
Master Thesis

University of Bergen

Department of Informatics

# Deep Reinforcement Learning for Computation Offloading in Mobile Edge Computing

*Author:*
Sondre Eide Omland

*Supervisors:*
Yauhen Yakimenka
Hsuan-Yin Lin
Eirik Rosnes

June 1, 2022

# Acknowledgements

Writing this thesis has been an amazing journey with numerous challenges and knowledgeable discussions about mobile edge computing, neural networks, and reinforcement learning. I want to thank my supervisors, Yauhen Yakimenka, Eirik Rosnes, and Hsuan-Yin Lin, for all the hours they have spent on guidance. The completion of this thesis would not have been possible without their help. Furthermore, I would like to thank my girlfriend and family for their support. Additionally, I would like to thank all my fellow students throughout my time in Bergen for all the good moments together.

# Abstract

As 5G-networks are deployed worldwide, mobile edge computing (MEC) has been developed to help alleviate resource-intensive computations from an application. Here, IoT devices can offload their computation to an MEC server and receive the computed result. This offloading scheme can be viewed as an optimization problem, where the complexity quickly increases when more devices join the system. In this thesis, we solve the optimization problem and introduce different strategies that are compared to the optimal solution. The strategies implemented are full local computing, full offload to an MEC server, random search, optimal solution, Q-learning, and a deep Q-network (DQN). The main objective for each strategy is to minimize the total cost of the system, where the cost is a combination of energy consumption and delay. However, as the number of devices in the system increases, the results view numerous challenges. This thesis shows that the performance of random search, Q-learning, and DQN strategies are very close to the optimal solution for up to 20 devices. However, the results show generally poor performance for the strategies that can address more than 20 devices. In the end, we further discuss the performance and convergence of a DQN in MEC.

# List of Figures

# List of Tables

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

In the past decade, the computational capacity of mobile devices has increased and has become an essential part of our day-to-day life. As mobile technologies continue to improve, new resource-intensive applications such as augmented reality, virtual reality, or online gaming can be utilized to a greater effect for business and entertainment purposes. These applications require a significant amount of resources from the device. Therefore, it is essential to develop new solutions to resource-intensive computations to mitigate these limitations.

In the following years, 5G networks will be deployed worldwide and provide higher connectivity and availability to users. With the expansion of 5G infrastructure, mobile edge computing (MEC) has been developed to solve resource-intensive computations and bring computation environments closer to the user. The users can send their computations to high-performance edge servers and receive the result of the computation. Although this solution may solve many of the issues with resource-intensive computations, it does not consider an efficient offloading scheme between multiple users, edge servers, or base stations.

Historically, mobile devices have been limited to standard communication methods such as text messaging and calls. With the introduction of smartphones and other IoT devices, the computational environment has become more dynamic. Here, neural networks and machine learning algorithms can be utilized as efficient tools for offloading and adaption to frequent changes in a device for better handling of the dynamic environment.

Machine learning algorithms that employ neural networks are a relatively new area of research that has seen significant advancements in the past decade and will continue increasing as machines become more capable of solving more advanced problems. However, in terms of finding an improved offloading scheme for MEC, a machine learning model with good performance that can scale for a higher number of devices is yet to be developed. The offloading scheme between users and edge servers has high computational complexity and is commonly viewed as an optimization problem.

## 1.2 Objective

This thesis aims to examine the use of reinforcement learning (RL) techniques, such as Q-learning and deep reinforcement learning (DRL), to find optimal resource allocations for offloading computations in MEC. These techniques will be compared and analyzed to heuristic solutions in the form of full local and full offloading to the edge server, random search, and a brute force of optimal solutions for different action vectors and resource allocations. Previous studies in the field of MEC will also be explained and reviewed.

## 1.3 Thesis Organization

- **Chapter 2:** Overview of relevant key concepts from MEC, RL, neural networks, and DRL
- **Chapter 3:** Overview of relevant related work within MEC, specifically RL and DRL
- **Chapter 4:** Explanation of methodology, strategies implemented, and model architecture
- **Chapter 5:** Detailed information of the results and findings
- **Chapter 6:** Conclusions regarding results obtained, and suggested future work
- **Chapter 7:** Appendices

# Nomenclature

| | |
|---|---|
| uniform($\mathcal{S}$) | A number drawn uniformly at random from a set $\mathcal{S}$ |
| DQN | Deep Q-network |
| DRL | Deep reinforcement learning |
| MAE | Mean absolute error |
| MEC | Mobile edge computing |
| MGD | Mini-batch gradient descent |
| MSE | Mean squared error |
| RL | Reinforcement learning |
| SL | Supervised learning |
| TD | Temporal difference |

| | |
|---|---|
| $[n]$ | $\{1, 2, \ldots, n\}$, where $n$ is a positive integer |
| $\beta$ | Size of the agent's neighborhood |
| $\boldsymbol{f}$ | Allocation vector |
| $\boldsymbol{\alpha}$ | Action vector |
| $\boldsymbol{\Theta}$ | Neural network parameters |
| $\boldsymbol{\Theta}^-$ | Target neural network parameters |
| $\eta$ | Learning rate for neural networks |
| $\gamma$ | Discount parameter |
| $\mathbb{E}$ | Expectation operator |
| $\mathcal{E}$ | Replay buffer |
| $\mathcal{N}$ | Set of devices |
| $\mathcal{N}^{\boldsymbol{\alpha}}$ | Set of offloading devices |

| | |
|---|---|
| $\mathcal{X}$ | Sets |
| $\mathsf{X}$ | Random variables |
| $\nabla$ | Gradient operator |
| $\nu_\pi$ | Value function in RL |
| $\psi$ | Maximum number of steps the agent can make |
| $\varphi$ | Available computational capacity left on server |
| $\zeta$ | Number of discrete allocation quantities |
| $b$ | Length of replay buffer |
| $M$ | Size of random mini-batch taken from the replay buffer |
| $\boldsymbol{X}$ | Matrices |
| $\boldsymbol{x}$ | Vectors |

# Chapter 2

# Background

For a better understanding of the model and architecture described later in the thesis, the fundamental concepts of MEC, machine learning, and deep learning will be explained. This chapter aims to provide a general understanding of the concepts within these areas for a more comprehensive analysis of methods and techniques used in Chapter 4.

## 2.1 Introduction to Mobile Edge Computing

This section provides an introduction to MEC and provides the basis of methods used. Unless otherwise specified, expressions, figures, and examples are taken from [1].

### 2.1.1 Mobile Edge Computing

MEC, also referred to as multi-access edge computing, is a recently emerged technology that provides cloud computing capabilities at the edge of the mobile network, within the radio access network, and in close proximity to users, as stated in [2]. MEC aims to provide enhanced computing capabilities with low latency for a better user experience. An MEC *system* is structured with multiple layers: *user layer*, *edge layer*, and *cloud layer* as illustrated in Figure 2.1.

#### User Layer

The *user layer* consists of several different IoT devices ranging from smartphones and laptops to vehicles and sensors that need to execute a computation. The computations, which vary for each device, are sent to the edge layer using wireless communication and separated into different networks.

The first network is the heterogeneous network and consists of devices that require high data rates. Here, a significant number of small macro base stations are deployed in populated areas to provide high connectivity and reduce the power consumption for mobile devices. Each base station is equipped with powerful computing equipment, to which mobile devices can offload their tasks.

