REINFORCEMENT LEARNING FOR REAL-TIME OPTIMIZATION IN NB-IOT NETWORKS

A thesis submitted in partial fulfilment of the requirements for
the award of the degree of

B.Tech

in

Electronics and Communication Engineering

By

SUHAIL AHAMED HITHAYATHULLAH (108120128)

OM SANDEEP SALPEKAR (108120086)

ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

TIRUCHIRAPALLI-620015

MAY 2024

BONAFIDE CERTIFICATE

This is to certify that the project titled **REINFORCEMENT LEARNING FOR REAL-TIME OPTIMIZATION IN NB-IOT NETWORKS** is a bonafide record of the work done by

Suhail Ahamed Hithayathullah (108120128)

Om Sandeep Salpekar (108120086)

in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**, during the year 20.

Dr. E. S. Gopi
Guide

Head of the Department

Project Viva-voice held on ________________________

Internal Examiner

External Examiner

ABSTRACT

This work investigates real-time configuration selection for Narrowband Internet-of-Things (NB-IoT) systems using deep reinforcement learning techniques. NB-IoT offers scalable radio access for massive device deployments, but optimal configuration allocation for random access and data transmission remains a challenge. The problem revolves around dynamically determining the configuration that maximizes the long-term average number of served devices per Transmission Time Interval (TTI) without prior knowledge of traffic statistics.

Given the computational complexity of exhaustive search for the optimal configuration, we propose a Deep Q-Network (DQN) based approach for real-time selection. This initial solution focuses on a single-parameter, single-group scenario. To address more complex multi-parameter and multi-group situations, the work explores advancements on DQN. This includes the Double DQN algorithm to improve training stability and Cooperative Multi-Agent Learning (CMA-DQN) for coordinated learning across multiple groups. The project then compares and evaluates the performance of these techniques.

Keywords: Reinforcement Learning; Q-Learning; Narrowband IoT; Multi-Agent Learning.

ACKNOWLEDGEMENTS

We humbly express our profound gratitude to the following esteemed individuals whose invaluable support and sagacious guidance have propelled us towards the culmination of this scholarly undertaking:

Foremost, we extend our heartfelt appreciation to the distinguished **Dr. E. S. Gopi**, our sagacious project guide, whose indefatigable support, perspicacious insights, and expert mentorship were indispensable to the successful completion of this arduous academic venture.

We also extend our deepest gratitude to the venerable **Dr. M.Bhaskar**, the esteemed Head of the Department of Electronics and Communication Engineering, whose unerring vision and generous resources facilitated our scholarly pursuits and enabled us to achieve excellence in our work.

Finally, we are indebted to our dear families and cherished friends, whose unwavering love, encouragement, and unwavering support sustained us through the rigors of this academic journey and inspired us to reach our full potential.

We are profoundly grateful to all these illustrious individuals whose generosity, wisdom, and guidance have immeasurably enriched our academic and personal lives.

TABLE OF CONTENTS

Title		Page No.
ABSTRACT		i
ACKNOWLEDGEMENTS		ii
TABLE OF CONTENTS		iii
LIST OF TABLES		iv
LIST OF FIGURES		v
CHAPTER 1	**INTRODUCTION**	
1.1	Literature Overview	1
1.2	Contributions	3
CHAPTER 2	**PROBLEM FORMULATION**	
2.1	System Model	4
2.2	Proposed Methodology	5
CHAPTER 3	**REINFORCEMENT LEARNING**	
3.1	Introduction	7
3.2	Learning and Policy	9
3.3	Q-Values	11
3.4	Q-Learning	12
3.5	Neural Networks	13
3.6	Deep Q-Networks	17
3.7	Multi-Agent Learning	21
CHAPTER 4	**EXPERIMENTS AND RESULTS**	
4.1	Simulation Setup	24
4.2	Results and Discussion	26
CHAPTER 5	**CONCLUSION**	
5.1	Conclusions	28

5.2 Scope for Future Work 28

REFERENCES

LIST OF TABLES

S. NO	CAPTION	PAGE NO.
1	Single Group Simulation Parameters	24
2	Multiple Group Simulation Parameters	25

LIST OF FIGURES

S. NO	CAPTION	PAGE NO.
1	Illustration of Network Setup	4
2	Uplink Channel Frame Structure	4
3	Process undergone by a Device	5
4	Illustration of Reinforcement Learning	7
5	Computing an action’s return: the sum of discounted future rewards	9
6	Training in Deep Q-Learning without target network	17
7	Training in Deep Q-Learning with target network	20
8	CMA-DQN with Value Decomposition Network	23
9	Observations of (a) DQN, (b) Double DQN, (c) CMA-DQN in blue and number of new devices in red	26

I. INTRODUCTION

1.1 LITERATURE OVERVIEW

Within the domain of machine-to-machine communication, Narrowband Internet of Things (NB-IoT) technology has emerged as a prominent facilitator for the interconnection of a vast array of low-power devices. Leveraging the established foundation of Long-Term Evolution (LTE) design principles, NB-IoT operates within a deliberately restricted bandwidth of 180 kHz. This strategic selection prioritizes Coverage Enhancement (CE)**,** particularly in geographically dispersed or signal-attenuated environments.

In stark contrast to its LTE predecessor, NB-IoT adopts a streamlined approach, employing only two uplink physical channel resources to manage all upstream transmissions. The first channel, designated as the Random-Access Channel (RACH), utilizes the Narrowband Physical Random-Access Channel (NPRACH) for the transmission of RACH preambles. This functionality establishes the initial handshake between devices and the network, akin to a formal request for connection. The second channel, the Narrowband Physical Uplink Shared Channel (NPUSCH), serves for the transmission of control information and device data.

To accommodate the diverse traffic patterns and coverage requirements inherent on the Internet of Things (IoT) landscape, NB-IoT supports up to three distinct CE groups of IoT devices. These groups share the uplink resource within the same frequency band. Each group caters to devices with specific coverage needs, differentiated using a dedicated broadcast signal transmitted by the evolved Node B (eNB). This broadcast signal, known as the Narrowband Physical Broadcast Channel (NPBCH), acts as a beacon, guiding devices within each group.

