 1e-3], dtype = float)
sigma_sqr_noise_40dB = np.array([1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4], dtype = float)
```

```
## Function to generate the vector b (K x 1)
def generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H):
  bi_list = []
  for k in range(F_H_size):
    for i in range(K):  # Total rows, i.e., total transmitters
      b = np.multiply(SINR_P_min[i], sigma_sqr_noise[i])
      bi_list.append(b)
  bi_array = np.array(bi_list)
  bi = bi_array.reshape((F_H_size, K, 1)) # H_size X row X column
  return bi
```

```
## Create vector b
b_0dB = generate_b(F_H_0dB_size, K, SINR_P_min, sigma_sqr_noise_0dB,
    F_H_0dB)
b_10dB = generate_b(F_H_10dB_size, K, SINR_P_min,
    sigma_sqr_noise_10dB, F_H_10dB)
b_20dB = generate_b(F_H_20dB_size, K, SINR_P_min,
    sigma_sqr_noise_20dB, F_H_20dB)
b_30dB = generate_b(F_H_30dB_size, K, SINR_P_min,
    sigma_sqr_noise_30dB, F_H_30dB)
b_40dB = generate_b(F_H_40dB_size, K, SINR_P_min,
    sigma_sqr_noise_40dB, F_H_40dB)

print(b_0dB.shape)
print(b_10dB.shape)
print(b_20dB.shape)
print(b_30dB.shape)
print(b_40dB.shape)

# print(b_0dB)
# print(b_10dB)
# print(b_20dB)
# print(b_30dB)
# print(b_40dB)
```

```
## Create matrix A_inv, i.e., the pseudo inverse of matrix A
A_inv_0dB = np.linalg.pinv(A_0dB)
A_inv_10dB = np.linalg.pinv(A_10dB)
A_inv_20dB = np.linalg.pinv(A_20dB)
A_inv_30dB = np.linalg.pinv(A_30dB)
A_inv_40dB = np.linalg.pinv(A_40dB)

A_inv_0dB[A_inv_0dB<0] = 0
A_inv_10dB[A_inv_10dB<0] = 0
A_inv_20dB[A_inv_20dB<0] = 0
A_inv_30dB[A_inv_30dB<0] = 0
A_inv_40dB[A_inv_40dB<0] = 0

print(A_inv_0dB.shape)
print(A_inv_10dB.shape)
```

```
print(A_inv_20dB.shape)
print(A_inv_30dB.shape)
print(A_inv_40dB.shape)

# print(A_inv_0dB)
# print(A_inv_10dB)
# print(A_inv_20dB)
# print(A_inv_30dB)
# print(A_inv_40dB)
```

```
## Create a vector p_hat = (A_inv x b)
p_hat_0dB = np.matmul(A_inv_0dB, b_0dB)
p_hat_10dB = np.matmul(A_inv_10dB, b_10dB)
p_hat_20dB = np.matmul(A_inv_20dB, b_20dB)
p_hat_30dB = np.matmul(A_inv_30dB, b_30dB)
p_hat_40dB = np.matmul(A_inv_40dB, b_40dB)

print(p_hat_0dB.shape)
print(p_hat_10dB.shape)
print(p_hat_20dB.shape)
print(p_hat_30dB.shape)
print(p_hat_40dB.shape)

# print(p_hat_0dB)
# print(p_hat_10dB)
# print(p_hat_20dB)
# print(p_hat_30dB)
# print(p_hat_40dB)
```

```
## Function to split datasets for training, validation, and testing.
def split(np_array):
  # data_size = np_array.shape[0]
  # train_data_size = int(data_size * 0.8)
  # valid_data_size = int(data_size * 0.1)
  # test_data_size = int(data_size * 0.1)

  train_data_size = int(200000)
  valid_data_size = int(25000)
  test_data_size = int(25000)

```

```
12  train_e_indx = train_data_size
13  valid_e_indx = train_e_indx + valid_data_size
14  test_e_indx = valid_e_indx + test_data_size
15  test_data_size_n = test_e_indx - valid_e_indx
16
17  row_count = np_array.shape[1]
18  column_count = np_array.shape[2]
19
20  train_data = np.empty((train_data_size, row_count, column_count),
     dtype = complex, order = 'C')
21  valid_data = np.empty((valid_data_size, row_count, column_count),
     dtype = complex, order = 'C')
22  test_data = np.empty((test_data_size_n, row_count, column_count),
     dtype = complex, order = 'C')
23
24  for i in range(train_e_indx):
25    train_data[i] = np_array[i]
26
27  xv = 0
28  for j in range(train_e_indx, valid_e_indx):
29    valid_data[xv] = np_array[j]
30    xv = xv + 1
31
32  xt = 0
33  for k in range(valid_e_indx, test_e_indx):
34    test_data[xt] = np_array[k]
35    xt = xt + 1
36
37  # print(train_data.shape, valid_data.shape, test_data.shape)
38
39
40  ## Training input will be the absolute value
41  train_input = np.absolute(train_data)
42  valid_input = np.absolute(valid_data)
43  test_input = np.absolute(test_data)
44
45  print(train_input.shape, valid_input.shape, test_input.shape)
46
47  return [train_input, valid_input, test_input, test_data]
```

```
## Split F_H matrix
F_H_S_0dB = split(F_H_0dB)
train_input_F_H_0dB = F_H_S_0dB[0]
valid_input_F_H_0dB = F_H_S_0dB[1]
test_input_F_H_0dB = F_H_S_0dB[2]
test_data_F_H_0dB = F_H_S_0dB[3]

F_H_S_10dB = split(F_H_10dB)
train_input_F_H_10dB = F_H_S_10dB[0]
valid_input_F_H_10dB = F_H_S_10dB[1]
test_input_F_H_10dB = F_H_S_10dB[2]
test_data_F_H_10dB = F_H_S_10dB[3]

F_H_S_20dB = split(F_H_20dB)
train_input_F_H_20dB = F_H_S_20dB[0]
valid_input_F_H_20dB = F_H_S_20dB[1]
test_input_F_H_20dB = F_H_S_20dB[2]
test_data_F_H_20dB = F_H_S_20dB[3]

F_H_S_30dB = split(F_H_30dB)
train_input_F_H_30dB = F_H_S_30dB[0]
valid_input_F_H_30dB = F_H_S_30dB[1]
test_input_F_H_30dB = F_H_S_30dB[2]
test_data_F_H_30dB = F_H_S_30dB[3]

F_H_S_40dB = split(F_H_40dB)
train_input_F_H_40dB = F_H_S_40dB[0]
valid_input_F_H_40dB = F_H_S_40dB[1]
test_input_F_H_40dB = F_H_S_40dB[2]
test_data_F_H_40dB = F_H_S_40dB[3]
```

```
## Split p_hat vector
p_hat_S_0dB = split(p_hat_0dB)
train_input_p_hat_0dB = p_hat_S_0dB[0]
valid_input_p_hat_0dB = p_hat_S_0dB[1]
test_input_p_hat_0dB = p_hat_S_0dB[2]
test_data_p_hat_0dB = p_hat_S_0dB[3]

p_hat_S_10dB = split(p_hat_10dB)
```

```
train_input_p_hat_10dB = p_hat_S_10dB[0]
valid_input_p_hat_10dB = p_hat_S_10dB[1]
test_input_p_hat_10dB = p_hat_S_10dB[2]
test_data_p_hat_10dB = p_hat_S_10dB[3]

p_hat_S_20dB = split(p_hat_20dB)
train_input_p_hat_20dB = p_hat_S_20dB[0]
valid_input_p_hat_20dB = p_hat_S_20dB[1]
test_input_p_hat_20dB = p_hat_S_20dB[2]
test_data_p_hat_20dB = p_hat_S_20dB[3]

p_hat_S_30dB = split(p_hat_30dB)
train_input_p_hat_30dB = p_hat_S_30dB[0]
valid_input_p_hat_30dB = p_hat_S_30dB[1]
test_input_p_hat_30dB = p_hat_S_30dB[2]
test_data_p_hat_30dB = p_hat_S_30dB[3]

p_hat_S_40dB = split(p_hat_40dB)
train_input_p_hat_40dB = p_hat_S_40dB[0]
valid_input_p_hat_40dB = p_hat_S_40dB[1]
test_input_p_hat_40dB = p_hat_S_40dB[2]
test_data_p_hat_40dB = p_hat_S_40dB[3]
```

```
## Create EsN0 vector
EsN0_array_0dB = np.full(shape = F_H_0dB_size, fill_value = 0, dtype
    = int)
EsN0_array_10dB = np.full(shape = F_H_10dB_size, fill_value = 10,
    dtype = int)
EsN0_array_20dB = np.full(shape = F_H_20dB_size, fill_value = 20,
    dtype = int)
EsN0_array_30dB = np.full(shape = F_H_30dB_size, fill_value = 30,
    dtype = int)
EsN0_array_40dB = np.full(shape = F_H_40dB_size, fill_value = 40,
    dtype = int)

EsN0_vector_0dB = EsN0_array_0dB.reshape((F_H_0dB_size, 1)) # row X
    column
EsN0_vector_10dB = EsN0_array_10dB.reshape((F_H_10dB_size, 1)) # row
    X column
```

```
EsN0_vector_20dB = EsN0_array_20dB.reshape((F_H_20dB_size, 1)) # row X column
EsN0_vector_30dB = EsN0_array_30dB.reshape((F_H_30dB_size, 1)) # row X column
EsN0_vector_40dB = EsN0_array_40dB.reshape((F_H_40dB_size, 1)) # row X column

print(EsN0_vector_0dB.shape)
print(EsN0_vector_10dB.shape)
print(EsN0_vector_20dB.shape)
print(EsN0_vector_30dB.shape)
print(EsN0_vector_40dB.shape)
```

```
## Function to split EsN0 vector for training, validation, and testing.
def split_EsN0(np_vector):
  # data_size = np_vector.shape[0]
  # train_data_size = int(data_size * 0.8)
  # valid_data_size = int(data_size * 0.1)
  # test_data_size = int(data_size * 0.1)

  train_data_size = int(200000)
  valid_data_size = int(25000)
  test_data_size = int(25000)

  train_e_indx = train_data_size
  valid_e_indx = train_e_indx + valid_data_size
  test_e_indx = valid_e_indx + test_data_size
  test_data_size_n = test_e_indx - valid_e_indx

  row_count = np_vector.shape[1]
  column_count = 1

  train_data = np.empty((train_data_size, row_count, column_count), dtype = int, order = 'C')
  valid_data = np.empty((valid_data_size, row_count, column_count), dtype = int, order = 'C')
  test_data = np.empty((test_data_size_n, row_count, column_count), dtype = int, order = 'C')
```

```
23
24    for i in range(train_e_indx):
25      train_data[i] = np_vector[i]
26
27    xv = 0
28    for j in range(train_e_indx, valid_e_indx):
29      valid_data[xv] = np_vector[j]
30      xv = xv + 1
31
32    xt = 0
33    for k in range(valid_e_indx, test_e_indx):
34      test_data[xt] = np_vector[k]
35      xt = xt + 1
36
37    # print(train_data.shape, valid_data.shape, test_data.shape)
38
39
40    ## Training input will be the absolute value
41    train_input = np.absolute(train_data)
42    valid_input = np.absolute(valid_data)
43    test_input = np.absolute(test_data)
44
45    print(train_input.shape, valid_input.shape, test_input.shape)
46
47    return [train_input, valid_input, test_input, test_data]
```

```
1 ## Split EsN0 vector
2 EsN0_S_0dB = split_EsN0(EsN0_vector_0dB)
3 train_input_EsN0_0dB = EsN0_S_0dB[0]
4 valid_input_EsN0_0dB = EsN0_S_0dB[1]
5 test_input_EsN0_0dB = EsN0_S_0dB[2]
6 test_data_EsN0_0dB = EsN0_S_0dB[3]
7
8 EsN0_S_10dB = split_EsN0(EsN0_vector_10dB)
9 train_input_EsN0_10dB = EsN0_S_10dB[0]
10 valid_input_EsN0_10dB = EsN0_S_10dB[1]
11 test_input_EsN0_10dB = EsN0_S_10dB[2]
12 test_data_EsN0_10dB = EsN0_S_10dB[3]
13
```

```
EsN0_S_20dB = split_EsN0(EsN0_vector_20dB)
train_input_EsN0_20dB = EsN0_S_20dB[0]
valid_input_EsN0_20dB = EsN0_S_20dB[1]
test_input_EsN0_20dB = EsN0_S_20dB[2]
test_data_EsN0_20dB = EsN0_S_20dB[3]

EsN0_S_30dB = split_EsN0(EsN0_vector_30dB)
train_input_EsN0_30dB = EsN0_S_30dB[0]
valid_input_EsN0_30dB = EsN0_S_30dB[1]
test_input_EsN0_30dB = EsN0_S_30dB[2]
test_data_EsN0_30dB = EsN0_S_30dB[3]

EsN0_S_40dB = split_EsN0(EsN0_vector_40dB)
train_input_EsN0_40dB = EsN0_S_40dB[0]
valid_input_EsN0_40dB = EsN0_S_40dB[1]
test_input_EsN0_40dB = EsN0_S_40dB[2]
test_data_EsN0_40dB = EsN0_S_40dB[3]
```