The second network is the vehicular network which consists of transportation units, pedestrians, and roadside units. A roadside unit is a wireless communication device with computing capabilities that enables communication between trans-

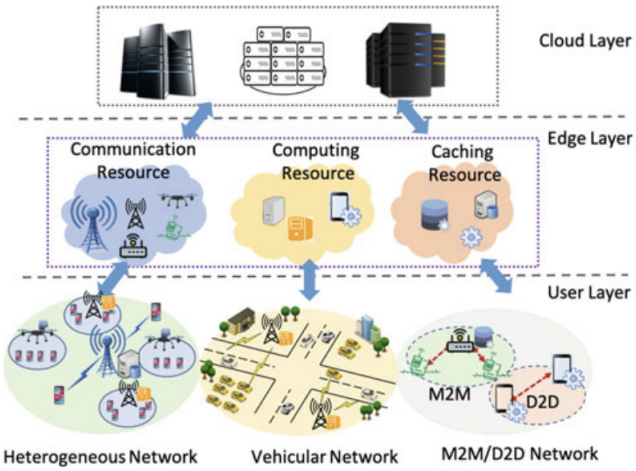


Figure 2.1: MEC architecture.

portation units, pedestrians, and nearby infrastructures, such as traffic lights or roadblocks, to improve safety in traffic and hinder violations. The vehicular network allows for smart applications, such as media streaming in cars or parking identification, to be utilized to a greater effect in traffic and provide a higher quality of service.

The third and last network is the device-to-device (D2D) network. The device-to-device network is a peer-to-peer network where IoT devices communicate directly through a wireless link. This network distributes the computing capabilities throughout the network where devices can offload tasks to other devices in addition to an edge server.

### Edge Layer

The *edge layer* consists of several distributed servers with enhanced computing capabilities that aim to solve tasks from the *user layer* and can be deployed to different areas such as subway stops, highways, and airports to reduce latency. An edge server's main goal is to efficiently solve both time-sensitive and computing-intensive tasks through an offloading scheme. Therefore, an edge server for the heterogeneous network needs enhanced communication resources such as bandwidth and transmission power to efficiently solve tasks. In contrast, an edge server for the vehicular network needs computing resources in the form of CPU power, while an edge server for the D2D network needs larger caching resources in terms of memory capacity.

### Cloud Layer

In the last layer, the *cloud layer*, multiple edge servers are connected through the cloud. The cloud layer can utilize data mining techniques to train large neural networks to help edge servers allocate resources more efficiently. Furthermore, the cloud layer is also able to store higher amounts of network metadata such as network connections, type of computation, and device information that will be too overwhelming for the edge server. Additionally, with the help of the cloud layer, edge servers will also be more efficiently managed and secured.

### 2.1.2 Offloading Schemes

One of the major challenges in MEC for IoT devices is the decision between local execution or offloading. There exist multiple different offloading schemes that can be utilized, with some key differences between them that will be discussed.

#### Local Execution

The first scheme is full local execution. With this scheme, the device solves the whole computation locally with the available resources on the device. If the computation is large, the device can suffer due to limited resources. However, local execution might be a faster solution if the MEC server is severely occupied or if the delay to the MEC server is large.

#### Full Offloading

The second offloading scheme is full offloading to the MEC server. This scheme offloads the whole computation to the MEC server with enhanced computing capabilities. If there are few devices in the system or the device needs to execute a large computation, offloading to an MEC server might be a faster solution.

#### Partial Offloading

The last offloading scheme for a device is partial offloading. Instead of a computation being offloaded as a whole, it will be separated into multiple segments where one computationally intensive part will be offloaded, and the other parts will be executed locally. The challenge for partial offloading is understanding which segments of a task should be offloaded and which should be executed locally. For simplicity, the rest of the thesis will concentrate on an offloading scheme where a task is offloaded or executed locally as a whole.

### 2.1.3 Performance Metrics

Regardless of the selected offloading scheme, a good offloading scheme is commonly measured with performance metrics such as task execution delay and energy consumption. This can be described as an optimization problem, which will be further discussed in Chapter 4. There have been different proposals to minimize both task execution delay and energy consumption for a system with numerous devices, edge computing servers, and base stations. Some papers only focus on minimizing one aspect of the offloading process, while others focus on a trade-off between task execution delay and energy consumption. Other performance metrics such as quality of service, response time, confidentiality, security, and bandwidth have also been researched. However, these performance metrics are not too common in research on optimal offloading schemes in MEC.

## 2.2 Machine Learning

This section provides an introduction to the concepts of machine learning, specifically RL. Unless otherwise specified, formulas and examples used are taken from [3].

### 2.2.1 General Knowledge of Machine Learning

As a sub-domain of artificial intelligence (AI), machine learning uses mathematical models to assist a computer in learning to solve a task from experience. The machine learning algorithm can make predictions on new unseen tasks by identifying patterns and correlations in a dataset, commonly referred to as training data. The objective of a good machine learning algorithm is to measure the accuracy or reward of the prediction made. This can be achieved in multiple ways with different learning techniques. Therefore, the different learning techniques in machine learning are commonly divided into supervised learning (SL), unsupervised learning (UL), and RL.

#### Supervised Learning

In SL, the machine learning algorithm interacts with a labeled dataset. The labeled dataset is a set of desired input and output pairs, where the model aims to find the correct output from the input. When the SL model makes a prediction, the accuracy of that prediction is measured with a loss function. For each training iteration, the model aims to minimize the loss. Common subsections of SL include regression problems, classification problems, decision trees, and neural networks.

#### Unsupervised Learning

In UL, the dataset is unlabeled and unstructured. Here, the algorithm aims to find structure and patterns within the dataset to group data together, referred to as a clustering problem. Furthermore, the algorithm may as well find associations between the data, often referred to as an association problem. Common methods of UL include hidden Markov models and game-AI, in addition to clustering and association problems.

### 2.2.2 Reinforcement Learning

In RL, an agent interacts with the environment using a control-theoretic trial and error approach to obtain rewards for favorable actions and punishments for incorrect actions. The agent stores rewards and actions performed in a table, $Q_{\text{table}}$, to learn from previous interactions. For a more detailed overview, when the agent observes the current state, $s_t$, at time step $t$ from the environment, the agent will perform an action, $a_t$, to transition to the next state, $s_{t+1}$, and receive a reward, $r_{t+1}$, that is calculated from the reward function, as illustrated in Figure 2.2. For each episode, consisting of multiple time steps $t$, the agent aims to maximize the cumulative reward based on a policy $\pi$. In RL, *policy*, *reward function*, *value function*, and *model* are the essential components of an RL *system*.

#### Policy

The agent's *policy* follows a probability distribution, $\pi(a_t|s_t) = \Pr(\mathsf{A}_t = a_t|\mathsf{S}_t = s_t)$, where there is a direct mapping between the current state of the agent, $s_t$, and the probabilities of selecting an action, $a_t$, from that state. The *policy* determines the behavior of the agent and follows a stochastic distribution over $a_t \in \mathcal{A}(s_t)$ for every

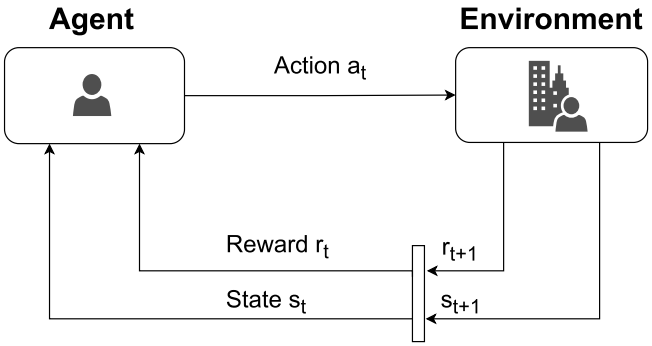


Figure 2.2: RL interaction model.

$s_t \in \mathcal{S}$. A policy method can be implemented as a function, lookup table, or a computational search algorithm.