At the outset of each uplink Transmission Time Interval (TTI), the eNB undertakes a critical system configuration selection process. This selection determines the allocation of radio resources amongst the various CE groups. The objective is to achieve an optimal balance between accommodating the RACH procedure, which facilitates new device access, and reserving sufficient resources for data transmission. Striking this balance is paramount to ensuring a high volume of successful accesses and transmissions within the NB-IoT network. Allocating an excessive number of resources to the RACH procedure, while potentially enhancing access performance, can lead to a scarcity of resources for data transmission. Conversely, an inadequate allocation of RACH resources may result in extended access delays and an over-allocation of uplink data resources.

In essence, NB-IoT technology presents a meticulously crafted solution for low-power device communication within the IoT domain. Through its emphasis on coverage enhancement, streamlined channel utilization, and dynamic resource allocation strategies, NB-IoT strives to establish a robust and efficient network infrastructure for the ever-expanding universe of interconnected devices.

Unfortunately, dynamically optimizing the allocation of resources between RACH (new connection requests) and data transmission in NB-IoT remains an unsolved problem. The network typically observes the success rates of both RACH preambles (initial connection requests) and data transmissions from all groups at the end of each time interval. However, this historical information alone is insufficient to accurately predict future traffic patterns from all groups and optimize resource allocation for upcoming intervals.

Even with complete knowledge of past reception statistics, solving this problem precisely would translate to a Partially Observable Markov Decision Process (POMDP) with a vast state and action space. Solving such a complex problem would be computationally intractable.

Despite the lack of a perfect solution, NB-IoT continues to be a well-designed solution for low-power device communication. With its focus on improved coverage, streamlined data channels, and ongoing research on resource allocation optimization, NB-IoT strives to establish a robust and efficient network infrastructure for the ever-expanding world of interconnected devices.

1.2 CONTRIBUTIONS

We've devised Reinforcement Learning (RL) techniques for optimizing the allocation of uplink resources in NB-IoT networks, dynamically adjusting to accommodate varying numbers of IoT devices being served. We evaluate the effectiveness of these RL approaches by comparing them against each other. Our approach can be summarized as follows:

1. We've created an RL framework to adjust the allocation of uplink resources in NB-IoT networks, aiming to optimize the number of IoT devices served. This framework simulates IoT traffic, the Random-Access Channel (RACH) procedure, and the scheduling of uplink data transmission in NB-IoT. We use this simulated environment to train RL agents before deployment. These agents are continuously updated based on real traffic conditions in practical NB-IoT networks.
2. Initially, we examine a simplified NB-IoT scenario with a single parameter and a single CE (Coverage Enhancement) group. We develop a basic Deep Q-Network (DQN) to handle this scenario. We then enhance the DQN's capabilities by implementing a Double Deep Q-Network (Double DQN), demonstrating its effectiveness in managing high-dimensional state spaces.
3. Subsequently, we explore a more complex NB-IoT scenario involving multiple parameters and multiple CE groups. Directly applying DQN or Double DQN becomes impractical due to the increased complexity of parameter combinations. To address this challenge, we propose Cooperative Multi-Agent learning based on DQN (CMA-DQN). CMA-DQN breaks down the selection process for high-dimensional parameters into multiple parallel sub-tasks. It achieves this by employing several DQN agents trained cooperatively to handle each parameter for each CE group.

II. PROBLEM FORMULATION

2.1 SYSTEM MODEL

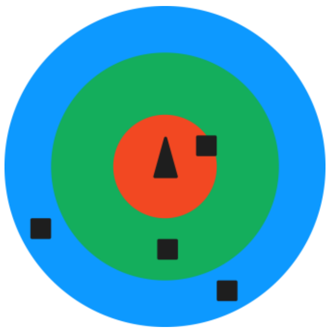


Fig. 1. Illustration of Network Setup

As illustrated in Fig. 1, we consider a single-cell NBIoT network composed of an eNB located at the centre of the cell (represented as a triangle), and a set of static IoT devices (represented as a square) randomly located in an area of the plane $\mathbf{R}^2$, and remain spatially static once deployed. The devices are divided into three CE groups as further discussed below, and the eNB is unaware of the status of these IoT devices, hence no uplink channel resource is scheduled to them in advance. In each IoT device, uplink data is generated according to random inter-arrival processes over the TTIs, which are Markovian and possibly time-varying.

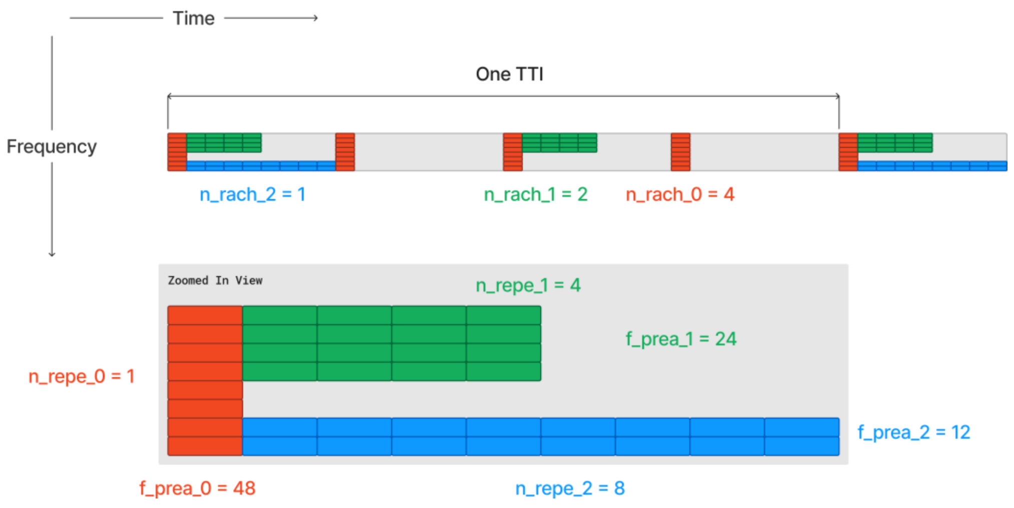


Fig. 2. Uplink Channel Frame Structure

As shown in Fig. 2, for each TTI t and for each CE group i = 0, 1, 2, in order to reduce the chance of a collision, the eNB can increase the number n_rach_i of RACH periods in the TTI or the number f_prea_i of preambles available in each RACH period. Furthermore, in order to increase the success rate, the eNB can increase the number n_repe_i of times that a preamble transmission is repeated by a device in group i in one RACH period of the TTI.