```
## Creating datasets for training
train_input_F_H = np.concatenate((train_input_F_H_0dB,
    train_input_F_H_10dB, train_input_F_H_20dB, train_input_F_H_30dB,
    train_input_F_H_40dB,), axis=0)

train_input_EsN0 = np.concatenate((train_input_EsN0_0dB,
    train_input_EsN0_10dB, train_input_EsN0_20dB,
    train_input_EsN0_30dB, train_input_EsN0_40dB), axis=0)

print(train_input_F_H.shape)
print(train_input_EsN0.shape)
```

```
## Creating datasets for validation
valid_input_F_H = np.concatenate((valid_input_F_H_0dB,
    valid_input_F_H_10dB, valid_input_F_H_20dB, valid_input_F_H_30dB,
    valid_input_F_H_40dB,), axis=0)

valid_input_EsN0 = np.concatenate((valid_input_EsN0_0dB,
    valid_input_EsN0_10dB, valid_input_EsN0_20dB,
    valid_input_EsN0_30dB, valid_input_EsN0_40dB), axis=0)

print(valid_input_F_H.shape)
```

```
print(valid_input_EsN0.shape)
```

```
## Shuffling the training datasets
train_shuffler = np.random.permutation(len(train_input_F_H))
train_input_F_H_shuffled = train_input_F_H[train_shuffler]
train_input_EsN0_shuffled = train_input_EsN0[train_shuffler]
```

```
## Shuffling the validation datasets
valid_shuffler = np.random.permutation(len(valid_input_F_H))
valid_input_F_H_shuffled = valid_input_F_H[valid_shuffler]
valid_input_EsN0_shuffled = valid_input_EsN0[valid_shuffler]
```

```
## Reshaping train_input_F_H_shuffled and adding
   train_input_EsN0_shuffled
const = K*K
len1 = train_input_F_H_shuffled.shape[0]
train_input_F_H_shuffled_reshaped = train_input_F_H_shuffled.reshape
   ((len1, 1, const)) # size X row X column
train_y_true = np.concatenate((train_input_F_H_shuffled_reshaped,
   train_input_EsN0_shuffled), axis=2)
print(train_y_true.shape)
```

```
## Reshaping train_input_F_H_shuffled and adding
   train_input_EsN0_shuffled
len2 = valid_input_F_H_shuffled.shape[0]
valid_input_F_H_shuffled_reshaped = valid_input_F_H_shuffled.reshape
   ((len2, 1, const)) # size X row X column
valid_y_true = np.concatenate((valid_input_F_H_shuffled_reshaped,
   valid_input_EsN0_shuffled), axis=2)
print(valid_y_true.shape)
```

```
## Define the DNN model - The Functional API
import tensorflow as tf
from tensorflow import keras
## from tensorflow.keras import layers # shows warning
from keras.api._v2.keras import layers
from keras.layers import Input, concatenate
from keras.models import Model

hij_inputs = keras.Input(shape=(K,K), name = "hij_inputs")
f1 = layers.Flatten(name = "flatten_layer_hij")(hij_inputs)
```

```
11
12 EsN0_inputs = keras.Input(shape=(1,1), name = "EsN0_inputs")
13 f2 = layers.Flatten(name = "flatten_layer_EsN0")(EsN0_inputs)
14
15 concat_layers = concatenate([f1, f2])
16
17 d1 = layers.Dense(2*K*K, activation="relu", name = "dense_layer_1")(
       concat_layers)
18 b1 = layers.BatchNormalization(name = "batch_norm_layer_1")(d1)
19
20 d2 = layers.Dense(K*K, activation="relu", name = "dense_layer_2")(b1)
21 b2 = layers.BatchNormalization(name = "batch_norm_layer_2")(d2)
22
23 # meu = layers.Dense(K, activation="relu", name = "meu")(b2)
24 P_hat = layers.Dense(K, activation="sigmoid", name = "P_hat")(b2)
25
26 model = keras.Model(inputs = [hij_inputs, EsN0_inputs], outputs =
       P_hat, name = "functional_api")
27 model.summary()
```

```
1 ## Plot the model as a graph
2 keras.utils.plot_model(model, "Functional_API_Model.png")
```

```
1 ## Display the input and output shapes of each layer
2 keras.utils.plot_model(model, "Functional_API_Model_with_shape_info.
       png", show_shapes=True)
```

```
1 ## Convert SINR_P_min from numpy array to tensor
2 SINR_P_min_t = tf.convert_to_tensor(SINR_P_min, dtype = float)
3 tf.print(SINR_P_min_t)
```

```
1 ## The customized loss function that penalizes the constraint
       violation
2 def custom_loss(y_true, y_pred):
3   # p = y_pred
4   p = tf.math.multiply(p_max, y_pred)
5
6   mtrx_elmnt = K*K
7   EsN0_val = y_true[0][0][mtrx_elmnt]
8   y_true_updt = y_true[:,:,:-1]
9
```

```
  if EsN0_val < 10:
    sigma_sqr_noise_lf = 1e-0
  elif EsN0_val >= 10 and EsN0_val < 20:
    sigma_sqr_noise_lf = 1e-1
  elif EsN0_val >= 20 and EsN0_val < 30:
    sigma_sqr_noise_lf = 1e-2
  elif EsN0_val >= 30 and EsN0_val < 40:
    sigma_sqr_noise_lf = 1e-3
  else:
    sigma_sqr_noise_lf = 1e-4

  hij = tf.reshape(y_true_updt[:,0:K*K], (-1,K,K))
  hij_abs_sqr = tf.math.square(tf.math.abs(hij))

  lambda_l = 5.0
  R_P = 0.0
  pnlty_f_CV = 0.0

  for i in range(K):  # Total rows
    ph = 0.0
    for j in range(K):  # Total columns
      ph_j = tf.math.multiply(p[:,j], hij_abs_sqr[:,i,j])
      ph = tf.math.add(ph, ph_j)

    numr = tf.math.multiply(p[:,i], hij_abs_sqr[:,i,i])
    dnumr = tf.math.add(sigma_sqr_noise_lf, tf.math.subtract(ph, numr
  ))
    SINR_i = tf.math.divide(numr, dnumr)
    R_P = tf.math.add(R_P, (tf.math.log(1 + SINR_i)/tf.math.log(2.0))
  )
    pnlty_f_CV = tf.math.add(pnlty_f_CV,
                             tf.nn.relu((tf.math.log(1 + SINR_P_min_t
  [i])/tf.math.log(2.0))
                                        - (tf.math.log(1 + SINR_i)/tf.
  math.log(2.0))))

  loss = tf.math.add(-R_P, tf.math.multiply(lambda_l, pnlty_f_CV))
  loss = tf.reduce_mean(loss) # batch mean
  return loss
```

```
## Build and compile the DNN model
## Training and Testing
import matplotlib.pyplot as plt

optA = tf.keras.optimizers.Adam(learning_rate = 0.0001)
model.compile(optimizer = optA, loss = custom_loss)

train_input = [train_input_F_H_shuffled, train_input_EsN0_shuffled]
valid_input = [valid_input_F_H_shuffled, valid_input_EsN0_shuffled]

history = model.fit(train_input, train_y_true, epochs = 50,
                    validation_data = (valid_input, valid_y_true),
    batch_size = 1000)

plt.plot(history.epoch, history.history['loss'], color = "blue",
    label = "Training")
plt.plot(history.epoch, history.history['val_loss'], color="black",
    label = "Validation")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
plt.show()
```

```
## Constraint violation probability and
## finding indexes of test_input_F_H matrix with the hij set that do
    not satisfy
## constraint on the minimum SINR_P_min rate but satisfy the maximum
    transmit
## power p_max

test_input = [test_input_F_H_0dB, test_input_EsN0_0dB]
# output_P_hat_temp = model.predict(test_input)
output_P_hat_temp = np.multiply(p_max, model.predict(test_input))
output_P_hat = output_P_hat_temp.reshape((output_P_hat_temp.shape[0],
     output_P_hat_temp.shape[1], 1)) # test_input_F_H_size X row X
    column
output_P_hat_size = output_P_hat.shape[0]
test_data_F_H_abs_sqr = cmplx_abs_sqr(test_data_F_H_0dB)

```

```
indx_n = []
count_v = 0

for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(output_P_hat[k,j], test_data_F_H_abs_sqr[k,i,j])
      ph = ph + ph_j

    numr = np.multiply(output_P_hat[k,i], test_data_F_H_abs_sqr[k,i,i])
    dnumr = sigma_sqr_noise_0dB[i] + ph - numr
    SINR_out = np.divide(numr, dnumr)

    if np.round(SINR_out, decimals= 3) < SINR_P_min[i]:
      indx_n.append(k)
      count_v = count_v + 1
      # print(SINR_out)
      break

violation_prb = (count_v / output_P_hat_size) * 100
print("Constraints Violation Probability: {:.2f}%".format(violation_prb))
# print(len(indx_n))
# print(indx_n)
```

```
## Function to calculate the average sum rate
# Here, p_model is the output of DNN, and it is a 2D array.
import math

def average_sum_rate(hij, p_model, sigma_sqr_noise, K):
  R = 0
  hij_size = hij.shape[0]
  hij_abs_sqr = cmplx_abs_sqr(hij)

  for k in range(hij_size):
    for i in range(K):  # Total rows
```

```
      phn = 0
      for j in range(K):  # Total columns
        phn_j = np.multiply(p_model[k,j], hij_abs_sqr[k,i,j])
        phn = phn + phn_j

      numr_s = np.multiply(p_model[k,i], hij_abs_sqr[k,i,i])
      dnumr_s = sigma_sqr_noise[i] + phn - numr_s
      R_temp = math.log2(1 + np.divide(numr_s, dnumr_s))
      R = R + R_temp

  return (R/hij_size)
```

```
# Calculating the curated power vector p_tilda
# p_tilda = test_input_p_hat when SINR_P_min is not met
# p_tilda = output_P_hat when SINR_P_min is met

p_tilda = np.empty((output_P_hat_size, K, 1), dtype = float, order =
    'C')

i = 0
for j in range(output_P_hat_size):
  if (i < len(indx_n)) and (j == indx_n[i]):
    p_tilda[j] = (test_input_p_hat_0dB[j] * p_max) / np.amax(
    test_input_p_hat_0dB[j])
    i = i + 1
  else:
    p_tilda[j] = output_P_hat[j]

print(p_tilda.shape)
# print(p_tilda)
```

```
## Checking p_tilda, i.e., the power for test_data_F_H for negative
    values
## and Hit Rate i.e. percentage for 0 <= p_tilda <= p_max
count_p_t = 0
count_n_t = 0

for n in range(output_P_hat_size):
  P_max = np.amax(p_tilda[n])
  if np.round(P_max, decimals = 3) <= 1:
```

```
    count_p_t = count_p_t + 1

  if np.any(p_tilda[n] < 0):
    count_n_t = count_n_t + 1
    print(n,'\n')
    print(p_tilda)

p_tilda_hit_rate = (count_p_t / output_P_hat_size) * 100
print("Hit Rate for Power p_tilda: {:.2f}%".format(p_tilda_hit_rate))
print("Negative power count: ", count_n_t)
```

```
## Constraint violation probability for p_tilda on the SINR_P_min
# indx_t = []
count_v_t = 0

for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(p_tilda[k,j], test_data_F_H_abs_sqr[k,i,j])
      ph = ph + ph_j

    numr = np.multiply(p_tilda[k,i], test_data_F_H_abs_sqr[k,i,i])
    dnumr = sigma_sqr_noise_0dB[i] + ph - numr
    SINR_out_t = np.divide(numr, dnumr)

    if np.round(SINR_out_t, decimals = 2) < SINR_P_min[i]:
      # indx_t.append(k)
      count_v_t = count_v_t + 1
      break

violation_prb_t = (count_v_t / output_P_hat_size) * 100
print("SINR_P_min Constraints Violation Probability for p_tilda: {:.2
   f}%".format(violation_prb_t))
```

```
## DNN Sum Rate for test_data_F_H
sumrate_F_H = average_sum_rate(test_data_F_H_0dB, p_tilda,
   sigma_sqr_noise_0dB, K)
print("Average Sum Rate for all H matrices: {:.3f} Bit/Second/Hertz".
   format(sumrate_F_H))
```

C.3 Codes for analyzing the Proposed Model

C.3.1 For training with a given background noise power

```
import numpy as np

## Number of transmitter-receiver pairs
K = 5

## Variances for noise signals
sigma_sqr_noise = np.array([1e-0, 1e-0, 1e-0, 1e-0, 1e-0], dtype =
    float)

## Minimum rate for the achievable SINR of multiple concurrent
## transmissions
SINR_P_min = np.array([0.5, 0.5, 0.5, 0.5, 0.5])

## Maximum transmit power
p_max = 1.0
```

```
## Loading a NumPy array from a CSV file
# Loading F_H array from a CSV file
from numpy import loadtxt

## Reading an array from the file
# If we want to read a file from our local drive, we have to first
# upload it to Collab's session storage.
F_H_2D_L = np.loadtxt('F_H_2D.csv', delimiter = ',', dtype = str)

## Reshaping the array from 2D to 3D
F_H_3D = F_H_2D_L.reshape(F_H_2D_L.shape[0], F_H_2D_L.shape[1] // K,
    K)
F_H_3D_size = F_H_3D.shape[0]
```

```
## Converting string data to complex data and removing the initial
## whitespace
F_H_list = []
for k in range(F_H_3D_size):
```

```
  for i in range(K):  # Total rows
    for j in range(K):  # Total columns
      F_H_temp = complex(F_H_3D[k][i][j].strip())
      F_H_list.append(F_H_temp)
F_H_array = np.array(F_H_list)
F_H = F_H_array.reshape((F_H_3D_size, K, K)) # H_size X row X
    column_count
print(F_H.shape)
F_H_size = F_H.shape[0]
# print(F_H)
```