### Reward Function

The *reward function* is crafted with the goal of providing information about the behavior of the agent, specifically how well it can learn. A properly constructed reward function accelerates the learning process by simplifying the distinction between good and wrong actions. When the agent interacts with the environment, the reward function provides feedback to the user with the rewards of the actions the agent performs. As previously mentioned, the agent's goal is to maximize the overall reward. However, for the agent to fulfill this goal, it has to balance between exploration and exploitation. If the agent performs an action that leads to a low reward in the next state, it might change the policy and disregard actions that yield higher rewards in future states.

### Value Function

The *value function*,

$$\nu_\pi(s) = \mathbb{E}_\pi[\mathsf{G}_t|\mathsf{S}_t = s], \quad \forall s \in \mathcal{S}, \tag{2.1}$$

estimates the long-term reward the agent can accumulate given the current state, $s_t$, and the policy, $\pi$, where $\mathbb{E}_\pi$ is defined as the expected value given the agent's policy. The random variable $\mathsf{G}_t$ is the expected return the agent can achieve for a given time step and is calculated as

$$\mathsf{G}_t = \sum_{i=0}^{\infty} \gamma^i \mathsf{R}_{t+i+1} = R_{t+1} + \gamma \mathsf{G}_{t+1}, \tag{2.2}$$

where $\gamma$ is defined as the discount parameter and determines how much future values should be considered, commonly set to be $0 \leq \gamma \leq 1$. By considering future states, the value function prevents the agent from failing to explore states that potentially yield higher rewards.

### Model

The *model* is an environment that the agent interacts with and consists of states and actions. The environment can be designed in many unique ways. For example,

a common way of implementing an RL model is using a grid with positions as states and the action being transitioning between positions, or the environment can be a trading platform where a state is the current stock price, and action is whether to sell, buy, or hold.

### 2.2.3 Optimality in Reinforcement Learning

We begin by defining the action-value function as

$$Q_\pi(s,a) = \mathbb{E}_\pi[\mathsf{G}_t|\mathsf{S}_t = s, \mathsf{A}_t = a]. \tag{2.3}$$

$Q_\pi(s,a)$ is a value indicating how good an action is from the current state while following a given policy. First define an optimal policy as $\pi_* = \arg\max_\pi Q_\pi(s,a)$. For a policy, $\pi$, to be classified as an optimal policy, there has to exist at least one policy that is equal to or better than every other policy. The optimal value function following an optimal policy can be defined according to

$$\nu_*(s) = \max_\pi \nu_\pi(s), \quad s \in \mathcal{S}. \tag{2.4}$$

The optimal value function ensures that the agent can accumulate the optimal long-term reward. After this, we define the optimal action-value function according to

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \tag{2.5}$$

where the agent follows an optimal policy. It can be shown that

$$\nu_*(s) = \max_a Q_*(s,a), \tag{2.6}$$

which leads to

$$\begin{aligned} Q_*(s,a) &= \mathbb{E}_{\pi_*}[\mathsf{G}_t|\mathsf{S}_t = s, \mathsf{A}_t = a] && (2.7)\\ &= \mathbb{E}_{\pi_*}[\mathsf{R}_{t+1} + \gamma\nu_*(\mathsf{S}_{t+1})|\mathsf{S}_t = s, \mathsf{A}_t = a] && (2.8)\\ &= \mathbb{E}_{\pi_*}[\mathsf{R}_{t+1} + \gamma \max_{a'} Q_*(\mathsf{S}_{t+1}, a')|\mathsf{S}_t = s, \mathsf{A}_t = a]. && (2.9) \end{aligned}$$

Furthermore, even though there might exist several optimal policies, the $Q_*$ values are unique as stated in [4].

The Q-learning algorithm, defined by

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)], \tag{2.10}$$

performs temporal difference (TD) updates to state-action pairs with respect to the step-size parameter $\eta$, where $0 \leq \eta \leq 1$. Here, TD updates means that $Q(s_t,a_t)$ is iteratively updated as the agent traverses the states. Furthermore, Q-learning is off-policy in the sense that $Q$ directly approximates $Q_*$ regardless of the policy the agent follows. In Q-learning, the policy only depends on which state-action pair can be traversed as long as the agent can repeatedly update $Q(s_t,a_t)$.

## 2.3 Deep Learning

This chapter provides an introduction to the concepts of deep learning for a more thorough understanding of the neural network used in Section 4.4.6. Unless otherwise specified, formulas and examples are taken from [5].

### 2.3.1 Feedforward Neural Network

A feedforward neural network, often referred to as a multi-layer perceptron, is a network where each neuron in the previous layer is directly connected to a neuron in the next layer. The goal of a feedforward neural network, denoted as

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\Theta}), \tag{2.11}$$

is to approximate a function, $f^*$, by mapping input, $\boldsymbol{x}$, to the category, $\boldsymbol{y}$, and learn the corresponding parameters of the approximation, denoted as $\boldsymbol{\Theta}$.

As the input information, $\boldsymbol{x}$, is passed forward through the network, hence the name feedforward, the series of functions,

$$f(\boldsymbol{x}) = f^3(f^2(f^1(\boldsymbol{x}))), \tag{2.12}$$

approximate the input from the previous function where each function is a layer in the network. If a network is composed of two layers or more, referred to as the depth of the network, it is called a deep neural network, hence the terminology deep learning. The last layer of the network, the output layer, outputs the approximation of $f^*(\boldsymbol{x})$, based on the input $\boldsymbol{x}$, to be as close to Equation (2.12) as possible. With the purpose of making the best approximation, the network evaluates each hidden layer for each training point by learning the best method to approximate $f^*(\boldsymbol{x})$.

### 2.3.2 Linear Neural Network

A linear neural network is a feedforward neural network, denoted as

$$\hat{\boldsymbol{y}} = \boldsymbol{\Theta}\boldsymbol{x} + \boldsymbol{b}, \tag{2.13}$$

where each layer in the network is a linear function that takes a vector, $\boldsymbol{x} \in \mathbb{R}^n$, as input and multiplies all the weights, $\boldsymbol{\Theta} \in \mathbb{R}^{m \times n}$, together with a bias denoted as $\boldsymbol{b}$. The set of weights determines how much value each prediction has in this network, where $0 \leq \theta_{ij} \leq 1$. If the value of the weight, $\theta_{ij}$, is increased, the network will favor the corresponding feature, $x_i$, when predicting $\hat{\boldsymbol{y}}$. On the other hand, if $\theta_{ij} = 0$ the corresponding feature is not considered. A linear function is derived from the linear regression formula, which does not include a bias.

By adding a bias to each layer, referred to as an affine transformation, the mapping from features to predictions is converted from a linear function to an affine function and does not have to pass through the origin of a graph. The bias can be considered as an additional set of weights where there is a bias toward $\boldsymbol{b}$ if there is no input to the model. In the end, even though it is an affine function, it is still commonly referred to as a linear function.

### 2.3.3 Activation Function

An activation function is a function used to compute values for each hidden layer in the neural network. There exist multiple different activation functions with a variety of different use cases. The most frequently applied is the rectified linear activation function, denoted as

$$g(z) = \max\{0, z\}, \tag{2.14}$$

and commonly referred to as ReLU. With this activation function, each neuron's output is reduced to zero for negative outputs by selecting the maximum of the input and zero. This selection solves the vanishing gradient problem where learning becomes unstable due to the neural network not being able to minimize the loss function and update the parameters correctly. However, as a result of selecting the maximum of zero and the input, negative outputs of the previous layer will result in the dead neurons problem, where the value of some neurons will remain zero and never be updated.

One solution addressing this issue is leaky rectified linear unit, denoted as

$$g(z) = \max\{0, z\} + \kappa \min\{0, z\}, \tag{2.15}$$

and referred to as leaky ReLU. This activation function controls the angle of negative values with a small slope compared to a flat slope in ReLU and is used in problems suffering from sparse gradients. The parameter for the negative slope, referred to as $\kappa$, is a hyper-parameter set before training initiates, where the default value is set to 0.01.