2.2 PROPOSED METHODOLOGY

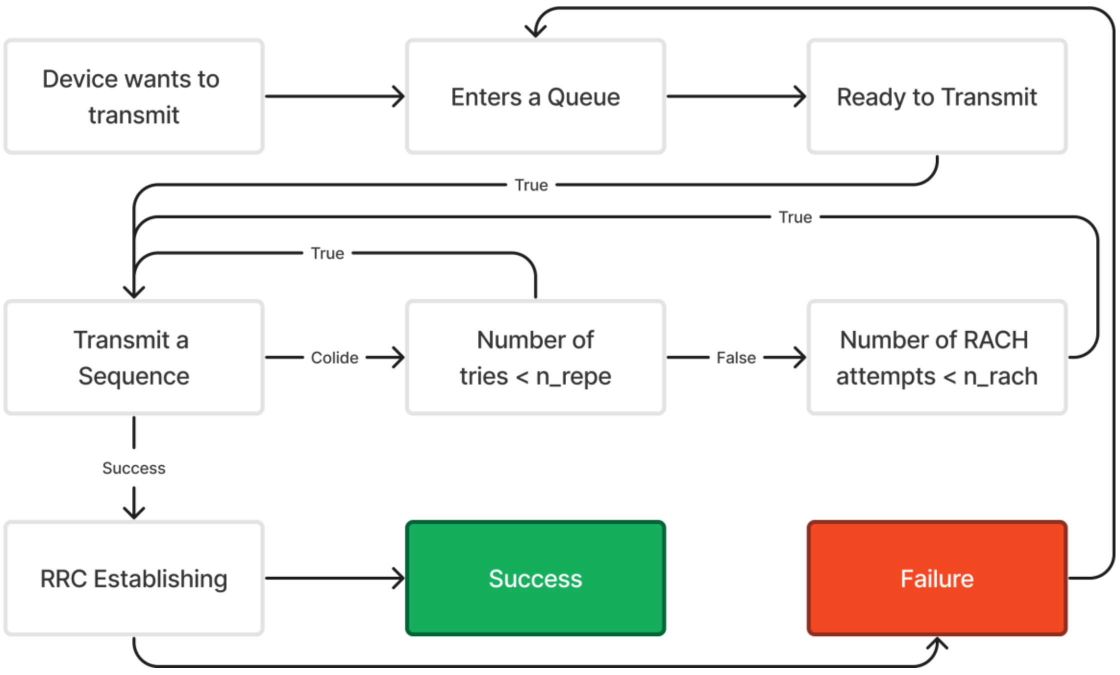


Fig. 3. Process undergone by a Device.

The flowchart above illustrates the formal procedure followed by a device when transmitting a data sequence. The device initially enters a queue, signifying its request to utilize a channel for transmission. Once positioned at the front of the queue, the device commences transmission. Following transmission, a collision check is performed to determine if concurrent transmissions from multiple devices corrupted the data. In the absence of a collision, transmission is successful. However, upon collision detection, the device verifies the number of attempted transmissions against a predefined maximum number of retries.

If attempts haven't reached the maximum threshold, the device re-attempts transmission. Conversely, when the maximum retry limit is surpassed, the flowchart investigates the number of Random-Access Channel (RACH) attempts. Similar to the previous step, the device is permitted a predefined number of RACH attempts to re-establish a Radio Resource Connection (RRC) essential for transmission. If RACH attempts exceed the allowed limit, the transmission process unfortunately fails. This flowchart meticulously outlines the device's data transmission endeavour, incorporating collision avoidance mechanisms and ensuring successful data delivery.

III. REINFORCEMENT LEARNING

3.1 INTRODUCTION

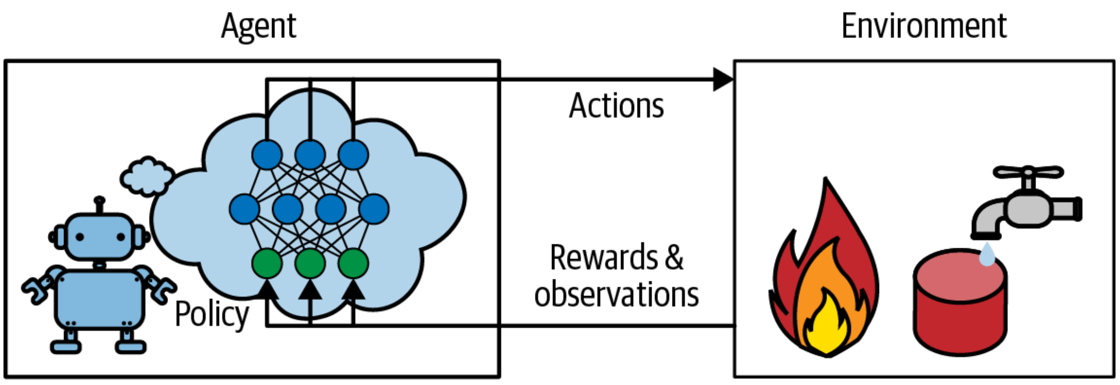


Fig. 4. Illustration of Reinforcement Learning [1]

Reinforcement learning (RL) is a type of machine learning technique where an agent learns to take actions in an environment to maximize a long-term reward. Unlike supervised learning, where the agent is trained on labelled data with correct answers, RL agents learn through trial and error by interacting with the environment. There are four main components in RL:

- **Agent**: The decision-maker that learns and interacts with the environment.
- **Environment**: Everything the agent interacts with, providing observations and rewards.
- **Actions**: The things the agent can do in the environment.
- **Rewards**: Feedback signals the agent receives from the environment based on its actions.

The components in our problem are:

- **Agent**: Deep Q-Network
- **Environment:** We observe the number of successfully preambles, idle preambles, and collided preambles
- **Actions**: [n_rach_0, n_repe_0, ..., f_prea_2]
- **Rewards**: The number of successfully served devices

The agent explores the environment by taking actions and observing the outcomes. It receives rewards or penalties based on the chosen action. Over time, the agent learns to associate its actions with the rewards it receives. It favours actions that lead to higher rewards, gradually improving its decision-making. The agent develops a policy, a set of rules that maps observations from the environment to the most suitable actions to take.

Essentially, RL allows machines to learn like humans do, through trial and error and by receiving feedback from their actions. This makes RL a powerful tool for training agents to perform tasks in complex and dynamic environments where traditional programming might be difficult.

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in reinforcement learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the credit assignment problem: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for? To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a discount factor, γ (gamma), at each step. This sum of discounted rewards is called the action's return.