```
import numba as nb

## Function to compute the square of the absolute value of an array
## of complex numbers
@nb.vectorize([nb.float64(nb.complex128),nb.float32(nb.complex64)])
def cmplx_abs_sqr(cmplx_var):
  return cmplx_var.real**2 + cmplx_var.imag**2
```

```
## Function to generate the matrix A (K x K)
def generate_A(F_H_size, K, SINR_P_min, F_H):
  Aij_list = []
  F_H_abs_sqr = cmplx_abs_sqr(F_H)

  for k in range(F_H_size):
    for i in range(K):  # Total rows
      Aj_list =[]
      for j in range(K): # Total columns
        if i==j:
          A = F_H_abs_sqr[k,i,j]
        else:
          A = np.multiply(-SINR_P_min[i], F_H_abs_sqr[k,i,j])
        Aj_list.append(A)
      Aij_list.append(Aj_list)
  Aij_array = np.array(Aij_list)
  Aij = Aij_array.reshape((F_H_size, K, K)) # H_size X row X column
  return Aij
```

```
## Create matrix A
A = generate_A(F_H_size, K, SINR_P_min, F_H)
```

```
print(A.shape)
# print(A)
```

```
## Function to generate the vector b (K x 1)
def generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H):
  bi_list = []
  for k in range(F_H_size):
    for i in range(K):  # Total rows, i.e., total transmitters
      b = np.multiply(SINR_P_min[i], sigma_sqr_noise[i])
      bi_list.append(b)
  bi_array = np.array(bi_list)
  bi = bi_array.reshape((F_H_size, K, 1)) # H_size X row X column
  return bi
```

```
## Create vector b
b = generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H)
print(b.shape)
# print(b)
```

```
## Create matrix A_inv, i.e., the pseudo inverse of matrix A
A_inv = np.linalg.pinv(A)
A_inv[A_inv<0] = 0
print(A_inv.shape)
# print(A_inv)
```

```
## Create a vector p_hat = (A_inv x b)
p_hat = np.matmul(A_inv, b)
print(p_hat.shape)
# print(p_hat)
```

```
## Convert p_max_array to (K x 1) vector
p_max_array = np.array([1.0, 1.0, 1.0, 1.0, 1.0], dtype = float)
p_max_vector = p_max_array.reshape((K, 1)) # row X column
print(p_max_vector)
```

```
## Create a vector X = (p_max_vector - p_hat)
X = p_max_vector - p_hat
print(X.shape)
# print(X)
```

```
## Create a vector beta = MIN[(p_max_vector - p_hat) / A_inv_cv]
beta_list = []
```

```
3
4  for k in range(F_H_size):
5    for i in range(K):  # Total columns
6      ak = A_inv[k,:,i]
7      akr = ak.reshape((K, 1)) # row X column
8      with np.errstate(divide='ignore'):
9        beta_w = np.where(akr != 0.0, np.divide(X[k], akr), np.inf)
10       # beta_w = np.divide(X[k], akr)
11     # beta_w = np.divide(X[k], akr)
12     beta_min = np.amin(beta_w)
13     beta_list.append(beta_min)
14
15 beta_array = np.array(beta_list)
16 beta = beta_array.reshape((F_H_size, K, 1)) # H_size X row X
       column_count
17 print(beta.shape)
18 beta_size = beta.shape[0]
19 # print(beta)
```

```
1  ## Function to split datasets for training, validation, and testing.
2  def split(np_array):
3    # data_size = np_array.shape[0]
4    # train_data_size = int(data_size * 0.8)
5    # valid_data_size = int(data_size * 0.1)
6    # test_data_size = int(data_size * 0.1)
7
8    train_data_size = int(200000)
9    valid_data_size = int(25000)
10   test_data_size = int(25000)
11
12   train_e_indx = train_data_size
13   valid_e_indx = train_e_indx + valid_data_size
14   test_e_indx = valid_e_indx + test_data_size
15   test_data_size_n = test_e_indx - valid_e_indx
16
17   row_count = np_array.shape[1]
18   column_count = np_array.shape[2]
19
20   train_data = np.empty((train_data_size, row_count, column_count),
```

```
    dtype = complex, order = 'C')
  valid_data = np.empty((valid_data_size, row_count, column_count),
    dtype = complex, order = 'C')
  test_data = np.empty((test_data_size_n, row_count, column_count),
    dtype = complex, order = 'C')

  for i in range(train_e_indx):
    train_data[i] = np_array[i]

  xv = 0
  for j in range(train_e_indx, valid_e_indx):
    valid_data[xv] = np_array[j]
    xv = xv + 1

  xt = 0
  for k in range(valid_e_indx, test_e_indx):
    test_data[xt] = np_array[k]
    xt = xt + 1

  # print(train_data.shape, valid_data.shape, test_data.shape)


  ## Training input will be the absolute value
  train_input = np.absolute(train_data)
  valid_input = np.absolute(valid_data)
  test_input = np.absolute(test_data)

  print(train_input.shape, valid_input.shape, test_input.shape)

  return [train_input, valid_input, test_input, test_data]
```

```
## Split F_H matrix
F_H_S = split(F_H)
train_input_F_H = F_H_S[0]
valid_input_F_H = F_H_S[1]
test_input_F_H = F_H_S[2]
test_data_F_H = F_H_S[3]
```

```
## Split A_inv matrix
A_inv_S = split(A_inv)
```

```
train_input_A_inv = A_inv_S[0]
valid_input_A_inv = A_inv_S[1]
test_input_A_inv = A_inv_S[2]
test_data_A_inv = A_inv_S[3]
```

```
## Split b vector
b_S = split(b)
train_input_b = b_S[0]
valid_input_b = b_S[1]
test_input_b = b_S[2]
test_data_b = b_S[3]
```

```
## Split X vector
X_S = split(X)
train_input_X = X_S[0]
valid_input_X = X_S[1]
test_input_X = X_S[2]
test_data_X = X_S[3]
```

```
## Split beta vector
beta_S = split(beta)
train_input_beta = beta_S[0]
valid_input_beta = beta_S[1]
test_input_beta = beta_S[2]
test_data_beta = beta_S[3]
```

```
## Split p_hat vector
p_hat_S = split(p_hat)
train_input_p_hat = p_hat_S[0]
valid_input_p_hat = p_hat_S[1]
test_input_p_hat = p_hat_S[2]
test_data_p_hat = p_hat_S[3]
```

```
## Define the DNN model - The Functional API
import tensorflow as tf
from tensorflow import keras
## from tensorflow.keras import layers # shows warning
from keras.api._v2.keras import layers
from keras.layers import Input, Lambda
from keras.models import Model

```

```
9
10 hij_inputs = keras.Input(shape=(K,K), name = "hij_inputs")
11 f1 = layers.Flatten(name = "flatten_layer_hij")(hij_inputs)
12
13 d1 = layers.Dense(2*K*K, activation="relu", name = "dense_layer_1")(
       f1)
14 b1 = layers.BatchNormalization(name = "batch_norm_layer_1")(d1)
15
16 d2 = layers.Dense(K*K, activation="relu", name = "dense_layer_2")(b1)
17 b2 = layers.BatchNormalization(name = "batch_norm_layer_2")(d2)
18
19 # meu = layers.Dense(K, activation="relu", name = "meu")(b2)
20 meu = layers.Dense(K, activation="sigmoid", name = "meu")(b2)
21
22 A_inv_inputs = keras.Input(shape=(K,K), name = "A_inv_inputs")
23 f2 = layers.Flatten(name = "flatten_layer_A_inv")(A_inv_inputs)
24
25 X_inputs = keras.Input(shape=(K,1), name = "X_inputs")
26 f3 = layers.Flatten(name = "flatten_layer_X")(X_inputs)
27
28 beta_inputs = keras.Input(shape=(K,1), name = "beta_inputs")
29 f4 = layers.Flatten(name = "flatten_layer_beta")(beta_inputs)
30
31 p_hat_inputs = keras.Input(shape=(K,1), name = "p_hat_inputs")
32 f5 = layers.Flatten(name = "flatten_layer_p_hat")(p_hat_inputs)
33
34 def custom_layer(tensor):
35   t_A_inv = tensor[0]
36   t_X = tensor[1]
37   t_beta = tensor[2]
38   t_p_hat = tensor[3]
39   t_meu = tensor[4]
40
41   A_inv_cl = tf.reshape(t_A_inv[:,0:K*K], (-1,K,K))
42   X_cl = tf.reshape(t_X[:,0:K*1], (-1,K,1))
43   beta_cl = tf.reshape(t_beta[:,0:K*1], (-1,K,1))
44   p_hat_cl = tf.reshape(t_p_hat[:,0:K*1], (-1,K,1))
45   meu_cl = tf.reshape(t_meu[:,0:K*1], (-1,K,1))
46
```

```
  meu_ewm = tf.math.multiply(beta_cl, meu_cl)

  alpha_dnumr = tf.matmul(A_inv_cl, meu_ewm)
  alpha_whole = tf.divide(X_cl, alpha_dnumr)
  alpha = tf.reduce_min(alpha_whole, axis = 1, keepdims = True)
  max_p = tf.constant([1.0])
  alpha = tf.math.minimum(max_p, alpha)
  meu_P = tf.multiply(meu_ewm, alpha)

  Z_cl = tf.matmul(A_inv_cl, meu_P)
  P_hat_cl = tf.add(p_hat_cl, Z_cl)
  P_hat_cl_Norm = tf.math.divide(P_hat_cl, tf.reduce_max(P_hat_cl,
    axis = 1, keepdims = True))

  # return P_hat_cl
  return P_hat_cl_Norm

lambda_layer = tf.keras.layers.Lambda(custom_layer, name="
    lambda_layer")([f2, f3, f4, f5, meu])
f6 = layers.Flatten(name = "flatten_layer_output")(lambda_layer)

model = keras.Model(inputs = [hij_inputs, A_inv_inputs, X_inputs,
    beta_inputs, p_hat_inputs], outputs = f6, name = "functional_api"
    )
model.summary()
```

```
## Plot the model as a graph
keras.utils.plot_model(model, "Functional_API_Model.png")
```

```
## Display the input and output shapes of each layer
keras.utils.plot_model(model, "Functional_API_Model_with_shape_info.
    png", show_shapes=True)
```

```
## Convert sigma_sqr_noise from numpy array to tensor
sigma_sqr_noise_t = tf.convert_to_tensor(sigma_sqr_noise, dtype =
    float)
tf.print(sigma_sqr_noise_t)
```

```
## The customized loss function

def custom_loss(y_true, y_pred):
```

```
  # p = y_pred
  p = tf.math.multiply(p_max, y_pred)
  hij = tf.reshape(y_true[:,0:K*K], (-1,K,K))
  hij_abs_sqr = tf.math.square(tf.math.abs(hij))

  R_P = 0.0
  for i in range(K):  # Total rows
    ph = 0.0
    for j in range(K):  # Total columns
      ph_j = tf.math.multiply(p[:,j], hij_abs_sqr[:,i,j])
      ph = tf.math.add(ph, ph_j)

    numr = tf.math.multiply(p[:,i], hij_abs_sqr[:,i,i])
    dnumr = tf.math.add(sigma_sqr_noise_t[i], tf.math.subtract(ph,
   numr))
    SINR_i = tf.math.divide(numr, dnumr)
    R_P = tf.math.add(R_P, (tf.math.log(1 + SINR_i)/tf.math.log(2.0))
   )

  loss = -R_P
  loss = tf.reduce_mean(loss) # batch mean
  return loss
```

```
## Build and compile the DNN model
## Training and Testing
import matplotlib.pyplot as plt

optA = tf.keras.optimizers.Adam(learning_rate = 0.0001)
model.compile(optimizer = optA, loss = custom_loss)

train_input = [train_input_F_H, train_input_A_inv, train_input_X,
   train_input_beta, train_input_p_hat]
valid_input = [valid_input_F_H, valid_input_A_inv, valid_input_X,
   valid_input_beta, valid_input_p_hat]

history = model.fit(train_input, train_input_F_H, epochs = 50,
   validation_data = (valid_input, valid_input_F_H), batch_size =
   1000)

```

```
plt.plot(history.epoch, history.history['loss'], color = "blue",
    label = "Training")
plt.plot(history.epoch, history.history['val_loss'], color="black",
    label = "Validation")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
plt.show()
```

```
## Constraint violation probability and
## finding indexes of test_input_F_H matrix with the hij set that do
## not satisfy constraint on the minimum SINR_P_min rate but satisfy
## the maximum transmit power p_max

test_input = [test_input_F_H, test_input_A_inv, test_input_X,
    test_input_beta, test_input_p_hat]
# output_P_hat_temp = model.predict(test_input)
output_P_hat_temp = np.multiply(p_max, model.predict(test_input))
output_P_hat = output_P_hat_temp.reshape((output_P_hat_temp.shape[0],
     output_P_hat_temp.shape[1], 1)) # test_input_F_H_size X row X
    column
output_P_hat_size = output_P_hat.shape[0]
test_data_F_H_abs_sqr = cmplx_abs_sqr(test_data_F_H)

indx_n = []
count_v = 0

for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(output_P_hat[k,j], test_data_F_H_abs_sqr[k,i
    ,j])
      ph = ph + ph_j

    numr = np.multiply(output_P_hat[k,i], test_data_F_H_abs_sqr[k,i,i
    ])
    dnumr = sigma_sqr_noise[i] + ph - numr
    SINR_out = np.divide(numr, dnumr)
```