### 2.3.4 Loss Functions

In neural networks, when the network is fed with input and generates predictions on those inputs, it has to determine how accurate the generated predictions are. The loss function compares the generated predictions to the expected output values. There exist multiple different loss functions, with the most common being mean absolute error (MAE) and mean squared error (MSE). The MAE loss function, denoted as

$$L(\boldsymbol{y}, \boldsymbol{p}) = \frac{1}{M} \sum_{l=1}^{M} |p_l - y_l|, \tag{2.16}$$

is commonly used for regression type problems where the function calculates the absolute difference between the target and the generated prediction, where $M$ is the number of samples from a dataset, $p_l \triangleq p_l(\{\theta_{ij}\}; \mathcal{D})$ is the prediction where $i, j$ range over the number of neurons, $\mathcal{D}$ contains samples of the dataset, and $y_l$ is the target.

The MSE loss function, denoted as

$$L(\boldsymbol{y}, \boldsymbol{p}) = \frac{1}{M} \sum_{l=1}^{M} (p_l - y_l)^2, \tag{2.17}$$

calculates the squared Euclidean distance between the value of the target and the value of the prediction. Similar to the MAE loss function, the MSE loss function is used for regression problems. However, it is susceptible to outliers where predictions from the neural network are not centered around a mean value.

To prevent outliers from having a significant impact on the loss function, the Huber loss, invented by Peter J. Huber [6], can be adopted. The Huber loss function, denoted as

$$L(\boldsymbol{y}, \boldsymbol{p}) = \frac{1}{M} \sum_{l=1}^{M} \begin{cases} \frac{1}{2}(p_l - y_l)^2 & \text{if } |p_l - y_l| < \delta, \\ \delta(|p_l - y_l \mid -\frac{\delta}{2}) & \text{otherwise,} \end{cases} \tag{2.18}$$

uses a combination of MSE and MAE to calculate the loss between the target and the generated predictions. By utilizing a hyper-parameter, $\delta$, the Huber loss function can apply the strengths of both MAE and MSE where both loss functions are differentiable. This ensures that the loss is not susceptible to outliers and provides further smoothness.

### 2.3.5 Backpropagation

After calculating the loss of the neural network's prediction using a loss function described in the previous section, the neural network now aims to minimize the loss function by updating its weights and biases using, e.g., gradient descent (as described next in Section 2.3.6). Hence, the partial derivatives of the loss function with respect to the weights of the network need to be calculated. This is done using the chain rule and iterating through the network backward, hence the name backpropagation. Backpropagation was first presented by Rumelhart *et al.* in [7] and works as follows.

The network's output for each neuron, $o_k$, is calculated according to

$$o_k = g(w_k) = g\left(\sum_k \theta_{hk} o_h\right), \tag{2.19}$$

where $k$ is the index for the neurons in the preceding layer,

$$w_k \triangleq \sum_h \theta_{hk} o_h \tag{2.20}$$

is the weighted sum of the outputs from all the neurons in the preceding layer, $\theta_{hk}$ is the weight between the $h$-th neuron of the previous layer and the $k$-th neuron of the present layer, and $g(\cdot)$ is the activation function, e.g., the ReLU activation function.

To identify the change, we use the chain rule and calculate the change in a particular weight according to

$$\frac{\partial L}{\partial \theta_{ik}} = \frac{\partial L}{\partial o_k}\frac{\partial o_k}{\partial \theta_{ik}} = \frac{\partial L}{\partial o_k}\frac{\partial o_k}{\partial w_k}\frac{\partial w_k}{\partial \theta_{ik}}. \tag{2.21}$$

From Equation (2.19), we can calculate the term $\frac{\partial o_k}{\partial w_k}$ as

$$\frac{\partial o_k}{\partial w_k} = \frac{\partial g(w_k)}{\partial w_k}, \tag{2.22}$$

and

$$\frac{\partial w_k}{\partial \theta_{ik}} = \frac{\partial}{\partial \theta_{ik}} \sum_h \theta_{hk} o_h = o_i \tag{2.23}$$

according to (2.20). Now, we introduce a useful notation,

$$\delta_k \triangleq -\frac{\partial L}{\partial w_k} = -\frac{\partial L}{\partial o_k}\frac{\partial o_k}{\partial w_k}. \tag{2.24}$$

Substituting Equation (2.23) and Equation (2.25) in Equation (2.21), we obtain

$$\frac{\partial L}{\partial \theta_{ik}} = -\delta_k o_i, \tag{2.25}$$

which suggests that the weight $\theta_{ik}$ can be updated according to $\theta_{ik} \leftarrow \theta_{ik} + \Delta\theta_{ik}$ with

$$\Delta\theta_{ik} = \eta\delta_k o_i, \tag{2.26}$$

where $\eta$ is the learning rate set to $0 \leq \eta \leq 1$.

To compute $\delta_k$, we consider two cases. First, if the $k$-th neuron is an output neuron in the last layer, then we have $o_k = p_k$, and from Equation (2.24) $\delta_k$ becomes

$$\delta_k = -\frac{\partial L}{\partial o_k}\frac{\partial o_k}{\partial w_k} = -\frac{\partial L}{\partial p_k}\frac{\partial g(w_k)}{\partial w_k}. \tag{2.27}$$

Otherwise, if the $k$-th neuron is not an output neuron in the last layer, the chain rule is utilized to iterate backward through each layer in the network using the recursive formula

$$\delta_k = -\frac{\partial L}{\partial o_k}\frac{\partial o_k}{\partial w_k} = -\left(\sum_j \theta_{kj}\delta_j\right)\frac{\partial g(w_k)}{\partial w_k}, \tag{2.28}$$

where $j$ is the index for the neurons in the next layer associated with the $k$-th neuron of the current layer.

### 2.3.6 Gradient Descent

Gradient descent is a form of gradient-based learning in which the neural network learns by stepping toward a local minimum. There exist several different extensions of the gradient descent algorithm, with the most common being stochastic gradient descent (SGD), batch gradient descent (BGD), and mini-batch gradient descent (MGD). Every gradient-based learning algorithm aims to minimize the loss function by first doing backpropagation and finding the slope, and then stepping toward the minimum. For example, for a given linear neural network, $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\Theta})$, a gradient descent algorithm aims to discover the quickest direction $f$ is decreasing. This can be achieved by taking the negative of each gradient in the network and reducing $f$ in the direction of the negative gradient according to

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta\nabla_{\boldsymbol{x}} f(\boldsymbol{x}), \eta > 0, \tag{2.29}$$

and is performed at every training step. Here, $\eta$ is the learning parameter and is used to specify the step size, $t$ is a given training step, and $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ is the gradient of all the partial derivatives of $f(\boldsymbol{x})$ comprised as a vector. Even though all the different versions of gradient descent aim to find the local minimum, there are some key differences between the gradient descent extensions.

BGD utilizes the whole dataset for gradient descent, making it computationally expensive for large datasets. On the other hand, SGD is computationally efficient as it only makes use of a single sample from the dataset to perform gradient descent. However, due to it being a single sample only, it will often lead to a poor estimation of the gradient. A good combination between BGD and SGD is MGD. MGD performs gradient descent on a mini-batch of samples from the dataset. For each training step, a mini-batch of samples of size $M$ is uniformly selected from the training set. The sample size is a hyper-parameter set before training initiates and held unchangeable throughout the training process. Note that we use $M$ as the size of a mini-batch for the rest of the thesis.

By utilizing the MGD algorithm, the gradient can be updated according to

$$\boldsymbol{G}_t = \nabla_{\boldsymbol{\Theta}} L(\boldsymbol{y}, \boldsymbol{p}), \tag{2.30}$$

where $L$ is some loss function where the prediction $\boldsymbol{p}$ depends on the sampled mini-batch and the weights $\boldsymbol{\Theta}$.