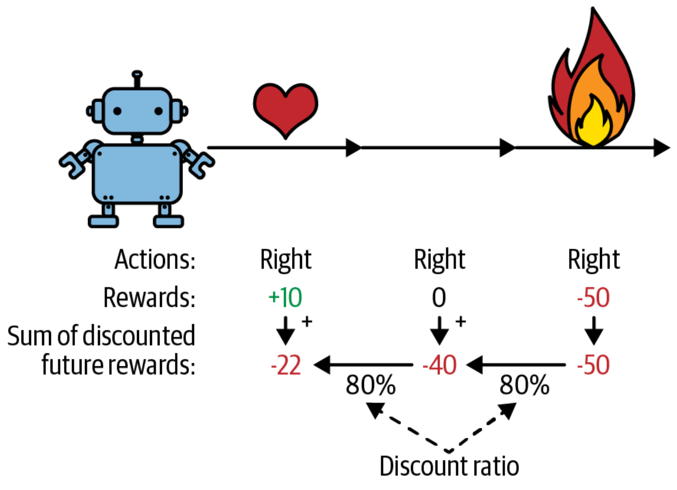


Fig. 5. Computing an action's return: the sum of discounted future rewards [1]

Consider the example in Fig. 5. If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount factor γ = 0.8, the first action will have a return of 10 + γ × 0 + γ2 × (–50) = –22.

3.2 LEARNING AND POLICY

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called Markov chains. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s′ is fixed, and it depends only on the pair (s, s′), not on past states. This is why we say that the system has no memory.

Markov decision processes were first described in the 1950s by Richard Bellman. They resemble Markov chains, but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

Bellman found a way to estimate the optimal state value of any state s, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches the state, assuming it acts optimally. He showed that if the agent acts optimally, then the Bellman optimality equation applies (see Equation 3.1). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

$$V^*(s) = \max_{a} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot V^*(s')] \tag{3.1}$$

In Equation 3.1,

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
- γ is the discount factor.

Reinforcement learning problems with discrete actions can often be modelled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know T(s, a, s′)), and it does not know what the rewards are going to be either (it does not know R(s, a, s′)). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The temporal difference (TD) learning algorithm is very similar to the Q-value iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general, we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an exploration policy—for example, a purely random policy—to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 3.2).

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\big(r + \gamma \cdot V_k(s')\big) \tag{3.2}$$

In equation 3.2,

- α is the learning rate.
- r is the reward.
- γ is the discount factor.

3.3 Q-Values

Q-values, in the context of reinforcement learning (RL), represent the expected cumulative rewards an agent can obtain by taking a specific action in a given state and then following a particular policy thereafter. Specifically, the Q-value of taking action *a* in state s, denoted as *Q*(*s*,*a*), indicates the expected return (or cumulative reward) if the agent starts in state *ss*, takes action *aa*, and then acts optimally thereafter, following a certain policy.

The Q-value is often computed using the Bellman equation, which expresses the relationship between the value of a state-action pair and the values of possible successor states:

$$Q(s,a) = R(s,a) + \gamma \sum_{s\prime} P(s' \mid s,a) \max_{a'} Q(s',a') \qquad (3.3)$$

Where:

- *R*(*s*, *a*) is the immediate reward received after taking action *a* in state *s*.
- *P*(*s*′|*s*, *a*) is the probability of transitioning to state *s*′ from state *s* by taking action *aa*.
- γ is the discount factor, which determines the importance of future rewards relative to immediate rewards.
- $\max_{a'} Q(s',a')$ represents the maximum Q-value achievable from the successor state *s*′, reflecting the agent's optimal action selection.

Q-values are essential in RL algorithms like Q-learning and Deep Q-Networks (DQN), where agents learn to estimate these values through interaction with the environment to make informed decisions and maximize long-term rewards.

3.4 Q-Learning

Q-learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-values. Once it has accurate Q-value estimates (or close enough), then the optimal policy is just choosing the action that has the highest Q-value (i.e., the greedy policy).

$$Q(s,a) \underset{a}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s',a') \tag{3.4}$$

Where:

- r is the immediate reward received after taking action a in state s.
- γ is the discount factor, which determines the importance of future rewards relative to immediate rewards.
- $\max_{a'} Q(s',a')$ represents the maximum Q-value achievable from the successor state s'

For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-value estimates for the next state s′, since we assume that the target policy will act optimally from then on.

This algorithm will converge to the optimal Q-values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. The Q-value iteration algorithm converges very quickly, in fewer than 20 iterations, while the Q-learning algorithm takes about 8,000 iterations to converge in simple problems. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder.

The Q-learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one used during training. After training, the optimal policy corresponds to systematically choosing the action with the highest Q-value. Conversely, the policy gradients algorithm is an *on-policy* algorithm: it explores the world using the policy being trained. It is somewhat surprising that Q-learning is capable of learning the optimal policy by just watching an agent act randomly.

The main problem with Q-learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-learning to train an agent to play Pac-Man, there are about 150 pellets that Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible states is greater than $2^{150} \approx 10^{45}$. And if you add all the possible combinations of positions for all the ghosts and Pac-Man, the number of possible states becomes larger than the number of atoms in our planet, so there's absolutely no way you can keep track of an estimate for every single Q-value.

The solution is to find a function $Q_{\boldsymbol{\theta}}(s, a)$ that approximates the Q-value of any state action pair (s, a) using a manageable number of parameters (given by the parameter vector $\boldsymbol{\theta}$). This is called approximate Q-learning. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., the distances of the closest ghosts, their directions, and so on) to estimate Q-values, but in 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-values is called a deep Q-network (DQN) and using a DQN for approximate Q-learning is called deep Q-learning.

3.5 Neural Networks

Neural networks, inspired by the intricate workings of the human brain, stand as a cornerstone of contemporary artificial intelligence (AI) and machine learning (ML). They wield formidable prowess across a spectrum of tasks, from pattern recognition and classification to regression and the generation of art and music. At their essence, neural networks consist of interconnected artificial neurons arranged in layers. Each neuron receives input signals, processes them through weighted connections, and generates an output signal. These connections, known as synapses, possess weights that govern the impact of input signals on a neuron's activation. Through a process of training, neural networks adjust these weights to discern underlying patterns in input data.

Typically, a neural network's architecture comprises an input layer, one or more hidden layers, and an output layer. Hidden layers facilitate the network's capacity to discern complex patterns by applying nonlinear transformations. Deep neural networks, with numerous hidden layers, excel at tasks necessitating sophisticated feature extraction and hierarchical representation learning.