```
26
27      if np.round(SINR_out, decimals= 3) < SINR_P_min[i]:
28        indx_n.append(k)
29        count_v = count_v + 1
30        # print(SINR_out)
31        break
32
33 violation_prb = (count_v / output_P_hat_size) * 100
34 print("Constraints Violation Probability: {:.2f}%".format(
       violation_prb))
35 # print(len(indx_n))
36 # print(indx_n)
```

```
1  ## Function to calculate the average sum rate
2  # Here, p_model is the output of DNN, and it is a 2D array.
3  import math
4
5  def average_sum_rate(hij, p_model, sigma_sqr_noise, K):
6    R = 0
7    hij_size = hij.shape[0]
8    hij_abs_sqr = cmplx_abs_sqr(hij)
9
10   for k in range(hij_size):
11     for i in range(K):  # Total rows
12       phn = 0
13       for j in range(K):  # Total columns
14         phn_j = np.multiply(p_model[k,j], hij_abs_sqr[k,i,j])
15         phn = phn + phn_j
16
17       numr_s = np.multiply(p_model[k,i], hij_abs_sqr[k,i,i])
18       dnumr_s = sigma_sqr_noise[i] + phn - numr_s
19       R_temp = math.log2(1 + np.divide(numr_s, dnumr_s))
20       R = R + R_temp
21
22   return (R/hij_size)
```

```
1  # DNN Sum Rate for test_data_F_H
2  sumrate_F_H = average_sum_rate(test_data_F_H, output_P_hat,
       sigma_sqr_noise, K)
3  print("Average Sum Rate for all H matrices: {:.3f} Bit/Second/Hertz".
```

```
    format(sumrate_F_H))
```

```
## Checking (A_inv x b), i.e., the power for negative values
count_n = 0
for c in range(output_P_hat_size):
  p_temp = np.matmul(test_input_A_inv[c], test_input_b[c])
  if np.any(p_temp < 0):
    count_n = count_n + 1
    print(c,'\n')
    print(p_temp)

print(count_n)
```

```
## Checking P_hat, i.e., the power for test_data_F_H for negative
## values and Hit Rate i.e. percentage for 0 <= P_hat <= p_max
count_p = 0
count_n = 0

for n in range(output_P_hat_size):
  P_max = np.amax(output_P_hat[n])
  if np.round(P_max, decimals = 3) <= 1:
    count_p = count_p + 1

  if np.any(output_P_hat[n] < 0):
    count_n = count_n + 1
    print(n,'\n')
    print(output_P_hat)

p_hit_rate = (count_p / output_P_hat_size) * 100
print("Hit Rate for Power : {:.2f}%".format(p_hit_rate))
print("Negative power count: ", count_n)
```

C.3.2 For enhanced generalization capacity

```
import numpy as np

## Number of transmitter-receiver pairs
K = 8

## Minimum rate for the achievable SINR of multiple concurrent
    transmissions
SINR_P_min = np.array([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2])

## Maximum transmit power
p_max = 1.0
```

```
## Loading a NumPy array from a CSV file
# Loading F_H array from a CSV file
from numpy import loadtxt

## Reading an array from the file
# If we want to read a file from our local drive, we have to first
    upload it to Collab's session storage.
F_H_2D_L_0dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_0dB.csv',
    delimiter = ',', dtype = str)
F_H_2D_L_10dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_10dB.csv',
    delimiter = ',', dtype = str)
F_H_2D_L_20dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_20dB.csv',
    delimiter = ',', dtype = str)
F_H_2D_L_30dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_30dB.csv',
    delimiter = ',', dtype = str)
F_H_2D_L_40dB = np.loadtxt('/content/drive/MyDrive/F_H_2D_40dB.csv',
    delimiter = ',', dtype = str)

# ## Reshaping the array from 2D to 3D
F_H_3D_0dB = F_H_2D_L_0dB.reshape(F_H_2D_L_0dB.shape[0], F_H_2D_L_0dB
    .shape[1] // K, K)
F_H_3D_10dB = F_H_2D_L_10dB.reshape(F_H_2D_L_10dB.shape[0],
    F_H_2D_L_10dB.shape[1] // K, K)
F_H_3D_20dB = F_H_2D_L_20dB.reshape(F_H_2D_L_20dB.shape[0],
    F_H_2D_L_20dB.shape[1] // K, K)
```

```
F_H_3D_30dB = F_H_2D_L_30dB.reshape(F_H_2D_L_30dB.shape[0],
    F_H_2D_L_30dB.shape[1] // K, K)
F_H_3D_40dB = F_H_2D_L_40dB.reshape(F_H_2D_L_40dB.shape[0],
    F_H_2D_L_40dB.shape[1] // K, K)

F_H_3D_0dB_size = F_H_3D_0dB.shape[0]
F_H_3D_10dB_size = F_H_3D_10dB.shape[0]
F_H_3D_20dB_size = F_H_3D_20dB.shape[0]
F_H_3D_30dB_size = F_H_3D_30dB.shape[0]
F_H_3D_40dB_size = F_H_3D_40dB.shape[0]
```

```
## Function to convert string data to complex data and to remove the
    initial whitespace
def cnvrt_2_cmplx_data(F_H_3D_size, F_H_3D):
  F_H_list = []
  for k in range(F_H_3D_size):
    for i in range(K):  # Total rows
      for j in range(K):  # Total columns
        F_H_temp = complex(F_H_3D[k][i][j].strip())
        F_H_list.append(F_H_temp)
  F_H_array = np.array(F_H_list)
  F_H = F_H_array.reshape((F_H_3D_size, K, K)) # H_size X row X
    column_count
  return F_H
```

```
## Converting string data to complex data and removing the initial
    whitespace
F_H_0dB = cnvrt_2_cmplx_data(F_H_3D_0dB_size, F_H_3D_0dB)
F_H_10dB = cnvrt_2_cmplx_data(F_H_3D_10dB_size, F_H_3D_10dB)
F_H_20dB = cnvrt_2_cmplx_data(F_H_3D_20dB_size, F_H_3D_20dB)
F_H_30dB = cnvrt_2_cmplx_data(F_H_3D_30dB_size, F_H_3D_30dB)
F_H_40dB = cnvrt_2_cmplx_data(F_H_3D_40dB_size, F_H_3D_40dB)

print(F_H_0dB.shape)
print(F_H_10dB.shape)
print(F_H_20dB.shape)
print(F_H_30dB.shape)
print(F_H_40dB.shape)

F_H_0dB_size = F_H_0dB.shape[0]
```

```
F_H_10dB_size = F_H_10dB.shape[0]
F_H_20dB_size = F_H_20dB.shape[0]
F_H_30dB_size = F_H_30dB.shape[0]
F_H_40dB_size = F_H_40dB.shape[0]

# print(F_H_0dB)
# print(F_H_10dB)
# print(F_H_20dB)
# print(F_H_30dB)
# print(F_H_40dB)
```

```
import numba as nb

## Function to compute the square of the absolute value of an array
    of complex numbers
@nb.vectorize([nb.float64(nb.complex128),nb.float32(nb.complex64)])
def cmplx_abs_sqr(cmplx_var):
  return cmplx_var.real**2 + cmplx_var.imag**2
```

```
## Function to generate the matrix A (K x K)
def generate_A(F_H_size, K, SINR_P_min, F_H):
  Aij_list = []
  F_H_abs_sqr = cmplx_abs_sqr(F_H)

  for k in range(F_H_size):
    for i in range(K):  # Total rows
      Aj_list =[]
      for j in range(K): # Total columns
        if i==j:
          A = F_H_abs_sqr[k,i,j]
        else:
          A = np.multiply(-SINR_P_min[i], F_H_abs_sqr[k,i,j])
        Aj_list.append(A)
      Aij_list.append(Aj_list)
  Aij_array = np.array(Aij_list)
  Aij = Aij_array.reshape((F_H_size, K, K)) # H_size X row X column
  return Aij
```

```
## Create matrix A
A_0dB = generate_A(F_H_0dB_size, K, SINR_P_min, F_H_0dB)
```

```
A_10dB = generate_A(F_H_10dB_size, K, SINR_P_min, F_H_10dB)
A_20dB = generate_A(F_H_20dB_size, K, SINR_P_min, F_H_20dB)
A_30dB = generate_A(F_H_30dB_size, K, SINR_P_min, F_H_30dB)
A_40dB = generate_A(F_H_40dB_size, K, SINR_P_min, F_H_40dB)

print(A_0dB.shape)
print(A_10dB.shape)
print(A_20dB.shape)
print(A_30dB.shape)
print(A_40dB.shape)

# print(A_0dB)
# print(A_10dB)
# print(A_20dB)
# print(A_30dB)
# print(A_40dB)
```

```
## Variances for noise signals
sigma_sqr_noise_0dB = np.array([1e-0, 1e-0, 1e-0, 1e-0, 1e-0, 1e-0, 1e-0, 1e-0], dtype = float)
sigma_sqr_noise_10dB = np.array([1e-1, 1e-1, 1e-1, 1e-1, 1e-1, 1e-1, 1e-1, 1e-1], dtype = float)
sigma_sqr_noise_20dB = np.array([1e-2, 1e-2, 1e-2, 1e-2, 1e-2, 1e-2, 1e-2, 1e-2], dtype = float)
sigma_sqr_noise_30dB = np.array([1e-3, 1e-3, 1e-3, 1e-3, 1e-3, 1e-3, 1e-3, 1e-3], dtype = float)
sigma_sqr_noise_40dB = np.array([1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4], dtype = float)
```

```
## Function to generate the vector b (K x 1)
def generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H):
  bi_list = []
  for k in range(F_H_size):
    for i in range(K):  # Total rows, i.e., total transmitters
      b = np.multiply(SINR_P_min[i], sigma_sqr_noise[i])
      bi_list.append(b)
  bi_array = np.array(bi_list)
  bi = bi_array.reshape((F_H_size, K, 1)) # H_size X row X column
  return bi
```

```
## Create vector b
b_0dB = generate_b(F_H_0dB_size, K, SINR_P_min, sigma_sqr_noise_0dB,
    F_H_0dB)
b_10dB = generate_b(F_H_10dB_size, K, SINR_P_min,
    sigma_sqr_noise_10dB, F_H_10dB)
b_20dB = generate_b(F_H_20dB_size, K, SINR_P_min,
    sigma_sqr_noise_20dB, F_H_20dB)
b_30dB = generate_b(F_H_30dB_size, K, SINR_P_min,
    sigma_sqr_noise_30dB, F_H_30dB)
b_40dB = generate_b(F_H_40dB_size, K, SINR_P_min,
    sigma_sqr_noise_40dB, F_H_40dB)

print(b_0dB.shape)
print(b_10dB.shape)
print(b_20dB.shape)
print(b_30dB.shape)
print(b_40dB.shape)

# print(b_0dB)
# print(b_10dB)
# print(b_20dB)
# print(b_30dB)
# print(b_40dB)
```

```
## Create matrix A_inv, i.e., the pseudo inverse of matrix A
A_inv_0dB = np.linalg.pinv(A_0dB)
A_inv_10dB = np.linalg.pinv(A_10dB)
A_inv_20dB = np.linalg.pinv(A_20dB)
A_inv_30dB = np.linalg.pinv(A_30dB)
A_inv_40dB = np.linalg.pinv(A_40dB)

A_inv_0dB[A_inv_0dB<0] = 0
A_inv_10dB[A_inv_10dB<0] = 0
A_inv_20dB[A_inv_20dB<0] = 0
A_inv_30dB[A_inv_30dB<0] = 0
A_inv_40dB[A_inv_40dB<0] = 0

print(A_inv_0dB.shape)
print(A_inv_10dB.shape)
```

```
print(A_inv_20dB.shape)
print(A_inv_30dB.shape)
print(A_inv_40dB.shape)

# print(A_inv_0dB)
# print(A_inv_10dB)
# print(A_inv_20dB)
# print(A_inv_30dB)
# print(A_inv_40dB)
```

```
## Create a vector p_hat = (A_inv x b)
p_hat_0dB = np.matmul(A_inv_0dB, b_0dB)
p_hat_10dB = np.matmul(A_inv_10dB, b_10dB)
p_hat_20dB = np.matmul(A_inv_20dB, b_20dB)
p_hat_30dB = np.matmul(A_inv_30dB, b_30dB)
p_hat_40dB = np.matmul(A_inv_40dB, b_40dB)

print(p_hat_0dB.shape)
print(p_hat_10dB.shape)
print(p_hat_20dB.shape)
print(p_hat_30dB.shape)
print(p_hat_40dB.shape)

# print(p_hat_0dB)
# print(p_hat_10dB)
# print(p_hat_20dB)
# print(p_hat_30dB)
# print(p_hat_40dB)
```

```
## Convert p_max_array to (K x 1) vector
p_max_array = np.array([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
    dtype = float)
p_max_vector = p_max_array.reshape((K, 1)) # row X column
print(p_max_vector)
```

```
## Create a vector X = (p_max_vector - p_hat)
X_0dB = p_max_vector - p_hat_0dB
X_10dB = p_max_vector - p_hat_10dB
X_20dB = p_max_vector - p_hat_20dB
X_30dB = p_max_vector - p_hat_30dB
```

```
X_40dB = p_max_vector - p_hat_40dB

print(X_0dB.shape)
print(X_10dB.shape)
print(X_20dB.shape)
print(X_30dB.shape)
print(X_40dB.shape)