After the gradient update, it will step toward a minimum according to

$$\boldsymbol{\Theta}_{t+1} \leftarrow \boldsymbol{\Theta}_t - \epsilon \boldsymbol{G}_t. \tag{2.31}$$

MGD is often preferred over BGD and SGD due to higher performance and accurate estimation of gradients. The algorithm may fit a larger dataset containing millions of examples for a batch size of a few hundred examples. While MGD has many good properties, it is essential that sampling is uniformly random. If samples are highly correlated, it may cause the neural network to generalize poorly and overfit. Furthermore, for the algorithm to be effective, it must be able to compute the gradients with an unbiased estimate from the sample. Additionally, for smaller datasets contrary to larger datasets, two succeeding mini-batch samples must also be independent of each other for two succeeding gradient estimates to be distinct.

### 2.3.7 Optimization

As previously mentioned, a machine learning algorithm aims to minimize the loss function by calculating the derivative of the gradients and stepping toward a local minimum. The minimization of the loss function for neural networks is considered to be a non-convex optimization problem with multiple local minimum points. The network cannot determine if a local minimum is a global minimum, making optimization difficult compared to convex optimization with a single global minimum. Another issue with non-convex optimization is that the network might become stuck at a saddle point when performing gradient descent. Since gradient descent performs small local steps toward a minimum, it might become stuck at a local minimum point surrounded by steep cliffs. Furthermore, since there is a limited correlation between the local and global structure of the network, finding a global minimum is deemed to be complicated.

Even though optimization in neural networks is considered incredibly challenging, some methods exist to increase learning acceleration. One such method is the momentum algorithm. This algorithm introduces an additional parameter, $\mu$, to adjust the parameters through direction and speed. The main goal of momentum is to solve poor conditioning and variance in the gradient. By setting momentum to be the exponential decaying average of the negative gradient in combination with the step size, $\eta$, the algorithm can determine the speed at which the previous gradients decay according to

$$\boldsymbol{V}_t \leftarrow \mu \boldsymbol{V}_t - \eta \nabla_{\boldsymbol{\Theta}} L(\boldsymbol{y}, \boldsymbol{p}). \tag{2.32}$$

After calculating the momentum, the parameter update can be calculated according to

$$\boldsymbol{\Theta}_{t+1} \leftarrow \boldsymbol{\Theta}_t + \boldsymbol{V}_t. \tag{2.33}$$

The step size, $\eta$, has a high correlation to the $\mu$ where a higher $\eta$ than $\mu$ causes an additional number of the previous gradients to influence the current direction.

After momentum was introduced to optimize neural networks, several optimization algorithms have been developed to include momentum. One popular optimization algorithm often utilized for non-convex optimization with momentum is the root mean squared propagation (RMSProp) algorithm. The RMSProp algorithm utilizes an exponentially decaying average similar to momentum. However, instead of determining the speed, the algorithm diminishes the previous extreme gradients for quick convergence by taking the mean squared of the gradients moving average, first by calculating the accumulated squared gradient element-wise as

$$\boldsymbol{M}_t = \boldsymbol{G}_t \odot \boldsymbol{G}_t, \tag{2.34}$$

and then the parameters according to

$$\Delta\boldsymbol{\Theta}_t \leftarrow -\frac{\mu}{\sqrt{\delta + \boldsymbol{M}_t}} \odot \boldsymbol{G}_t, \tag{2.35}$$

with $\boldsymbol{G}_t$ being the gradient, $\delta$ being a very small number for stable division, and $\odot$ meaning that $\frac{1}{\sqrt{\delta+\boldsymbol{M}_t}}$ is applied to $\boldsymbol{G}_t$ element-wise. Note that in this chapter division, square, and multiplication with matrices is done element-wise. Instead of a single fixed learning rate for weight updates, RMSProp will dynamically update the weights by the magnitude of the moving average according to

$$\boldsymbol{\Theta}_{t+1} \leftarrow \boldsymbol{\Theta}_t + \Delta\boldsymbol{\Theta}_t. \tag{2.36}$$

Another common optimization algorithm for neural networks is the adaptive moments estimation (Adam) optimizer introduced by Kingma and Ba in [8]. This stochastic optimization algorithm is derived from RMSProp and momentum. However, there are some key implementation differences. Adam utilizes momentum for direct approximation of the gradient compared to the mean square in RMSProp. Furthermore, Adam introduces bias calculations for the first and second-order moments, where the first moment is referred to as the mean and the second referred to as the uncentered variance. The bias for the first moment is calculated according to

$$\boldsymbol{B}_{t+1} \leftarrow p_1\boldsymbol{B}_t + (1-p_1)\boldsymbol{G}_t, \tag{2.37}$$

while the second moment is calculated according to

$$\boldsymbol{M}_{t+1} \leftarrow p_2\boldsymbol{M}_t + (1-p_2)\boldsymbol{G}_t \odot \boldsymbol{G}_t. \tag{2.38}$$

The correct bias approximation for the first moment can be calculated according to

$$\widehat{\boldsymbol{B}}_t \leftarrow \frac{\boldsymbol{B}_t}{1-p_1^t}, \tag{2.39}$$

with the second moment according to

$$\widehat{\boldsymbol{M}}_t \leftarrow \frac{\boldsymbol{M}_t}{1-p_2^t}. \tag{2.40}$$

After the bias approximations are calculated, the weight change is calculated according to Equation (2.36) with

$$\Delta\boldsymbol{\Theta}_t = -\eta\frac{\widehat{\boldsymbol{B}}_t}{\sqrt{\widehat{\boldsymbol{M}}_t + \delta}}. \tag{2.41}$$

By introducing bias approximations, Adam is considered a powerful tool for different hyper-parameters, where $p_1$ and $p_2$ are universally set to be between 0.9 and 0.999. Both $\eta$ and $\mu$ are commonly set to be around 0.001.

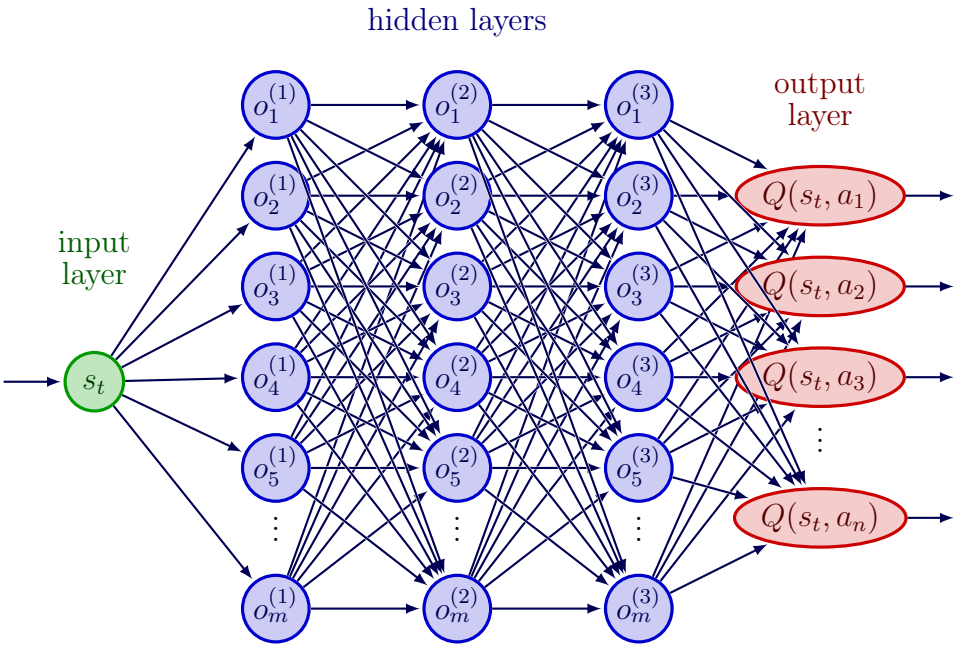


Figure 2.3: A DQN-model.