Training neural networks involves exposing them to labelled data in supervised learning or enabling them to interact with the environment and learn from feedback in reinforcement learning. Optimization algorithms like stochastic gradient descent iteratively adjust the network's weights to minimize the disparity between predicted and actual outputs, honing its ability to generalize from training data to unseen instances.

Neural networks' impact transcends myriad domains, spanning computer vision, natural language processing, speech recognition, robotics, healthcare, finance, and beyond. In computer vision, convolutional neural networks (CNNs) have revolutionized image recognition tasks, while in natural language processing, recurrent neural networks (RNNs) and transformers have achieved groundbreaking results in tasks like language translation and text generation.

Despite their successes, neural networks pose challenges such as the demand for extensive labelled data, computational resources, and concerns regarding robustness against adversarial attacks and biases in training data. Nevertheless, relentless research continues to expand the frontiers of neural network capabilities, driving innovations poised to redefine AI and revolutionize our interaction with technology.

In this project we have used the following neural networks:

1. Fully Connected Neural Network (FNN)
2. Long-short term memory Network (LSTM)

A fully connected neural network, also known as a dense or feedforward neural network, is a fundamental architecture in the realm of artificial intelligence and machine learning. It serves as the building block for more complex models and has found extensive applications across various domains, including image recognition, natural language processing, and time series forecasting.

At its core, a fully connected neural network comprises layers of neurons organized in a sequential manner. Each neuron in a layer is connected to every neuron in the subsequent layer, hence the term "fully connected." This connectivity pattern allows the network to capture intricate relationships and patterns in the data it processes.

The structure of a fully connected neural network typically consists of three types of layers: input layers, hidden layers, and output layers. The input layer receives the raw data, such as pixel values in an image or words in a text document. Each neuron in the input layer represents a feature or attribute of the input data.

The hidden layers, situated between the input and output layers, are responsible for learning complex representations of the input data. These layers perform nonlinear transformations on the input data, extracting higher-level features that are increasingly abstract and informative. The depth and width of the hidden layers determine the network's capacity to capture intricate patterns in the data.

The output layer produces the network's predictions or decisions based on the learned representations. The number of neurons in the output layer depends on the nature of the task the network is designed to solve. For example, in a binary classification task, the output layer typically consists of a single neuron representing the probability of belonging to one of the two classes. In multi-class classification tasks, the output layer may contain multiple neurons, each corresponding to a distinct class.

During the training phase, a fully connected neural network learns to make accurate predictions by adjusting the weights and biases associated with its connections. This process, known as backpropagation, involves iteratively updating the network's parameters to minimize the discrepancy between its predictions and the ground truth labels in the training data. Optimization algorithms such as gradient descent are commonly used to adjust the parameters in the direction that minimizes the loss function, which quantifies the difference between predicted and actual outputs.

In conclusion, fully connected neural networks serve as a foundational architecture in the field of machine learning, enabling computers to learn from data and make intelligent decisions. Their ability to learn complex patterns and representations from raw data has led to significant advancements in various domains, laying the groundwork for the development of more sophisticated and specialized neural network architectures.

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture specifically designed to overcome the limitations of traditional RNNs in capturing long-range dependencies and retaining information over extended sequences. LSTM networks have found widespread applications in various domains, including natural language processing, speech recognition, time series forecasting, and sequential data analysis.

At the heart of an LSTM network are memory cells, which serve as the building blocks responsible for storing and updating information across time steps. Unlike standard RNNs, which have a simple recurrent connection, LSTM networks feature a more complex architecture that includes multiple interconnected gates—namely, the input gate, forget gate, and output gate—allowing for precise control over the flow of information.

The input gate regulates the flow of new information into the memory cell. It decides which information from the current input and the previous hidden state should be stored in the cell state. The forget gate controls the retention or removal of information stored in the cell state from the previous time step. It determines which information is irrelevant or outdated and should be discarded. Finally, the output gate governs the flow of information from the memory cell to the output of the LSTM network.

These gates, each implemented as a sigmoid neural network layer followed by a pointwise multiplication operation, collectively enable LSTM networks to learn to store and update information over long sequences while mitigating the vanishing gradient problem commonly encountered in standard RNNs.

One of the key advantages of LSTM networks is their ability to capture and retain long-term dependencies in sequential data. This is achieved through the mechanism of the cell state, which serves as a conveyor belt, allowing information to persist and propagate across multiple time steps. By selectively updating and forgetting information, LSTM networks can maintain relevant context and memory over extended sequences, making them particularly well-suited for tasks requiring modelling of temporal dynamics and context dependencies.

In addition to their effectiveness in modelling long-range dependencies, LSTM networks also excel in handling variable-length sequences and addressing the issue of exploding or vanishing gradients during training. The gated architecture of LSTM networks enables them to learn more stable and robust representations of sequential data, leading to improved performance on tasks such as language modelling, machine translation, sentiment analysis, and speech recognition.

Despite their advantages, LSTM networks are not without limitations. They require more computational resources and training data compared to simpler architectures like feedforward neural networks or traditional RNNs. Additionally, designing, and tuning LSTM architectures can be challenging, requiring careful consideration of factors such as the number of layers, hidden units, and training hyperparameters.

3.6 Deep Q-Network

Deep Q-Network (DQN) is a pioneering reinforcement learning algorithm developed by DeepMind that integrates deep learning with Q-learning to enable agents to learn optimal policies directly from raw sensory inputs, such as images or sensor data. While DQN typically incorporates a target network to stabilize training.