# print(X_0dB)
# print(X_10dB)
# print(X_20dB)
# print(X_30dB)
# print(X_40dB)
```

```
## Function to generate a vector beta = MIN[(p_max_vector - p_hat) / A_inv_cv]
def generate_beta(F_H_size, A_inv, X):
  beta_list = []

  for k in range(F_H_size):
    for i in range(K):  # Total columns
      ak = A_inv[k,:,i]
      akr = ak.reshape((K, 1)) # row X column
      with np.errstate(divide='ignore'):
        beta_w = np.where(akr != 0.0, np.divide(X[k], akr), np.inf)
        # beta_w = np.divide(X[k], akr)
      # beta_w = np.divide(X[k], akr)
      beta_min = np.amin(beta_w)
      beta_list.append(beta_min)

  beta_array = np.array(beta_list)
  beta = beta_array.reshape((F_H_size, K, 1)) # H_size X row X column_count
  return beta
```

```
## Generate a vector beta = MIN[(p_max_vector - p_hat) / A_inv_cv]
beta_0dB = generate_beta(F_H_0dB_size, A_inv_0dB, X_0dB)
beta_10dB = generate_beta(F_H_10dB_size, A_inv_10dB, X_10dB)
beta_20dB = generate_beta(F_H_20dB_size, A_inv_20dB, X_20dB)
beta_30dB = generate_beta(F_H_30dB_size, A_inv_30dB, X_30dB)
```

```
beta_40dB = generate_beta(F_H_40dB_size, A_inv_40dB, X_40dB)

print(beta_0dB.shape)
print(beta_10dB.shape)
print(beta_20dB.shape)
print(beta_30dB.shape)
print(beta_40dB.shape)

beta_0dB_size = beta_0dB.shape[0]
beta_10dB_size = beta_10dB.shape[0]
beta_20dB_size = beta_20dB.shape[0]
beta_30dB_size = beta_30dB.shape[0]
beta_40dB_size = beta_40dB.shape[0]

# print(beta_0dB)
# print(beta_10dB)
# print(beta_20dB)
# print(beta_30dB)
# print(beta_40dB)
```

```
## Function to split datasets for training, validation, and testing.
def split(np_array):
  # data_size = np_array.shape[0]
  # train_data_size = int(data_size * 0.8)
  # valid_data_size = int(data_size * 0.1)
  # test_data_size = int(data_size * 0.1)

  train_data_size = int(200000)
  valid_data_size = int(25000)
  test_data_size = int(25000)

  train_e_indx = train_data_size
  valid_e_indx = train_e_indx + valid_data_size
  test_e_indx = valid_e_indx + test_data_size
  test_data_size_n = test_e_indx - valid_e_indx

  row_count = np_array.shape[1]
  column_count = np_array.shape[2]

```

```
  train_data = np.empty((train_data_size, row_count, column_count),
   dtype = complex, order = 'C')
  valid_data = np.empty((valid_data_size, row_count, column_count),
   dtype = complex, order = 'C')
  test_data = np.empty((test_data_size_n, row_count, column_count),
   dtype = complex, order = 'C')

  for i in range(train_e_indx):
    train_data[i] = np_array[i]

  xv = 0
  for j in range(train_e_indx, valid_e_indx):
    valid_data[xv] = np_array[j]
    xv = xv + 1

  xt = 0
  for k in range(valid_e_indx, test_e_indx):
    test_data[xt] = np_array[k]
    xt = xt + 1

  # print(train_data.shape, valid_data.shape, test_data.shape)


  ## Training input will be the absolute value
  train_input = np.absolute(train_data)
  valid_input = np.absolute(valid_data)
  test_input = np.absolute(test_data)

  print(train_input.shape, valid_input.shape, test_input.shape)

  return [train_input, valid_input, test_input, test_data]
```

```
## Split F_H matrix
F_H_S_0dB = split(F_H_0dB)
train_input_F_H_0dB = F_H_S_0dB[0]
valid_input_F_H_0dB = F_H_S_0dB[1]
test_input_F_H_0dB = F_H_S_0dB[2]
test_data_F_H_0dB = F_H_S_0dB[3]

```

```
F_H_S_10dB = split(F_H_10dB)
train_input_F_H_10dB = F_H_S_10dB[0]
valid_input_F_H_10dB = F_H_S_10dB[1]
test_input_F_H_10dB = F_H_S_10dB[2]
test_data_F_H_10dB = F_H_S_10dB[3]

F_H_S_20dB = split(F_H_20dB)
train_input_F_H_20dB = F_H_S_20dB[0]
valid_input_F_H_20dB = F_H_S_20dB[1]
test_input_F_H_20dB = F_H_S_20dB[2]
test_data_F_H_20dB = F_H_S_20dB[3]

F_H_S_30dB = split(F_H_30dB)
train_input_F_H_30dB = F_H_S_30dB[0]
valid_input_F_H_30dB = F_H_S_30dB[1]
test_input_F_H_30dB = F_H_S_30dB[2]
test_data_F_H_30dB = F_H_S_30dB[3]

F_H_S_40dB = split(F_H_40dB)
train_input_F_H_40dB = F_H_S_40dB[0]
valid_input_F_H_40dB = F_H_S_40dB[1]
test_input_F_H_40dB = F_H_S_40dB[2]
test_data_F_H_40dB = F_H_S_40dB[3]
```

```
## Split A_inv matrix
A_inv_S_0dB = split(A_inv_0dB)
train_input_A_inv_0dB = A_inv_S_0dB[0]
valid_input_A_inv_0dB = A_inv_S_0dB[1]
test_input_A_inv_0dB = A_inv_S_0dB[2]
test_data_A_inv_0dB = A_inv_S_0dB[3]

A_inv_S_10dB = split(A_inv_10dB)
train_input_A_inv_10dB = A_inv_S_10dB[0]
valid_input_A_inv_10dB = A_inv_S_10dB[1]
test_input_A_inv_10dB = A_inv_S_10dB[2]
test_data_A_inv_10dB = A_inv_S_10dB[3]

A_inv_S_20dB = split(A_inv_20dB)
train_input_A_inv_20dB = A_inv_S_20dB[0]
```

```
valid_input_A_inv_20dB = A_inv_S_20dB[1]
test_input_A_inv_20dB = A_inv_S_20dB[2]
test_data_A_inv_20dB = A_inv_S_20dB[3]

A_inv_S_30dB = split(A_inv_30dB)
train_input_A_inv_30dB = A_inv_S_30dB[0]
valid_input_A_inv_30dB = A_inv_S_30dB[1]
test_input_A_inv_30dB = A_inv_S_30dB[2]
test_data_A_inv_30dB = A_inv_S_30dB[3]

A_inv_S_40dB = split(A_inv_40dB)
train_input_A_inv_40dB = A_inv_S_40dB[0]
valid_input_A_inv_40dB = A_inv_S_40dB[1]
test_input_A_inv_40dB = A_inv_S_40dB[2]
test_data_A_inv_40dB = A_inv_S_40dB[3]
```

```
## Split b vector
b_S_0dB = split(b_0dB)
train_input_b_0dB = b_S_0dB[0]
valid_input_b_0dB = b_S_0dB[1]
test_input_b_0dB = b_S_0dB[2]
test_data_b_0dB = b_S_0dB[3]

b_S_10dB = split(b_10dB)
train_input_b_10dB = b_S_10dB[0]
valid_input_b_10dB = b_S_10dB[1]
test_input_b_10dB = b_S_10dB[2]
test_data_b_10dB = b_S_10dB[3]

b_S_20dB = split(b_20dB)
train_input_b_20dB = b_S_20dB[0]
valid_input_b_20dB = b_S_20dB[1]
test_input_b_20dB = b_S_20dB[2]
test_data_b_20dB = b_S_20dB[3]

b_S_30dB = split(b_30dB)
train_input_b_30dB = b_S_30dB[0]
valid_input_b_30dB = b_S_30dB[1]
test_input_b_30dB = b_S_30dB[2]
```

```
test_data_b_30dB = b_S_30dB[3]

b_S_40dB = split(b_40dB)
train_input_b_40dB = b_S_40dB[0]
valid_input_b_40dB = b_S_40dB[1]
test_input_b_40dB = b_S_40dB[2]
test_data_b_40dB = b_S_40dB[3]
```

```
## Split X vector
X_S_0dB = split(X_0dB)
train_input_X_0dB = X_S_0dB[0]
valid_input_X_0dB = X_S_0dB[1]
test_input_X_0dB = X_S_0dB[2]
test_data_X_0dB = X_S_0dB[3]

X_S_10dB = split(X_10dB)
train_input_X_10dB = X_S_10dB[0]
valid_input_X_10dB = X_S_10dB[1]
test_input_X_10dB = X_S_10dB[2]
test_data_X_10dB = X_S_10dB[3]

X_S_20dB = split(X_20dB)
train_input_X_20dB = X_S_20dB[0]
valid_input_X_20dB = X_S_20dB[1]
test_input_X_20dB = X_S_20dB[2]
test_data_X_20dB = X_S_20dB[3]

X_S_30dB = split(X_30dB)
train_input_X_30dB = X_S_30dB[0]
valid_input_X_30dB = X_S_30dB[1]
test_input_X_30dB = X_S_30dB[2]
test_data_X_30dB = X_S_30dB[3]

X_S_40dB = split(X_40dB)
train_input_X_40dB = X_S_40dB[0]
valid_input_X_40dB = X_S_40dB[1]
test_input_X_40dB = X_S_40dB[2]
test_data_X_40dB = X_S_40dB[3]
```

```
## Split beta vector
```

```
beta_S_0dB = split(beta_0dB)
train_input_beta_0dB = beta_S_0dB[0]
valid_input_beta_0dB = beta_S_0dB[1]
test_input_beta_0dB = beta_S_0dB[2]
test_data_beta_0dB = beta_S_0dB[3]

beta_S_10dB = split(beta_10dB)
train_input_beta_10dB = beta_S_10dB[0]
valid_input_beta_10dB = beta_S_10dB[1]
test_input_beta_10dB = beta_S_10dB[2]
test_data_beta_10dB = beta_S_10dB[3]

beta_S_20dB = split(beta_20dB)
train_input_beta_20dB = beta_S_20dB[0]
valid_input_beta_20dB = beta_S_20dB[1]
test_input_beta_20dB = beta_S_20dB[2]
test_data_beta_20dB = beta_S_20dB[3]

beta_S_30dB = split(beta_30dB)
train_input_beta_30dB = beta_S_30dB[0]
valid_input_beta_30dB = beta_S_30dB[1]
test_input_beta_30dB = beta_S_30dB[2]
test_data_beta_30dB = beta_S_30dB[3]

beta_S_40dB = split(beta_40dB)
train_input_beta_40dB = beta_S_40dB[0]
valid_input_beta_40dB = beta_S_40dB[1]
test_input_beta_40dB = beta_S_40dB[2]
test_data_beta_40dB = beta_S_40dB[3]
```

```
## Split p_hat vector
p_hat_S_0dB = split(p_hat_0dB)
train_input_p_hat_0dB = p_hat_S_0dB[0]
valid_input_p_hat_0dB = p_hat_S_0dB[1]
test_input_p_hat_0dB = p_hat_S_0dB[2]
test_data_p_hat_0dB = p_hat_S_0dB[3]

p_hat_S_10dB = split(p_hat_10dB)
train_input_p_hat_10dB = p_hat_S_10dB[0]
```

```
valid_input_p_hat_10dB = p_hat_S_10dB[1]
test_input_p_hat_10dB = p_hat_S_10dB[2]
test_data_p_hat_10dB = p_hat_S_10dB[3]

p_hat_S_20dB = split(p_hat_20dB)
train_input_p_hat_20dB = p_hat_S_20dB[0]
valid_input_p_hat_20dB = p_hat_S_20dB[1]
test_input_p_hat_20dB = p_hat_S_20dB[2]
test_data_p_hat_20dB = p_hat_S_20dB[3]

p_hat_S_30dB = split(p_hat_30dB)
train_input_p_hat_30dB = p_hat_S_30dB[0]
valid_input_p_hat_30dB = p_hat_S_30dB[1]
test_input_p_hat_30dB = p_hat_S_30dB[2]
test_data_p_hat_30dB = p_hat_S_30dB[3]

p_hat_S_40dB = split(p_hat_40dB)
train_input_p_hat_40dB = p_hat_S_40dB[0]
valid_input_p_hat_40dB = p_hat_S_40dB[1]
test_input_p_hat_40dB = p_hat_S_40dB[2]
test_data_p_hat_40dB = p_hat_S_40dB[3]
```

```
## Create EsN0 vector
EsN0_array_0dB = np.full(shape = F_H_0dB_size, fill_value = 0, dtype
    = int)
EsN0_array_10dB = np.full(shape = F_H_10dB_size, fill_value = 10,
    dtype = int)
EsN0_array_20dB = np.full(shape = F_H_20dB_size, fill_value = 20,
    dtype = int)
EsN0_array_30dB = np.full(shape = F_H_30dB_size, fill_value = 30,
    dtype = int)
EsN0_array_40dB = np.full(shape = F_H_40dB_size, fill_value = 40,
    dtype = int)

EsN0_vector_0dB = EsN0_array_0dB.reshape((F_H_0dB_size, 1)) # row X
    column
EsN0_vector_10dB = EsN0_array_10dB.reshape((F_H_10dB_size, 1)) # row
    X column
EsN0_vector_20dB = EsN0_array_20dB.reshape((F_H_20dB_size, 1)) # row
```