## 2.4 Deep Reinforcement Learning

This section provides an introduction to the concepts of DRL for a more thorough understanding of the strategy used in Section 4.4.6. Unless otherwise specified, formulas and examples are taken from [9] and [10].

### 2.4.1 Approximation of $Q(s, a)$

A deep Q-network (DQN) was first introduced by Mnih *et al.* in [9] to train an agent to play Atari games. While $Q(s, a)$ is updated iteratively in RL through TD updates, Mnih *et al.* proposed to use a neural network to approximate the action-value function $Q(s, a; \mathbf{\Theta}) \approx Q_*(s, a)$. Here, $\mathbf{\Theta}$ is the network's weights and is referred to as a Q-network. A DQN is trained by minimizing

$$\mathbb{E}_{\mathsf{S}_t, \mathsf{A}_t}\left[(y_t - Q(\mathsf{S}_t, \mathsf{A}_t; \mathbf{\Theta}))^2\right] \tag{2.42}$$

for each training round $t$. Here,

$$y_t = \mathbb{E}_{\mathsf{R}_{t+1}, \mathsf{S}_{t+1}}[\mathsf{R}_{t+1} + \gamma \max_a Q(S_{t+1}, a; \mathbf{\Theta}) | \mathsf{S}_t = s_t, \mathsf{A}_t = a_t] \tag{2.43}$$

is the target vector, while following an $\epsilon$-*greedy strategy*, described as follows. The value of $\epsilon$ will decrease toward a greedy action selection at an episodic level, and is a hyper-parameter set before training initiates. The agent will begin by selecting a random action with probability $\epsilon$ before selecting a greedy action with probability $1 - \epsilon$ as it continues to explore. By taking advantage of a neural network, DQN can minimize the loss function by utilizing MGD for each step the agent performs. The agent's state is transmitted through the network as input for training before the action with the highest Q-value is selected according to

$$a_t = \arg\max_a Q(s_t, a; \mathbf{\Theta}). \tag{2.44}$$

This is portrayed in Figure 2.3 where the agent's state is sent through the network to approximate the action that yields the highest Q-value. The outputs of the neurons of the hidden layers, denoted by $o_i^{(j)}$, and $m$ being the number of neurons for each hidden layer, aim to approximate the action yielding the highest $Q(s_t, a)$.

### 2.4.2 Experience Replay

Different from classical RL, where the agent's experiences are often stored in a $Q_{\text{table}}$, a technique called experience replay is utilized in DQN. Experience replay is a replay buffer with a fixed length $b$, $\mathcal{E} = \left\{e_{t^{(1)}}, e_{t^{(2)}}, \ldots, e_{t^{(b)}}\right\}$, that stores the agent's interactions with the environment. An experience at a single time step $t$ is denoted as $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. Here, we store the agent's current state, $s_t$, the agent's action, $a_t$, the reward, $r_{t+1}$, and the next state, $s_{t+1}$. During training, we sample a random mini-batch from the buffer to update the Q-values for greater data efficiency and to reduce the variance of updates. Random mini-batch sampling is preferred over iterative sampling due to a high correlation between states that may lead to inefficient learning. Furthermore, since each experience $e_t$ will be used for numerous weight updates in the neural network, the network may encourage certain actions over others that may result in poor convergence. If this occurs, the agent will be trapped in a bad local minimum and believe it has found an optimal solution. In the end, for environments with state spaces exceeding the size of the replay buffer, the agent overrides the existing experiences in the replay buffer with new experiences.

### 2.4.3 Stable Training

In [10], Mnih *et al.* introduced a target network to DQN for more stable convergence. The target network, referred to as $\widehat{Q}$ with weights $\boldsymbol{\Theta}^-$, is another instance of the already existing neural network, $Q$, used to produce the target according to

$$\hat{y}_t = \mathbb{E}_{\mathsf{R}_{t+1}, \mathsf{S}_{t+1}}[\mathsf{R}_{t+1} + \gamma \max_a \widehat{Q}(\mathsf{S}_{t+1}, a; \boldsymbol{\Theta}^-) | \mathsf{S}_t = s_t, \mathsf{A}_t = a_t]. \tag{2.45}$$

The network duplicates the weights of the Q-network to the target network every $C$ updates in order to produce the target $\hat{y}_t$. Here, the parameter $C$ is a hyper-parameter set before training initiates. By utilizing a target network, the algorithm will become more stable whenever an update occurs in $Q(s_t, a)$. It may also increase $Q(s_{t+1}, a)$ for every action. Furthermore, as a result of an increase in $Q(s_{t+1}, a)$, the target will also improve, leading to more consistent convergence. In the end, the update parameter, $C$, has to be carefully selected through hyper-parameter testing. If the network updates too often, parameters duplicated from the Q-network to the target network, the agent may diverge to a worse policy by following a newly updated target $\hat{y}_t$ despite the fact that the previous policy might have yielded better results.

The last modification proposed in Mnih *et al.* was a clipping of

$$r_{t+1} + \gamma \max_a \widehat{Q}(s_{t+1}, a; \boldsymbol{\Theta}^-) - Q(s_t, a_t; \boldsymbol{\Theta}) \tag{2.46}$$

to be in the range of $(-1, 1)$ for given $\mathsf{R}_{t+1} = r_{t+1}$, $\mathsf{S}_{t+1} = s_{t+1}$, $\mathsf{S}_t = s_t$, and $\mathsf{A}_t = a_t$. Here, clipping forces the network not to consider large errors that may

cause divergence, resulting in further improvement of the stability of the algorithm. For values produced by the loss function, the derivative of the absolute values will be $-1$ if the produced value is negative and 1 for positive values.

## 2.5 Lagrange Multipliers

This section provides an introduction to Lagrange multipliers for a more thorough understanding of the strategy used in Section 4.4.4 to solve the optimization problem. Unless otherwise specified, formulas and examples are taken from [11].

### 2.5.1 Introduction to Lagrange Multipliers

*Lagrange multipliers* are a mathematical optimization method on a function with $m$ variables to find the minimum or maximum with $k$ constraints. For example, for a given function with multiple variables, $f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$, and the constraints $c_1(\boldsymbol{x}), c_2(\boldsymbol{x}), \ldots, c_k(\boldsymbol{x})$ where $c_j(\boldsymbol{x}) = 0$, $j = 1, 2 \ldots, k$, we can define the Lagrangian function as

$$\mathcal{L}(\boldsymbol{x}, \lambda_1, \ldots, \lambda_k) = f(\boldsymbol{x}) + \sum_{j=1}^{k} \lambda_j c_j(\boldsymbol{x}). \tag{2.47}$$

We can calculate the partial derivatives as

$$\frac{\partial \mathcal{L}}{\partial x_i} = 0, \quad i = 1, 2 \ldots, m \tag{2.48}$$

and

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = c_j(\boldsymbol{x}) = 0, \quad j = 1, 2 \ldots, k. \tag{2.49}$$

Lagrange multipliers are commonly used to find the minimum or maximum inside of a region, where the region is defined by the constraints.

# Chapter 3

# Related Work

This chapter aims to provide a better overview of related work in the field of MEC, specifically the use of RL, DRL, and different variants of DRL techniques. Each paper will be described in detail with the proposed algorithm, simulation methods, and results.