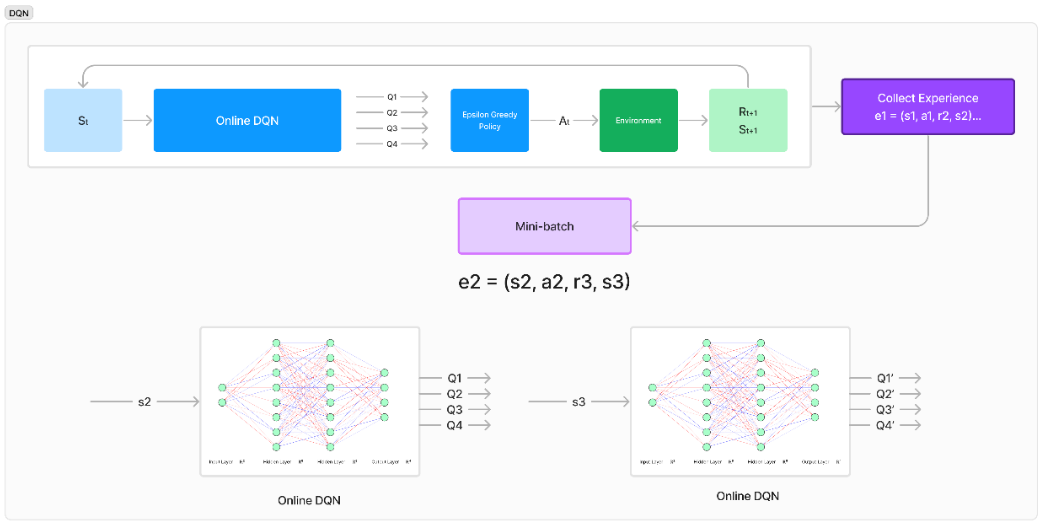


Fig. 6. Training in Deep Q-Learning without target network

1. Q-Learning Update:
    - At its core, DQN aims to learn an optimal action-value function $Q(s,a)$ that estimates the expected cumulative reward for taking action aa in state ss. The action-value function represents the value of selecting each action in each state and guides the agent's decision-making process.
    - DQN updates the action-value function iteratively based on the temporal difference error, which is the discrepancy between the current estimate of the Q-value and the target Q-value. The Q-learning update rule is:
    $$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)] \quad (3.5)$$
    - where r is the immediate reward, γ is the discount factor, s' is the next state, and α is the learning rate.
2. Experience Replay:
    - DQN utilizes experience replay to improve learning efficiency and stability by storing past experiences (transitions) in a replay memory buffer.
    - During training, experiences, consisting of tuples (state, action, reward, next state), are collected by interacting with the environment and stored in the replay buffer.
    - Mini-batches of experiences are sampled randomly from the replay buffer to decorrelate the data and mitigate the impact of temporal correlations.
3. Q-Network:
    - DQN employs a deep neural network, known as the Q-network, to approximate the action-value function $Q(s,a)$. This network is composed of few layers of
    - The Q-network takes the state ss as input and outputs the predicted Q-values for all possible actions.
    - The neural network architecture typically consists of convolutional layers followed by fully connected layers, enabling the model to learn hierarchical representations from raw sensory inputs.
4. Training Process:

- During training, the Q-network is updated to minimize the temporal difference error between predicted and target Q-values.
- Target Q-values are computed directly using the same Q-network, replacing the need for a separate target network.
- The Q-network parameters are updated using gradient descent, optimizing the network to better approximate the true action-value function.

5. Exploration vs. Exploitation:
    - To balance exploration (trying new actions) and exploitation (choosing actions with high predicted values), DQN typically employs an epsilon-greedy strategy.
    - Initially, the agent explores the environment by taking random actions with high probability. As training progresses, the exploration rate decreases to favour exploitation.

DQN is a powerful reinforcement learning algorithm that leverages deep learning to learn optimal policies directly from raw sensory inputs. By combining experience replay, neural network approximation, and Q-learning updates, DQN achieves impressive performance in various challenging reinforcement learning tasks, making it a foundational algorithm in the field.

In Deep Q-Network (DQN), the inclusion of a target network is essential for stabilizing training and enhancing learning efficiency. The primary purpose of the target network is to provide stable target Q-values for training, thereby mitigating issues related to the instability of the learning process.

One of the core challenges in training deep reinforcement learning algorithms like DQN is the instability of learning caused by the interplay between the target estimation and the parameter updates of the neural network. During training, the Q-network's parameters are updated to minimize the temporal difference error between predicted and target Q-values. However, since the target values are derived from the same Q-network that is being updated, the target Q-values can become highly volatile and can "chase" the rapidly changing Q-values during training. This phenomenon, known as the "moving target syndrome," can lead to oscillations, divergence, and slow convergence of the Q-network.

The target network addresses this issue by decoupling the target estimation from the Q-network's parameter updates. It provides a fixed reference point for the target Q-values, ensuring that the targets remain stable over multiple updates. The target network is typically a separate copy of the Q-network, with its parameters periodically updated to match those of the Q-network. By freezing the parameters of the target network for a certain number of iterations or updating them slowly over time, the target Q-values become more consistent and less volatile, providing a smoother and more reliable training signal for the Q-network.

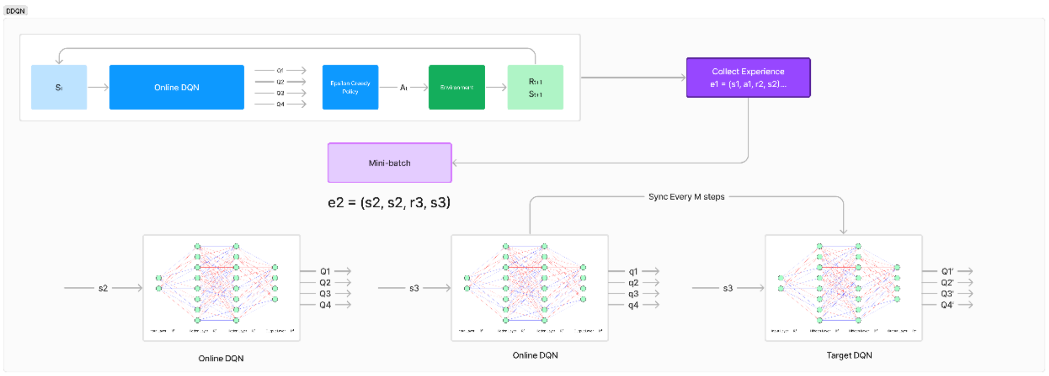


Fig. 7. Training in Deep Q-Learning with target network

While both DQN and Double DQN use neural networks to approximate the action-value function, the key difference lies in how they handle the estimation of target Q-values. Double DQN mitigates the overestimation bias inherent in DQN by employing a separate target Q-network for target value estimation, resulting in more accurate and stable learning.

This stability in target estimation has several benefits for the training process:

- **Reduction of Oscillations**: By providing consistent target Q-values, the target network helps reduce oscillations in the learning process. This allows the Q-network to learn more smoothly and converge faster towards the optimal policy.