```
    X column
EsN0_vector_30dB = EsN0_array_30dB.reshape((F_H_30dB_size, 1)) # row
    X column
EsN0_vector_40dB = EsN0_array_40dB.reshape((F_H_40dB_size, 1)) # row
    X column

print(EsN0_vector_0dB.shape)
print(EsN0_vector_10dB.shape)
print(EsN0_vector_20dB.shape)
print(EsN0_vector_30dB.shape)
print(EsN0_vector_40dB.shape)
```

```
## Function to split EsN0 vector for training, validation, and
    testing.
def split_EsN0(np_vector):
  # data_size = np_vector.shape[0]
  # train_data_size = int(data_size * 0.8)
  # valid_data_size = int(data_size * 0.1)
  # test_data_size = int(data_size * 0.1)

  train_data_size = int(200000)
  valid_data_size = int(25000)
  test_data_size = int(25000)

  train_e_indx = train_data_size
  valid_e_indx = train_e_indx + valid_data_size
  test_e_indx = valid_e_indx + test_data_size
  test_data_size_n = test_e_indx - valid_e_indx

  row_count = np_vector.shape[1]
  column_count = 1

  train_data = np.empty((train_data_size, row_count, column_count),
    dtype = int, order = 'C')
  valid_data = np.empty((valid_data_size, row_count, column_count),
    dtype = int, order = 'C')
  test_data = np.empty((test_data_size_n, row_count, column_count),
    dtype = int, order = 'C')

```

```
  for i in range(train_e_indx):
    train_data[i] = np_vector[i]

  xv = 0
  for j in range(train_e_indx, valid_e_indx):
    valid_data[xv] = np_vector[j]
    xv = xv + 1

  xt = 0
  for k in range(valid_e_indx, test_e_indx):
    test_data[xt] = np_vector[k]
    xt = xt + 1

  # print(train_data.shape, valid_data.shape, test_data.shape)


  ## Training input will be the absolute value
  train_input = np.absolute(train_data)
  valid_input = np.absolute(valid_data)
  test_input = np.absolute(test_data)

  print(train_input.shape, valid_input.shape, test_input.shape)

  return [train_input, valid_input, test_input, test_data]
```

```
## Split EsN0 vector
EsN0_S_0dB = split_EsN0(EsN0_vector_0dB)
train_input_EsN0_0dB = EsN0_S_0dB[0]
valid_input_EsN0_0dB = EsN0_S_0dB[1]
test_input_EsN0_0dB = EsN0_S_0dB[2]
test_data_EsN0_0dB = EsN0_S_0dB[3]

EsN0_S_10dB = split_EsN0(EsN0_vector_10dB)
train_input_EsN0_10dB = EsN0_S_10dB[0]
valid_input_EsN0_10dB = EsN0_S_10dB[1]
test_input_EsN0_10dB = EsN0_S_10dB[2]
test_data_EsN0_10dB = EsN0_S_10dB[3]

EsN0_S_20dB = split_EsN0(EsN0_vector_20dB)
```

```
train_input_EsN0_20dB = EsN0_S_20dB[0]
valid_input_EsN0_20dB = EsN0_S_20dB[1]
test_input_EsN0_20dB = EsN0_S_20dB[2]
test_data_EsN0_20dB = EsN0_S_20dB[3]

EsN0_S_30dB = split_EsN0(EsN0_vector_30dB)
train_input_EsN0_30dB = EsN0_S_30dB[0]
valid_input_EsN0_30dB = EsN0_S_30dB[1]
test_input_EsN0_30dB = EsN0_S_30dB[2]
test_data_EsN0_30dB = EsN0_S_30dB[3]

EsN0_S_40dB = split_EsN0(EsN0_vector_40dB)
train_input_EsN0_40dB = EsN0_S_40dB[0]
valid_input_EsN0_40dB = EsN0_S_40dB[1]
test_input_EsN0_40dB = EsN0_S_40dB[2]
test_data_EsN0_40dB = EsN0_S_40dB[3]
```

```
## Creating datasets for training
train_input_F_H = np.concatenate((train_input_F_H_0dB,
    train_input_F_H_10dB, train_input_F_H_20dB, train_input_F_H_30dB,
    train_input_F_H_40dB,), axis=0)

train_input_EsN0 = np.concatenate((train_input_EsN0_0dB,
    train_input_EsN0_10dB, train_input_EsN0_20dB,
    train_input_EsN0_30dB, train_input_EsN0_40dB), axis=0)

train_input_A_inv = np.concatenate((train_input_A_inv_0dB,
    train_input_A_inv_10dB, train_input_A_inv_20dB,
    train_input_A_inv_30dB, train_input_A_inv_40dB), axis=0)

train_input_X = np.concatenate((train_input_X_0dB,
    train_input_X_10dB, train_input_X_20dB, train_input_X_30dB,
    train_input_X_40dB), axis=0)

train_input_beta = np.concatenate((train_input_beta_0dB,
    train_input_beta_10dB, train_input_beta_20dB,
    train_input_beta_30dB, train_input_beta_40dB), axis=0)

train_input_p_hat = np.concatenate((train_input_p_hat_0dB,
```

```
    train_input_p_hat_10dB, train_input_p_hat_20dB,
    train_input_p_hat_30dB, train_input_p_hat_40dB), axis=0)

print(train_input_F_H.shape)
print(train_input_EsN0.shape)
print(train_input_A_inv.shape)
print(train_input_X.shape)
print(train_input_p_hat.shape)
```

```
## Creating datasets for validation
valid_input_F_H = np.concatenate((valid_input_F_H_0dB,
    valid_input_F_H_10dB, valid_input_F_H_20dB,
    valid_input_F_H_30dB, valid_input_F_H_40dB,), axis=0)

valid_input_EsN0 = np.concatenate((valid_input_EsN0_0dB,
    valid_input_EsN0_10dB, valid_input_EsN0_20dB,
    valid_input_EsN0_30dB, valid_input_EsN0_40dB), axis=0)

valid_input_A_inv = np.concatenate((valid_input_A_inv_0dB,
    valid_input_A_inv_10dB, valid_input_A_inv_20dB,
    valid_input_A_inv_30dB, valid_input_A_inv_40dB), axis=0)

valid_input_X = np.concatenate((valid_input_X_0dB,
    valid_input_X_10dB, valid_input_X_20dB,
    valid_input_X_30dB, valid_input_X_40dB), axis=0)

valid_input_beta = np.concatenate((valid_input_beta_0dB,
    valid_input_beta_10dB, valid_input_beta_20dB,
    valid_input_beta_30dB, valid_input_beta_40dB), axis=0)

valid_input_p_hat = np.concatenate((valid_input_p_hat_0dB,
    valid_input_p_hat_10dB, valid_input_p_hat_20dB,
    valid_input_p_hat_30dB, valid_input_p_hat_40dB), axis=0)

print(valid_input_F_H.shape)
print(valid_input_EsN0.shape)
print(valid_input_A_inv.shape)
print(valid_input_X.shape)
print(valid_input_p_hat.shape)
```

```
## Shuffling the training datasets
train_shuffler = np.random.permutation(len(train_input_F_H))
train_input_F_H_shuffled = train_input_F_H[train_shuffler]
train_input_EsN0_shuffled = train_input_EsN0[train_shuffler]
train_input_A_inv_shuffled = train_input_A_inv[train_shuffler]
train_input_X_shuffled = train_input_X[train_shuffler]
train_input_beta_shuffled = train_input_beta[train_shuffler]
train_input_p_hat_shuffled = train_input_p_hat[train_shuffler]
```

```
## Shuffling the validation datasets
valid_shuffler = np.random.permutation(len(valid_input_F_H))
valid_input_F_H_shuffled = valid_input_F_H[valid_shuffler]
valid_input_EsN0_shuffled = valid_input_EsN0[valid_shuffler]
valid_input_A_inv_shuffled = valid_input_A_inv[valid_shuffler]
valid_input_X_shuffled = valid_input_X[valid_shuffler]
valid_input_beta_shuffled = valid_input_beta[valid_shuffler]
valid_input_p_hat_shuffled = valid_input_p_hat[valid_shuffler]
```

```
## Reshaping train_input_F_H_shuffled and adding
   train_input_EsN0_shuffled
const = K*K
len1 = train_input_F_H_shuffled.shape[0]
train_input_F_H_shuffled_reshaped = train_input_F_H_shuffled.reshape
   ((len1, 1, const)) # size X row X column
train_y_true = np.concatenate((train_input_F_H_shuffled_reshaped,
   train_input_EsN0_shuffled), axis=2)
print(train_y_true.shape)
```

```
## Reshaping train_input_F_H_shuffled and adding
   train_input_EsN0_shuffled
len2 = valid_input_F_H_shuffled.shape[0]
valid_input_F_H_shuffled_reshaped = valid_input_F_H_shuffled.reshape
   ((len2, 1, const)) # size X row X column
valid_y_true = np.concatenate((valid_input_F_H_shuffled_reshaped,
   valid_input_EsN0_shuffled), axis=2)
print(valid_y_true.shape)
```

```
## Define the DNN model - The Functional API
import tensorflow as tf
from tensorflow import keras
## from tensorflow.keras import layers # shows warning
```

```
from keras.api._v2.keras import layers
from keras.layers import Input, concatenate, Lambda
from keras.models import Model


hij_inputs = keras.Input(shape=(K,K), name = "hij_inputs")
f1 = layers.Flatten(name = "flatten_layer_hij")(hij_inputs)

EsN0_inputs = keras.Input(shape=(1,1), name = "EsN0_inputs")
f2 = layers.Flatten(name = "flatten_layer_EsN0")(EsN0_inputs)

concat_layers = concatenate([f1, f2])

d1 = layers.Dense(2*K*K, activation="relu", name = "dense_layer_1")(
    concat_layers)
b1 = layers.BatchNormalization(name = "batch_norm_layer_1")(d1)

d2 = layers.Dense(K*K, activation="relu", name = "dense_layer_2")(b1)
b2 = layers.BatchNormalization(name = "batch_norm_layer_2")(d2)

# meu = layers.Dense(K, activation="relu", name = "meu")(b2)
meu = layers.Dense(K, activation="sigmoid", name = "meu")(b2)

A_inv_inputs = keras.Input(shape=(K,K), name = "A_inv_inputs")
f3 = layers.Flatten(name = "flatten_layer_A_inv")(A_inv_inputs)

X_inputs = keras.Input(shape=(K,1), name = "X_inputs")
f4 = layers.Flatten(name = "flatten_layer_X")(X_inputs)

beta_inputs = keras.Input(shape=(K,1), name = "beta_inputs")
f5 = layers.Flatten(name = "flatten_layer_beta")(beta_inputs)

p_hat_inputs = keras.Input(shape=(K,1), name = "p_hat_inputs")
f6 = layers.Flatten(name = "flatten_layer_p_hat")(p_hat_inputs)

def custom_layer(tensor):
  t_A_inv = tensor[0]
  t_X = tensor[1]
  t_beta = tensor[2]
```

```
  t_p_hat = tensor[3]
  t_meu = tensor[4]

  A_inv_cl = tf.reshape(t_A_inv[:,0:K*K], (-1,K,K))
  X_cl = tf.reshape(t_X[:,0:K*1], (-1,K,1))
  beta_cl = tf.reshape(t_beta[:,0:K*1], (-1,K,1))
  p_hat_cl = tf.reshape(t_p_hat[:,0:K*1], (-1,K,1))
  meu_cl = tf.reshape(t_meu[:,0:K*1], (-1,K,1))

  meu_ewm = tf.math.multiply(beta_cl, meu_cl)

  alpha_dnumr = tf.matmul(A_inv_cl, meu_ewm)
  alpha_whole = tf.divide(X_cl, alpha_dnumr)
  alpha = tf.reduce_min(alpha_whole, axis = 1, keepdims = True)
  max_p = tf.constant([1.0])
  alpha = tf.math.minimum(max_p, alpha)
  meu_P = tf.multiply(meu_ewm, alpha)

  Z_cl = tf.matmul(A_inv_cl, meu_P)
  P_hat_cl = tf.add(p_hat_cl, Z_cl)
  P_hat_cl_Norm = tf.math.divide(P_hat_cl, tf.reduce_max(P_hat_cl,
    axis = 1, keepdims = True))

  # return P_hat_cl
  return P_hat_cl_Norm

lambda_layer = tf.keras.layers.Lambda(custom_layer, name="
    lambda_layer")([f3, f4, f5, f6, meu])
f7 = layers.Flatten(name = "flatten_layer_output")(lambda_layer)

model = keras.Model(inputs = [hij_inputs, EsN0_inputs, A_inv_inputs,
    X_inputs, beta_inputs, p_hat_inputs], outputs = f7, name = "
    functional_api")
model.summary()
```

```
## Plot the model as a graph
keras.utils.plot_model(model, "Functional_API_Model.png")
```

```
## Display the input and output shapes of each layer
keras.utils.plot_model(model, "Functional_API_Model_with_shape_info.
```

```
    png", show_shapes=True)
```

```
## The customized loss function

def custom_loss(y_true, y_pred):
  # p = y_pred
  p = tf.math.multiply(p_max, y_pred)

  mtrx_elmnt = K*K
  EsN0_val = y_true[0][0][mtrx_elmnt]
  y_true_updt = y_true[:,:,:-1]

  if EsN0_val < 10:
    sigma_sqr_noise_lf = 1e-0
  elif EsN0_val >= 10 and EsN0_val < 20:
    sigma_sqr_noise_lf = 1e-1
  elif EsN0_val >= 20 and EsN0_val < 30:
    sigma_sqr_noise_lf = 1e-2
  elif EsN0_val >= 30 and EsN0_val < 40:
    sigma_sqr_noise_lf = 1e-3
  else:
    sigma_sqr_noise_lf = 1e-4

  hij = tf.reshape(y_true_updt[:,0:K*K], (-1,K,K))
  hij_abs_sqr = tf.math.square(tf.math.abs(hij))