## 3.1 Reinforcement Learning

In [12], Wang *et al.* propose to utilize a Q-learning-based mobility management scheme (QPI) to train an agent to find the optimal mobility management strategy through trial and error. The paper's objective is to train an agent to find the best decisions in an environment with numerous obstacles. The users in the system are first connected to the base station with the highest received signal strength (RSS). After this, the users move between different base stations using an $\epsilon$-greedy policy to find the base station with the lowest task delay. Every time a user decides to switch base stations, the Q-value must be calculated and updated. For simulation, Wang *et al.* compare the performance of the QPI algorithm to local computation, computation at a random base station, and at the base station with the highest RSS. The proposed QPI algorithm can efficiently solve computation-intensive tasks with a lower delay than the other strategies. In the end, through Q-learning, the users can improve their decision-making in challenging environments.

In [13], Hao *et al.* implemented a cognitive learning-based computation offloading (CLCO) algorithm to find the optimal policy of offloading tasks in a multi-edge and multi-user environment. The CLCO algorithm calculates pre-computation offloading for each task in advance if the mobile device is in an idle state. After this, the CLCO algorithm is utilized to optimize the offloading process when a device is not in an idle state. For simulation, Hao *et al.* consider 5 edge servers with 300 mobile devices attached. The results show that the CLCO algorithm has better performance for task duration and larger data sizes than local and edge computation. In the end, the algorithm is also capable of solving more extensive offloading computations more rapidly compared to random and uniform offloading.

## 3.2 Deep Reinforcement Learning

In [14], Zhang *et al.* utilize a DQN to find optimal offloading schemes for vehicles in the heterogeneous vehicular network with multiple edge servers. The network is trained on real-life traffic data with 1.4 billion GPS traces for 14000 taxis over a 3 week period. Furthermore, three heavily trafficked streets in China were chosen based on traffic information from Google maps. For performance evaluation of the DQN, other strategies such as the best transmission path, best MEC server, greedy algorithm, and a game-theoretic approach are utilized. In the end, the DQN performed better as an offloading utility for rush hour traffic and traffic density compared to the other utilized strategies.

In [15], Huang *et al.* implemented a joint task offloading and bandwidth allocation (JTOBA) algorithm as a DQN to find a good offloading scheme for multiple users. The JTOBA algorithm is compared to other strategies such as local only, edge only, greedy, and the MUMTO algorithm, which is a multi-user multi-edge algorithm proposed in [16]. For simulation, Huang *et al.* keep the total number of users in the system fixed at 5 with 4 tasks per user. The total cost of each strategy is simulated with an increased task size. The greedy strategy has the lowest cost compared to JTOBA, MUMTO, edge, and local. However, the difference between greedy and JTOBA is minimal compared to the other strategies. In the end, Huang *et al.* compare the total cost performance of the JTOBA algorithm with respect to different batch sizes, learning rates, and increased task sizes. The algorithm's performance is overall improved with a larger batch size, lower learning rate, and a smaller task size.

In [17], Chen *et al.* utilize a DQN to find an optimal offloading scheme in a multi-edge system. The different strategies implemented for comparison are local, edge, and greedy, where the greedy strategy selects the minimum cost between local and edge computation. For simulation, the number of base stations in the system is set to 6. First, the average cost of the proposed DQN with different numbers of layers and neurons is presented. Here, the average cost stops increasing after 4 layers and 512 neurons. After this, the total cost of all the strategies is presented with energy unit arrival rates of 0.3 and 0.5, where the proposed DQN yielded a lower total cost than the other strategies. Furthermore, the proposed DQN also had fewer task drops between the different base stations with an increased energy unit arrival rate, in addition to fewer handovers of tasks between the different base stations for an increased energy unit arrival rate.

In [18], Li *et al.* utilize a DQN for task offloading via unmanned aerial vehicle (UAV) based MEC. In a UAV-based system, the UAV acts as an edge server in close proximity to users for reduced latency, where each user in the system offloads via the UAV. However, the UAV can only serve one user at a time if the location of the UAV is at a given access point due to limited computational capacity and energy. For simulation, the number of users is set to 15 with 25 fixed access points for communication between a user and the UAV. Furthermore, each user is randomly scattered within these access points, while the height of the UAV is 100 meters. The only utilized strategies in this paper are the proposed DQN and Q-learning. For the results, both strategies are tested with 3 different battery levels for the UAV. The DQN with the highest UAV battery level yielded the best possible return over 800 training episodes, compared to the DQN with the second and third highest

battery levels. The Q-learning strategy with the highest battery level had a worse average return compared to the DQN with the lowest battery level. In contrast, the second and third highest battery levels for Q-learning performed even worse. The last simulation, which shows the relation between throughput and the number of training episodes for different battery levels, shows that a higher battery level for the UAV yields a higher throughput for each user in the network.

## 3.3 Variants of Deep Reinforcement Learning

In this section, other variants of DQN for MEC will be explained together with their proposed algorithm and results.

In [19], Lu *et al.* implement the multi-agent deep deterministic policy gradient (MADDPG) algorithm in combination with the soft actor-critic (SAC) algorithm to find the best offloading scheme for a continuous action space. The MADDPG algorithm is an extension of the DDPG algorithm that can train multiple agents simultaneously. In contrast, the SAC algorithm is an improved actor-critic algorithm for maximized entropy RL. The paper's objective is to prove that the MADDPG algorithm, in combination with the SAC algorithm, has good convergence and is capable of discovering an offloading policy for multiple devices in a continuous action space. The MADDPG+SAC is compared to normal MDDPG, SAC, DDPG, edge, and mobile strategies. For simulation, Lu *et al.* select information about edge servers located in Melbourne from the EUA dataset. The proposed MADDPG+SAC algorithm yields a higher average reward for 1000 training episodes compared to MADDPG, DDPG, and SAC. For the next set of results with up to 1000 mobile devices, the edge strategy yielded the best energy consumption while MADDPG+SAC had the second-lowest. For total cost, the mobile strategy yielded the lowest cost, with DDPG giving the second-lowest. Overall, MADDPG+SAC yielded the best results for latency and task drop rate for up to 1000 devices but had poor overall performance for total cost and energy compared to the other strategies.

In [20], He *et al.* utilize a double-dueling-deep Q-network for improved networking, caching, and computing in the vehicular network that is formulated as a joint optimization problem. The proposed algorithm is the only strategy implemented in the paper other than the existing static scheme. However, He *et al.* have implemented different versions of the algorithm. One without edge caching, another without MEC offloading, and one without virtualization. For simulation, 5 base stations with an MEC server connected and 1 mobile virtual private network. Both the base stations and vehicles are randomly scattered in close proximity to each other. The first result shows the convergence of the total utilities with respect to the number of training episodes. Here, the proposed algorithm has significantly higher total utility than the other versions, where the algorithm without virtualization yields the worst results. The second result, which shows the total utility with an increased content size, shows that the proposed algorithm is better equipped to handle increased content size compared to the other versions. For the remaining results, which describe the total utility with an increased charging price for accessing the base stations, increased charging price for MEC offloading, and increased charging price for accessing the cache server, the proposed algorithm yielded the highest total utility for every result. On the contrary, the existing static scheme yielded the worst overall results.

# Chapter 4

# Methodology and Model Architecture

This chapter presents the strategies used in this thesis. First, the problem formulation will be introduced along with the computation model. Each strategy will be described in detail with model design and training algorithm implemented. Unless otherwise specified, formulas and expressions are taken from [21] and [22]. In Chapter 5, the results and findings of each strategy will be presented and further discussed.

## 4.1 Problem Formulation

We will begin with the assumption that there is a group of devices, denoted as

$$\mathcal{N} = \{1, 2, 3, \ldots, N\}, \tag{4.1}$$

which have some resource-intensive tasks they seek to complete either locally or through offloading to an MEC server with limited resources. Furthermore, we consider that there exists a base station with an MEC server connected that the devices interact with through wireless communication, as illustrated in Figure 4.1. Additionally, if several devices offload simultaneously, the bandwidth, denoted as $W$, will be shared equally.