- **Improved Sample Efficiency**: Stable target Q-values enable more efficient use of training samples. Without fluctuations in the target values, the Q-network can learn from each sample more effectively, leading to faster learning and better performance.
- **Enhanced Generalization**: Stable target Q-values promote better generalization of the learned policy to unseen states. The Q-network learns a more robust representation of the environment, leading to improved performance in real-world scenarios.
- **Mitigation of Overestimation Bias**: Target networks also help mitigate overestimation bias, a common issue in Q-learning algorithms where the estimated Q-values are consistently higher than the true values. By providing more stable target values, the target network helps reduce the impact of overestimation bias, leading to more accurate and reliable Q-value estimates.

When dealing with a Partially Observable Markov Decision Process (POMDP), we may not have full access to the state information, and therefore, we need to record the history or trajectory information to assist in choosing the action. Long Short-Term Memory (LSTM) networks are introduced to Deep Q Learning to handle POMDPs by encoding the history into the hidden state of the LSTM.

In this case, the optimization target is changed to predicting the next Q-value given the current state and the history. We can use the LSTM to encode the history of past states and actions, and then pass this encoded information along with the current state to the Q network to predict the Q-value. The goal is to minimize the difference between the predicted Q-value and the target Q-value obtained from the Bellman equation.

3.7 Multi-Agent Learning

Multi-Agent Reinforcement Learning (MARL) is a subfield of reinforcement learning (RL) where multiple agents learn to interact with an environment to achieve individual or collective goals. Unlike single-agent RL, where there's only one learner interacting with the environment, MARL involves multiple learners simultaneously interacting with each other and the environment.

In MARL, agents can be cooperative, competitive, or a combination of both. They learn by observing the environment's state, taking actions, receiving rewards, and updating their policies accordingly. The challenge in MARL lies in the complexity introduced by the interactions between agents. Actions taken by one agent can influence the environment and affect the learning process of other agents.

There are various approaches to address MARL problems, including centralized training with decentralized execution, where agents share information during training but act independently during execution. Other approaches include learning communication protocols between agents or using self-play techniques to train agents in competitive scenarios.

MARL finds applications in diverse domains such as robotics, autonomous driving, game theory, and decentralized control systems. It offers a framework to model complex systems where multiple entities need to learn and adapt to dynamic environments, facilitating the development of intelligent and collaborative autonomous systems.

In our problem, single-group single-parameter environment can be solved with traditional Reinforcement Learning approaches but when it comes to the actual scenario, we have several groups and parameters. Hence, our action state is enormous and thus we cannot have that many Q-values produced out of a network.

Therefore, we use multiple agents, one per parameter per group, therefore we would have a total of 9 agents who need to work cooperatively to maximise the reward signal which is common to all. Therefore, we shall go with a Cooperative Multi-Agent DQN (CMA-DQN) approach.

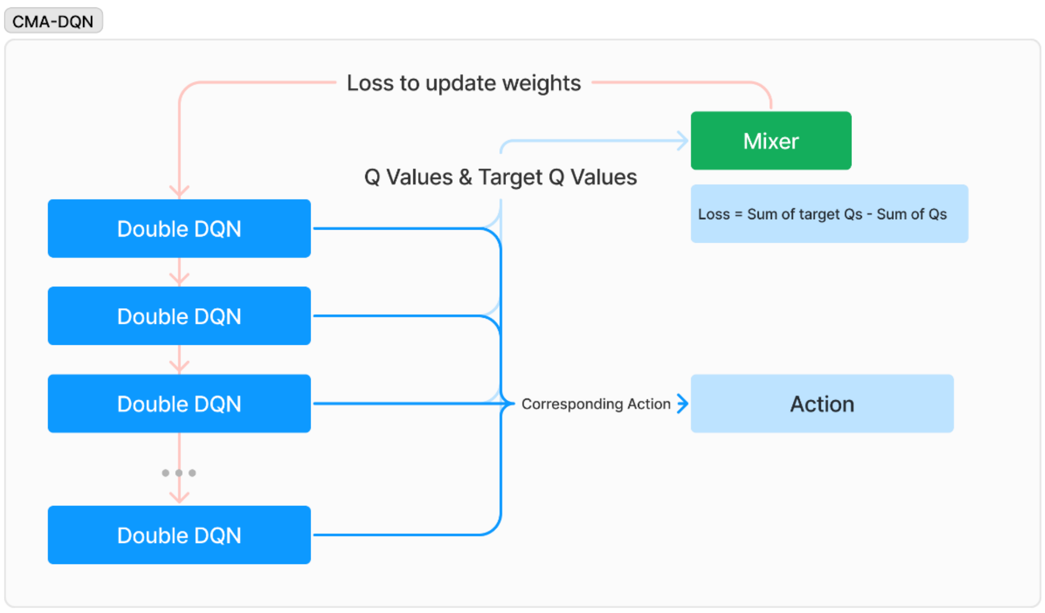


Fig. 8. CMA-DQN with Value Decomposition Network

Optimizing the joint policy of multiple agents with a single team reward can be a challenging task due to the large, combined action and observation space. Value Decomposition Network (VDN) was introduced as a solution to this problem. The algorithm decomposes the joint Q value into the sum of individual Q values for each agent. This allows each agent to learn and optimize its own policy independently while still contributing to the team reward. In VDN, each agent follows a standard D(R)QN sampling pipeline and shares its Q value and target Q value with other agents before entering the training loop. The Q value and target Q value of the current agent and other agents are then summed in the training loop to get the total Q value.

- Each agent is still a standard Q, use self-observation as input and output the action Q values.
- The Q values of all agents are added together for mixed Q value annotated a Q^{tot}
- Using standard DQN to optimize the Q net using with the team reward r.
- The gradient each Q net received depends on the contribution of its Q value to the Q^{tot}.

The Q net that outputs a larger Q will be updated more; the smaller will be updated less.

IV. EXPERIMENTS AND RESULTS

4.1 SIMULATION SETUP

In the single-parameter single-group scenario, eNB is located at the centre of a circular area, and the IoT devices are randomly located within the cell. We set the number of RACH periods as n_rach = 1, the repetition value as n_repe = 4, and f_prea to be decided by the agent during the process. The DQN is set with three hidden layers, each with 128 ReLU units.

We generated a flow of generated new devices with its peak being at around 400 TTI and the total being 1000 TTI. The parameters given in Table 1, were used.