  R_P = 0.0
  for i in range(K):  # Total rows
    ph = 0.0
    for j in range(K):  # Total columns
      ph_j = tf.math.multiply(p[:,j], hij_abs_sqr[:,i,j])
      ph = tf.math.add(ph, ph_j)

    numr = tf.math.multiply(p[:,i], hij_abs_sqr[:,i,i])
    dnumr = tf.math.add(sigma_sqr_noise_lf, tf.math.subtract(ph, numr
   ))
    SINR_i = tf.math.divide(numr, dnumr)
    R_P = tf.math.add(R_P, (tf.math.log(1 + SINR_i)/tf.math.log(2.0))
   )
```

```
36
37   loss = -R_P
38   loss = tf.reduce_mean(loss) # batch mean
39   return loss
```

```
1 ## Build and compile the DNN model
2 ## Training and Testing
3 import matplotlib.pyplot as plt
4
5 optA = tf.keras.optimizers.Adam(learning_rate = 0.0001)
6 # optA = tf.keras.optimizers.Adam(learning_rate = 0.0001, clipnorm
    =0.92)
7 model.compile(optimizer = optA, loss = custom_loss)
8
9 train_input = [train_input_F_H_shuffled, train_input_EsN0_shuffled,
    train_input_A_inv_shuffled,
10                train_input_X_shuffled, train_input_beta_shuffled,
    train_input_p_hat_shuffled]
11
12 valid_input = [valid_input_F_H_shuffled, valid_input_EsN0_shuffled,
    valid_input_A_inv_shuffled,
13                valid_input_X_shuffled, valid_input_beta_shuffled,
    valid_input_p_hat_shuffled]
14
15 history = model.fit(train_input, train_y_true, epochs = 50,
16                     validation_data = (valid_input, valid_y_true),
    batch_size = 1000)
17
18 plt.plot(history.epoch, history.history['loss'], color = "blue",
    label = "Training")
19 plt.plot(history.epoch, history.history['val_loss'], color="black",
    label = "Validation")
20 plt.xlabel("epochs")
21 plt.ylabel("loss")
22 plt.legend()
23 plt.show()
```

```
1 ## Constraint violation probability and
2 ## finding indexes of test_input_F_H matrix with the hij set that do
    not satisfy
```

```
## constraint on the minimum SINR_P_min rate but satisfy the maximum transmit
## power p_max

test_input = [test_input_F_H_0dB, test_input_EsN0_0dB, test_input_A_inv_0dB,
              test_input_X_0dB, test_input_beta_0dB, test_input_p_hat_0dB]
# output_P_hat_temp = model.predict(test_input)
output_P_hat_temp = np.multiply(p_max, model.predict(test_input))
output_P_hat = output_P_hat_temp.reshape((output_P_hat_temp.shape[0], output_P_hat_temp.shape[1], 1)) # test_input_F_H_size X row X column
output_P_hat_size = output_P_hat.shape[0]
test_data_F_H_abs_sqr = cmplx_abs_sqr(test_data_F_H_0dB)

indx_n = []
count_v = 0

for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(output_P_hat[k,j], test_data_F_H_abs_sqr[k,i,j])
      ph = ph + ph_j

    numr = np.multiply(output_P_hat[k,i], test_data_F_H_abs_sqr[k,i,i])
    dnumr = sigma_sqr_noise_0dB[i] + ph - numr
    SINR_out = np.divide(numr, dnumr)

    if np.round(SINR_out, decimals= 3) < SINR_P_min[i]:
      indx_n.append(k)
      count_v = count_v + 1
      # print(SINR_out)
      break

violation_prb = (count_v / output_P_hat_size) * 100
```

```
print("Constraints Violation Probability: {:.2f}%".format(
    violation_prb))
# print(len(indx_n))
# print(indx_n)
```

```
## Function to calculate the average sum rate
# Here, p_model is the output of DNN, and it is a 2D array.
import math

def average_sum_rate(hij, p_model, sigma_sqr_noise, K):
  R = 0
  hij_size = hij.shape[0]
  hij_abs_sqr = cmplx_abs_sqr(hij)

  for k in range(hij_size):
    for i in range(K):  # Total rows
      phn = 0
      for j in range(K):  # Total columns
        phn_j = np.multiply(p_model[k,j], hij_abs_sqr[k,i,j])
        phn = phn + phn_j

      numr_s = np.multiply(p_model[k,i], hij_abs_sqr[k,i,i])
      dnumr_s = sigma_sqr_noise[i] + phn - numr_s
      R_temp = math.log2(1 + np.divide(numr_s, dnumr_s))
      R = R + R_temp

  return (R/hij_size)
```

```
## DNN Sum Rate for test_data_F_H
sumrate_F_H = average_sum_rate(test_data_F_H_0dB, output_P_hat,
    sigma_sqr_noise_0dB, K)
print("Average Sum Rate for all H matrices: {:.3f} Bit/Second/Hertz".
    format(sumrate_F_H))
```

```
## Checking (A_inv x b), i.e., the power for negative values
count_n = 0
for c in range(output_P_hat_size):
  p_temp = np.matmul(test_input_A_inv_0dB[c], test_input_b_0dB[c])
  if np.any(p_temp < 0):
    count_n = count_n + 1
```

```
    print(c,'\n')
    print(p_temp)

print(count_n)
```

```
## Checking P_hat, i.e., the power for test_data_F_H for negative
   values
## and Hit Rate i.e. percentage for 0 <= P_hat <= p_max
count_p = 0
count_n = 0

for n in range(output_P_hat_size):
  P_max = np.amax(output_P_hat[n])
  if np.round(P_max, decimals = 3) <= 1:
    count_p = count_p + 1

  if np.any(output_P_hat[n] < 0):
    count_n = count_n + 1
    print(n,'\n')
    print(output_P_hat)

p_hit_rate = (count_p / output_P_hat_size) * 100
print("Htt Rate for Power : {:.2f}%".format(p_hit_rate))
print("Negative power count: ", count_n)
```

C.4 Codes for analyzing the Model A

```
import numpy as np

## Number of transmitter-receiver pairs
K = 5

## Variances for noise signals
sigma_sqr_noise = np.array([1e-0, 1e-0, 1e-0, 1e-0, 1e-0], dtype =
    float)

## Minimum rate for the achievable SINR of multiple concurrent
## transmissions
SINR_P_min = np.array([0.5, 0.5, 0.5, 0.5, 0.5])

## Maximum transmit power
p_max = 1.0
```

```
## Loading a NumPy array from a CSV file
# Loading F_H array from a CSV file
from numpy import loadtxt

## Reading an array from the file
# If we want to read a file from our local drive, we have to first
# upload it to Collab's session storage.
F_H_2D_L = np.loadtxt('F_H_2D.csv', delimiter = ',', dtype = str)

## Reshaping the array from 2D to 3D
F_H_3D = F_H_2D_L.reshape(F_H_2D_L.shape[0], F_H_2D_L.shape[1] // K,
    K)
F_H_3D_size = F_H_3D.shape[0]
```

```
## Converting string data to complex data and removing the initial
## whitespace
F_H_list = []
for k in range(F_H_3D_size):
  for i in range(K):  # Total rows
    for j in range(K):  # Total columns
      F_H_temp = complex(F_H_3D[k][i][j].strip())
```

```
      F_H_list.append(F_H_temp)
F_H_array = np.array(F_H_list)
F_H = F_H_array.reshape((F_H_3D_size, K, K)) # H_size X row X
    column_count
print(F_H.shape)
F_H_size = F_H.shape[0]
# print(F_H)
```

```
import numba as nb

## Function to compute the square of the absolute value of an array
## of complex numbers
@nb.vectorize([nb.float64(nb.complex128),nb.float32(nb.complex64)])
def cmplx_abs_sqr(cmplx_var):
  return cmplx_var.real**2 + cmplx_var.imag**2
```

```
## Function to generate the matrix A (K x K)
def generate_A(F_H_size, K, SINR_P_min, F_H):
  Aij_list = []
  F_H_abs_sqr = cmplx_abs_sqr(F_H)

  for k in range(F_H_size):
    for i in range(K):  # Total rows
      Aj_list =[]
      for j in range(K): # Total columns
        if i==j:
          A = F_H_abs_sqr[k,i,j]
        else:
          A = np.multiply(-SINR_P_min[i], F_H_abs_sqr[k,i,j])
        Aj_list.append(A)
      Aij_list.append(Aj_list)
  Aij_array = np.array(Aij_list)
  Aij = Aij_array.reshape((F_H_size, K, K)) # H_size X row X column
  return Aij
```

```
## Create matrix A
A = generate_A(F_H_size, K, SINR_P_min, F_H)
print(A.shape)
# print(A)
```

```
## Function to generate the vector b (K x 1)
def generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H):
  bi_list = []
  for k in range(F_H_size):
    for i in range(K):  # Total rows, i.e., total transmitters
      b = np.multiply(SINR_P_min[i], sigma_sqr_noise[i])
      bi_list.append(b)
  bi_array = np.array(bi_list)
  bi = bi_array.reshape((F_H_size, K, 1)) # H_size X row X column
  return bi
```

```
## Create vector b
b = generate_b(F_H_size, K, SINR_P_min, sigma_sqr_noise, F_H)
print(b.shape)
# print(b)
```

```
## Create matrix A_inv, i.e., the pseudo inverse of matrix A
A_inv = np.linalg.pinv(A)
A_inv[A_inv<0] = 0
print(A_inv.shape)
# print(A_inv)
```

```
## Create a vector p_hat = (A_inv x b)
p_hat = np.matmul(A_inv, b)
print(p_hat.shape)
# print(p_hat)
```

```
## Convert p_max_array to (K x 1) vector
p_max_array = np.array([1.0, 1.0, 1.0, 1.0, 1.0], dtype = float)
p_max_vector = p_max_array.reshape((K, 1)) # row X column
print(p_max_vector)
```

```
## Create a vector X = (p_max_vector - p_hat)
X = p_max_vector - p_hat
print(X.shape)
# print(X)
```

```
## Function to split datasets for training, validation, and testing.
def split(np_array):
  # data_size = np_array.shape[0]
  # train_data_size = int(data_size * 0.8)
```

```
# valid_data_size = int(data_size * 0.1)
# test_data_size = int(data_size * 0.1)

train_data_size = int(200000)
valid_data_size = int(25000)
test_data_size = int(25000)

train_e_indx = train_data_size
valid_e_indx = train_e_indx + valid_data_size
test_e_indx = valid_e_indx + test_data_size
test_data_size_n = test_e_indx - valid_e_indx

row_count = np_array.shape[1]
column_count = np_array.shape[2]

train_data = np.empty((train_data_size, row_count, column_count),
 dtype = complex, order = 'C')
valid_data = np.empty((valid_data_size, row_count, column_count),
 dtype = complex, order = 'C')
test_data = np.empty((test_data_size_n, row_count, column_count),
 dtype = complex, order = 'C')

for i in range(train_e_indx):
  train_data[i] = np_array[i]

xv = 0
for j in range(train_e_indx, valid_e_indx):
  valid_data[xv] = np_array[j]
  xv = xv + 1

xt = 0
for k in range(valid_e_indx, test_e_indx):
  test_data[xt] = np_array[k]
  xt = xt + 1

# print(train_data.shape, valid_data.shape, test_data.shape)


## Training input will be the absolute value
```

```
  train_input = np.absolute(train_data)
  valid_input = np.absolute(valid_data)
  test_input = np.absolute(test_data)

  print(train_input.shape, valid_input.shape, test_input.shape)

  return [train_input, valid_input, test_input, test_data]
```

```
## Split F_H matrix
F_H_S = split(F_H)
train_input_F_H = F_H_S[0]
valid_input_F_H = F_H_S[1]
test_input_F_H = F_H_S[2]
test_data_F_H = F_H_S[3]
```

```
## Split A_inv matrix
A_inv_S = split(A_inv)
train_input_A_inv = A_inv_S[0]
valid_input_A_inv = A_inv_S[1]
test_input_A_inv = A_inv_S[2]
test_data_A_inv = A_inv_S[3]
```

```
## Split b vector
b_S = split(b)
train_input_b = b_S[0]
valid_input_b = b_S[1]
test_input_b = b_S[2]
test_data_b = b_S[3]
```

```
## Split X vector
X_S = split(X)
train_input_X = X_S[0]
valid_input_X = X_S[1]
test_input_X = X_S[2]
test_data_X = X_S[3]
```

```
## Split p_hat vector
p_hat_S = split(p_hat)
train_input_p_hat = p_hat_S[0]
valid_input_p_hat = p_hat_S[1]
test_input_p_hat = p_hat_S[2]
```

```
test_data_p_hat = p_hat_S[3]
```

```
## Define the DNN model - The Functional API
import tensorflow as tf
from tensorflow import keras
## from tensorflow.keras import layers # shows warning
from keras.api._v2.keras import layers
from keras.layers import Input, Lambda
from keras.models import Model


hij_inputs = keras.Input(shape=(K,K), name = "hij_inputs")
f1 = layers.Flatten(name = "flatten_layer_hij")(hij_inputs)

d1 = layers.Dense(2*K*K, activation="relu", name = "dense_layer_1")(
    f1)
b1 = layers.BatchNormalization(name = "batch_norm_layer_1")(d1)

d2 = layers.Dense(K*K, activation="relu", name = "dense_layer_2")(b1)
b2 = layers.BatchNormalization(name = "batch_norm_layer_2")(d2)