Table 1. Single Group Simulation Parameters

Parameters	Values
Probability of RCC Failure	0.6
Set of number of preambles	{12, 24, 36, 48}
Repetition Value	4
Number of RACH Period	1
Learning Rate	0.01
RMSProp Learning Rate	0.0001
Initial Exploration	0.1
Discount Rate	0.5
Minibatch Size	32
Replay Memory	10000
Target Q-network update frequency	1000

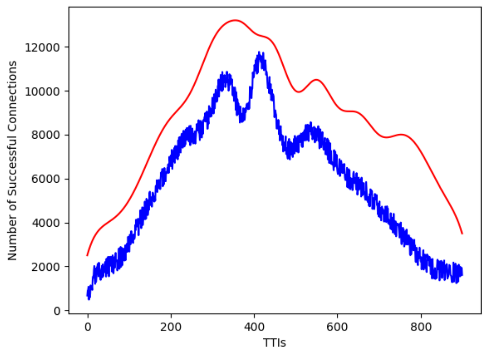
We generated a flow of generated new devices with its peak being at around 400 TTI and the total being 1000 TTI. The parameters given in Table 1, were used.

Whereas for Multiple Parameter scenarios, we adjusted the Probability of RCC Failure to increase for different CE groups, and the values of n_rach and n_repe also to be decided by the agent. The DQN is set with three hidden layers of LSTM, each with 128 ReLU units. The parameters given in Table 2, were used.

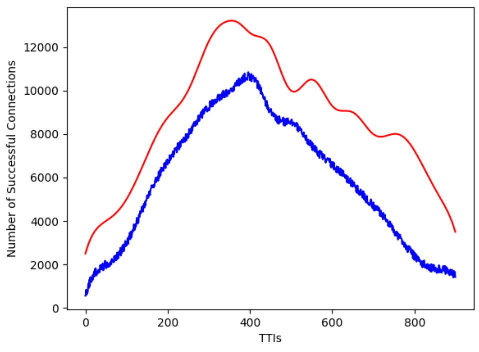
Table 2. Multiple Group Simulation Parameters

Parameters	**Values**
Probability of RCC Failure for i^{th} CE group	0.5 + 0.05*i
Set of number of preambles	{12, 24, 36, 48}
Set of Repetition Value	{1, 2, 4, 8, 16, 32}
Set of Number of RACH Period	{1, 2, 4}
Learning Rate	0.01
RMSProp Learning Rate	0.0001
Initial Exploration	0.1
Discount Rate	0.5
Minibatch Size	32
Replay Memory	10000
Target Q-network update frequency	1000

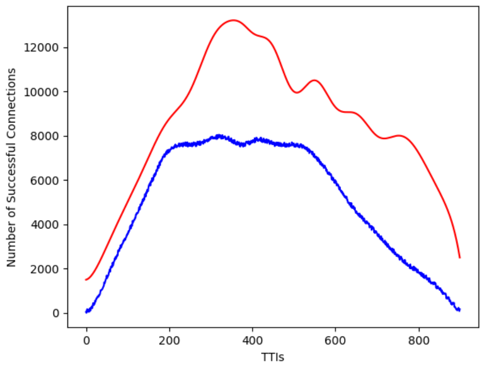
4.2 RESULTS AND DISCUSSION



(a)



(b)



(c)

Figure 9. Observations of (a) DQN, (b) Double DQN, (c) CMA-DQN in blue and number of new devices in red

It is evident from our empirical analysis that the Double Deep Q-Network (Double DQN) algorithm exhibits markedly enhanced stability when compared to its predecessor, the standard Deep Q-Network (DQN). Moreover, Double DQN not only demonstrates superior performance under high load conditions but also showcases heightened robustness in the face of dynamic and fluctuating observation environments. The observed fluctuations, while anticipated due to the absence of heuristic guidance within the employed algorithms, emphasizing the pure learning nature of the implemented methodologies. This volatility underscores the inherent adaptability and capacity for learning exhibited by this reinforcement learning systems, paving the way for further optimization and refinement in algorithmic design.

In the intricate landscape of multi-agent multi-parameter scenarios, our findings indicate a trend where the outputs exhibit lower magnitudes yet are characterized by a notably heightened level of stability. This phenomenon can likely be attributed to the inherent complexity of the problem domain, where the interplay of multiple agents and parameters introduces layers of intricacy that demand more nuanced learning strategies. Moreover, it is plausible that the observed lower output values stem from the insufficient duration of training time allocated to these complex scenarios.

V. CONCLUSION

5.1 CONCLUSION

In this project, we developed Q-learning based uplink resource configuration approaches to optimize the number of served IoT devices in real-time in NB-IoT networks. We first implemented DQN based approaches for the single-parameter single-group scenario. Our results demonstrated that Double DQN can be good alternatives for DQN to achieve almost the same system performance with much higher stability. To support traffic with different coverage requirements, we then studied the multi-parameter multi-group scenario as defined in NB-IoT standard, which introduced the high-dimensional configurations. To solve it, we developed CMA-DQN by dividing high dimensional configurations into multiple parallel sub-tasks, which achieved the best performance in terms of the number of successfully served IoT devices with the least training time.

5.2 SCOPE FOR FUTURE WORK

To enhance the effectiveness of existing methodologies, several strategic considerations can be implemented:

- Hyperparameter Variation: Enhance the robustness and adaptability of existing methodologies by systematically varying hyperparameters.
- Complex Environment Modelling: Elevate the fidelity of simulation environments to mirror real-life complexities more accurately.
- Model Size Optimization: Streamline the computational footprint of models to enable deployment in resource-constrained environments. By refining model architectures and employing techniques such as parameter pruning and compression, the size of the model can be significantly reduced without compromising performance.

REFERENCES

1. Géron, Aurélien. 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. "O'Reilly Media, Inc."
2. Jiang, Nan et al. 2019. "Reinforcement Learning for Real-Time Optimization in NB-IoT Networks," IEEE Journal on Selected Areas in Communications 37(6): 1424–40.
3. Morales, Miguel. 2020. Grokking Deep Reinforcement Learning. Manning Publications.
4. Van Hasselt, Hado et al. 2015. "Deep Reinforcement Learning with Double Q-Learning," arXiv (Cornell University).
5. Z. Wang, T. Schaul, M. Hessel, V. H. Hado, M. Lanctot, and D. F. Nando, "Dueling network architectures for deep reinforcement learning," arXiv.org, Nov. 20, 2015. https://arxiv.org/abs/1511.06581
6. P. Sunehag et al., "Value-Decomposition networks for cooperative Multi-Agent learning," arXiv (Cornell University), Jan. 2017, doi: 10.48550/arxiv.1706.05296.
7. H. Kinsley and D. Kukieła, Neural Networks from Scratch in Python. 2020.