# meu = layers.Dense(K, activation="relu", name = "meu")(b2)
meu = layers.Dense(K, activation="sigmoid", name = "meu")(b2)
# def meu_layer(tensor_meu):
#   tf.print("\nmeu output:\n", tensor_meu)
#   return tensor_meu
# meu_layer = tf.keras.layers.Lambda(meu_layer, name="meu_layer")(meu
    )

A_inv_inputs = keras.Input(shape=(K,K), name = "A_inv_inputs")
f2 = layers.Flatten(name = "flatten_layer_A_inv")(A_inv_inputs)

X_inputs = keras.Input(shape=(K,1), name = "X_inputs")
f3 = layers.Flatten(name = "flatten_layer_X")(X_inputs)

p_hat_inputs = keras.Input(shape=(K,1), name = "p_hat_inputs")
f4 = layers.Flatten(name = "flatten_layer_p_hat")(p_hat_inputs)

def custom_layer(tensor):
```

```
  t_A_inv = tensor[0]
  t_X = tensor[1]
  t_p_hat = tensor[2]
  t_meu = tensor[3]

  A_inv_cl = tf.reshape(t_A_inv[:,0:K*K], (-1,K,K))
  X_cl = tf.reshape(t_X[:,0:K*1], (-1,K,1))
  p_hat_cl = tf.reshape(t_p_hat[:,0:K*1], (-1,K,1))
  meu_cl = tf.reshape(t_meu[:,0:K*1], (-1,K,1))

  alpha_dnumr = tf.matmul(A_inv_cl, meu_cl)
  alpha_whole = tf.divide(X_cl, alpha_dnumr)
  alpha = tf.reduce_min(alpha_whole, axis = 1, keepdims = True)
  max_p = tf.constant([1.0])
  alpha = tf.math.minimum(max_p, alpha)
  meu_P = tf.multiply(meu_cl, alpha)

  Z_cl = tf.matmul(A_inv_cl, meu_P)
  P_hat_cl = tf.add(p_hat_cl, Z_cl)
  P_hat_cl_Norm = tf.math.divide(P_hat_cl, tf.reduce_max(P_hat_cl,
    axis = 1, keepdims = True))

  # return P_hat_cl
  return P_hat_cl_Norm

lambda_layer = tf.keras.layers.Lambda(custom_layer, name="
    lambda_layer")([f2, f3, f4, meu])
f5 = layers.Flatten(name = "flatten_layer_output")(lambda_layer)

model = keras.Model(inputs = [hij_inputs, A_inv_inputs, X_inputs,
    p_hat_inputs], outputs = f5, name = "functional_api")
model.summary()
```

```
## Plot the model as a graph
keras.utils.plot_model(model, "Functional_API_Model.png")
```

```
## Display the input and output shapes of each layer
keras.utils.plot_model(model, "Functional_API_Model_with_shape_info.
    png", show_shapes=True)
```

```
## Convert sigma_sqr_noise from numpy array to tensor
sigma_sqr_noise_t = tf.convert_to_tensor(sigma_sqr_noise, dtype =
    float)
tf.print(sigma_sqr_noise_t)
```

```
## The customized loss function

def custom_loss(y_true, y_pred):
  # p = y_pred
  p = tf.math.multiply(p_max, y_pred)
  hij = tf.reshape(y_true[:,0:K*K], (-1,K,K))
  hij_abs_sqr = tf.math.square(tf.math.abs(hij))

  R_P = 0.0
  for i in range(K):  # Total rows
    ph = 0.0
    for j in range(K):  # Total columns
      ph_j = tf.math.multiply(p[:,j], hij_abs_sqr[:,i,j])
      ph = tf.math.add(ph, ph_j)

    numr = tf.math.multiply(p[:,i], hij_abs_sqr[:,i,i])
    dnumr = tf.math.add(sigma_sqr_noise_t[i], tf.math.subtract(ph,
    numr))
    SINR_i = tf.math.divide(numr, dnumr)
    R_P = tf.math.add(R_P, (tf.math.log(1 + SINR_i)/tf.math.log(2.0))
    )

  loss = -R_P
  loss = tf.reduce_mean(loss) # batch mean
  return loss
```

```
## Build and compile the DNN model
## Training and Testing
import matplotlib.pyplot as plt

optA = tf.keras.optimizers.Adam(learning_rate = 0.0001)
model.compile(optimizer = optA, loss = custom_loss)

train_input = [train_input_F_H, train_input_A_inv, train_input_X,
    train_input_p_hat]
```

```
valid_input = [valid_input_F_H, valid_input_A_inv, valid_input_X,
    valid_input_p_hat]

history = model.fit(train_input, train_input_F_H, epochs = 50,
    validation_data = (valid_input, valid_input_F_H), batch_size =
    1000)

plt.plot(history.epoch, history.history['loss'], color = "blue",
    label = "Training")
plt.plot(history.epoch, history.history['val_loss'], color="black",
    label = "Validation")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend()
plt.show()
```

```
## Constraint violation probability and
## finding indexes of test_input_F_H matrix with the hij set that do
## not satisfy constraint on the minimum SINR_P_min rate but satisfy
## the maximum transmit power p_max

test_input = [test_input_F_H, test_input_A_inv, test_input_X,
    test_input_p_hat]
# output_P_hat_temp = model.predict(test_input)
output_P_hat_temp = np.multiply(p_max, model.predict(test_input))
output_P_hat = output_P_hat_temp.reshape((output_P_hat_temp.shape[0],
     output_P_hat_temp.shape[1], 1)) # test_input_F_H_size X row X
    column
output_P_hat_size = output_P_hat.shape[0]
test_data_F_H_abs_sqr = cmplx_abs_sqr(test_data_F_H)

indx_n = []
count_v = 0

for k in range(output_P_hat_size):
  for i in range(K):  # Total rows
    ph = 0
    for j in range(K):  # Total columns
      ph_j = np.multiply(output_P_hat[k,j], test_data_F_H_abs_sqr[k,i
```

```
    ,j])
21      ph = ph + ph_j
22
23     numr = np.multiply(output_P_hat[k,i], test_data_F_H_abs_sqr[k,i,i
    ])
24     dnumr = sigma_sqr_noise[i] + ph - numr
25     SINR_out = np.divide(numr, dnumr)
26
27     if np.round(SINR_out, decimals= 3) < SINR_P_min[i]:
28       indx_n.append(k)
29       count_v = count_v + 1
30       # print(SINR_out)
31       break
32
33 violation_prb = (count_v / output_P_hat_size) * 100
34 print("Constraints Violation Probability: {:.2f}%".format(
    violation_prb))
35 # print(len(indx_n))
36 # print(indx_n)
```

```
1  ## Function to calculate the average sum rate
2  # Here, p_model is the output of DNN, and it is a 2D array.
3  import math
4
5  def average_sum_rate(hij, p_model, sigma_sqr_noise, K):
6    R = 0
7    hij_size = hij.shape[0]
8    hij_abs_sqr = cmplx_abs_sqr(hij)
9
10   for k in range(hij_size):
11     for i in range(K):  # Total rows
12       phn = 0
13       for j in range(K):  # Total columns
14         phn_j = np.multiply(p_model[k,j], hij_abs_sqr[k,i,j])
15         phn = phn + phn_j
16
17       numr_s = np.multiply(p_model[k,i], hij_abs_sqr[k,i,i])
18       dnumr_s = sigma_sqr_noise[i] + phn - numr_s
19       R_temp = math.log2(1 + np.divide(numr_s, dnumr_s))
```

```
      R = R + R_temp

  return (R/hij_size)
```

```
# DNN Sum Rate for test_data_F_H
sumrate_F_H = average_sum_rate(test_data_F_H, output_P_hat,
    sigma_sqr_noise, K)
print("Average Sum Rate for all H matrices: {:.3f} Bit/Second/Hertz".
    format(sumrate_F_H))
```

```
## Checking (A_inv x b), i.e., the power for negative values
count_n = 0
for c in range(output_P_hat_size):
  p_temp = np.matmul(test_input_A_inv[c], test_input_b[c])
  if np.any(p_temp < 0):
    count_n = count_n + 1
    print(c,'\n')
    print(p_temp)

print(count_n)
```

```
## Checking P_hat, i.e., the power for test_data_F_H for negative
## values and Heat Rate i.e. percentage for 0 <= P_hat <= p_max
count_p = 0
count_n = 0

for n in range(output_P_hat_size):
  P_max = np.amax(output_P_hat[n])
  if np.round(P_max, decimals = 3) <= 1:
    count_p = count_p + 1

  if np.any(output_P_hat[n] < 0):
    count_n = count_n + 1
    print(n,'\n')
    print(output_P_hat)

p_heat_rate = (count_p / output_P_hat_size) * 100
print("Heat Rate for Power : {:.2f}%".format(p_heat_rate))
print("Negative power count: ", count_n)
```

C.4.1 Codes to calculate the average sum rate for the basic model

```
# DNN Sum Rate for test_data_F_H
output_P_hat = abs(test_data_p_hat)
sumrate_F_H = average_sum_rate(test_data_F_H, output_P_hat,
    sigma_sqr_noise, K)
print("Average Sum Rate for all H matrices: {:.3f} Bit/Second/Hertz".
    format(sumrate_F_H))
```

Special Terms

3G Third Generation of wireless mobile telecommunications technology. 11

4G Fourth Generation of wireless mobile telecommunications technology. 10, 11, 178

5G Fifth Generation of wireless mobile telecommunications technology. 1, 7–12, 16, 71

6G Sixth Generation of wireless mobile telecommunications technology. 11, 16

AI Artificial Intelligence. 23, 28

ASR Average Sum Rate. 42, 53, 64–67

BN Batch Normalization. 42

BS Base Station. viii, 1, 2, 8, 11, 15

CNN Convolutional Neural Network. 28, 29

Colab Google Colaboratory. 43, 67

CSCG Circularly Symmetric Complex Gaussian. 34, 40

CSI Channel State Information. 22, 31, 32

CU Cellular User. 18

CUE Cellular User Equipment. 29, 30

CVP Constraint Violation Probability. 42, 47, 79

D2D Device-to-Device. ii, 1–20, 23, 24, 26–32, 39, 40, 43, 68–70, 72

DC-DC D2D Communication with a Device-Controlled link establishment. 14

DC3 Deep Constraint Completion and Correction algorithm. 5, 32

DL Deep Learning. ii, 4, 27

DNN Deep Neural Network. ii, 27, 29–32, 35, 38, 39, 41, 42, 44, 46, 47, 53, 64, 67, 70

DPC Deep Power Control. 29

DR-DC Device Relaying with Device-Controlled link setup. 13

DR-OC Device Relaying with an Operator-Controlled link establishment. 12

DRL Deep Reinforcement Learning. 28, 30

DSL Digital Subscriber Line. 29

DUE Device-to-Device User Equipment. 29, 30

DUL Deep Unsupervised Learning. ii, 5–8, 30, 32, 33, 39, 42–44, 53, 64, 67–70, 72

EE Energy Efficiency. 29

eMBB extreme Mobile BroadBand. 9

eMTC enhanced Machine-Type Communication. 11

ePCNet ensemble Power Control Network. 31

GB GigaByte. 45, 78

GIC Gaussian Interference Channel. 21

gNB Next generation Node B. 11, 12

GPU Graphics Processing Unit. 4, 43

HR Hit Rate. 42, 47, 53, 64, 79

IoT The Internet of Things. 7, 9, 11, 14, 178

ITIS Information-Theoretic Independent Set. 18

ITLinQ Information-Theoretic Link scheduling. 18

LTE Long-Term Evolution is a 4G wireless standard. 10

MAPEL The Method for Assigning Priority Levels algorithm. 21

MARL Multi-Agent Reinforcement Learning. 26

MDP Markov Decision Processes. 4

MIMO Multiple-Input and Multiple-Output. 9

MISO Multiple-Input and Single-Output. 21

ML Machine Learning. 23

MScDS Master of Science in Data Science. iii

MSE Mean Squared Error. 24, 25

NB-IoT NarrowBand IoT. 11

NN Neural Network. 4, 5, 32, 44, 53, 64, 67

NP-hard Nondeterministic Polynomial time problem. 18, 30, 31

PAN Personal Area Network. 10

PCA Principal Component Analysis. 25

PCNet Power Control Network. 31, 44, 53, 64, 67, 69, 79

PHY/MAC Physical Layer or medium/ Medium Access Control. 10

QoS Quality of Service. 2, 16, 20, 64, 67

RAM Random-Access Memory. 43

RB Resource Block. 18

ReLU Rectified Linear Unit activation function. 38, 41

RL Reinforcement Learning. 4, 26, 27

SE Spectral Efficiency. 29

SIMO Single-Input and Multiple-Output. 21

SINR Signal-to-Interference plus Noise Ratio. x, 19, 21, 35, 40, 42, 45, 46, 53, 64

SIR Signal-to-Interference Ratio. 20

SISO Single-Input and Single-Output. 21

SL Supervised Learning. 4, 24

SP Signal Processing. 28, 29

TPU Tensor Processing Unit. 4, 43

UE User Equipment. 3

UL Unsupervised Learning. ii, 4, 25, 42

URLLC Ultra-Reliable Low-Latency Communication. 8

VANET Vehicular Ad hoc Network. 9

WMMSE Weighted Minimum Mean Square Error. 29–31

WSR Weighted Sum Rate. 21

WTM Weighted Throughput Maximization. 20, 21