The task of each device is denoted as

$$R_i = (B_i, D_i), \quad i \in \mathcal{N}. \tag{4.2}$$

Parameter $B_i$ is the size of the computation (in kilobytes) the device wants to solve. The other parameter, $D_i$, is the total number of CPU cycles required to complete the task. Both $D_i$ and $B_i$ are positively related, where $D_i$ remains unchanged throughout the computation process. In the end, the task parameters are scheduled through task profiles from an application and can vary between different applications.

In our system, we assume that each device has a task it wants to solve through local computation or offloading to the MEC server. We denote the choice the device $i$ has made as $\alpha_i = 0$ for local computation and $\alpha_i = 1$ for offloading. For multiple devices in the system, we define the action vector for the devices offloading decision as $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_N]$. Furthermore, for the sake of simplicity, we assume that the task has to be solved as a whole and cannot be divided into multiple segments.

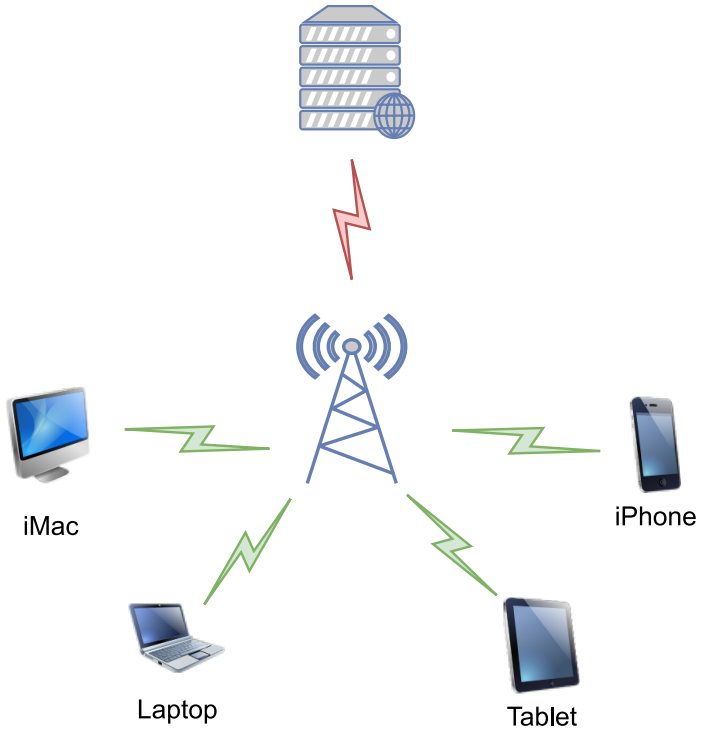


Figure 4.1: MEC network model.

## 4.2 Computation Model

This section describes the formulas used to calculate energy, delay, and cost used in the different strategies.

### 4.2.1 Local Computing Model

The local computing model is used when a device decides to execute a task locally. The local execution delay, denoted as

$$T_i^{\mathrm{l}} = \frac{D_i}{f_i^{\mathrm{l}}}, \tag{4.3}$$

is composed of the total number of CPU cycles, $D_i$, and the computational capacity of the CPU denoted as $f_i^{\mathrm{l}}$. Essentially, $T_i^{\mathrm{l}}$ determines how long it will take the device to finish $R_i$, which can differ between devices based on the computing power of the local CPU.

The energy used to finish $R_i$ is calculated according to

$$E_i^{\mathrm{l}} = z_i D_i. \tag{4.4}$$

The formula for energy consumption is composed of the energy consumption per CPU cycle, denoted as $z_i$, and the total number of CPU cycles needed. For simplicity, the energy consumption per CPU cycle is set to $z_i = 10^{-27}(f_i^{\mathrm{l}})^2$ from [23].

The total cost of local computing can be calculated according to

$$C_i^{\mathrm{l}} = I_i^{\mathrm{t}} T_i^{\mathrm{l}} + I_i^{\mathrm{e}} E_i^{\mathrm{l}}, \tag{4.5}$$

and is derived from Equation (4.3) and Equation (4.4). Here, $I_i^{\mathrm{t}}$ and $I_i^{\mathrm{e}}$ stand for the weights of time and energy, and must fulfil $0 \leq I_i^{\mathrm{t}} \leq 1$ and $0 \leq I_i^{\mathrm{e}} \leq 1$ in addition to $I_i^{\mathrm{e}} + I_i^{\mathrm{t}} = 1$. Furthermore, as the weights may be different for each task, let us assume that the weights remain unchanged throughout the computation process.

### 4.2.2 Offloading Model

The offloading computation model is used when a device chooses to offload a task through wireless communication with the MEC server. First, the upload rate for the devices is calculated according to

$$r_i = \frac{W}{K} \log\left(1 + \frac{P_i h_i}{\frac{W}{K} N_0}\right). \tag{4.6}$$

The first parameter is the bandwidth, W, shared between all the $K$ devices in the system that choose to offload, $P_i$ is the transmission power, $N_0$ is the variance of complex white Gaussian channel noise, and $h_i$ is the channel gain for the wireless channel calculated as $\frac{1}{d_i^2}$ where $d_i$ is the distance from the device to the base station.

After the upload rate is calculated, each device will begin to upload input parameters to the base station before the computation task is transmitted to the MEC server. The transmission delay can be computed according to

$$T_{i,\mathrm{t}}^{\mathrm{o}} = \frac{B_i}{r_i}. \tag{4.7}$$

$T_{i,\mathrm{t}}^{\mathrm{o}}$ measures the time it will take to upload the computation task through wireless communication with the MEC server.

After calculating the delay from the device to the server, we can calculate the processing delay of the MEC server according to

$$T_{i,\mathrm{p}}^{\mathrm{o}} = \frac{D_i}{f_i}. \tag{4.8}$$

Here, $T_{i,\mathrm{p}}^{\mathrm{o}}$ is the time the MEC server uses to process and complete the task sent by the device, where $f_i$ is a resource allocation from the MEC server to complete the task on the offloading device's behalf.

When the server has finished processing the computation, the delay of transmitting the result of the computation from the server to the device can be calculated according to

$$T_{i,\mathrm{b}}^{\mathrm{o}} = \frac{B_{\mathrm{b}}}{r_{\mathrm{b}}}. \tag{4.9}$$

The download delay is calculated as the size of the processed computation, $B_{\mathrm{b}}$, where $r_{\mathrm{b}}$ is the download rate from the server to the device. As a result of offloading, the data size of the processed computation will be significantly lower compared to the size of the computation task transmitted by the device. Additionally, the download rate from the server to the device is considerably higher compared to the uplink rate from the device to the server. With both the size of the processed result and the increased download rate, we disregard this process for the sake of simplicity.

In the end, as a combination of Equation (4.7) and Equation (4.8), we can calculate the total delay from offloading to completing the task according to

$$T_i^{\mathrm{o}} = \frac{B_i}{r_i} + \frac{D_i}{f_i}. \tag{4.10}$$

The energy consumption from offloading the computation to the MEC server can be calculated according to

$$E_{i,\mathrm{p}}^{\mathrm{o}} = P_i T_{i,\mathrm{p}}^{\mathrm{o}} = \frac{P_i B_i}{r_i}. \tag{4.11}$$

Here, $P_i$ is defined as the transmission power, where $E_{i,\mathrm{p}}^{\mathrm{o}}$ determines how much energy is used to send the computation task to the MEC server.

After calculating the energy used to transmit the computation task, we can now calculate the energy used when the device is in an idle state according to

$$E_{i,\mathrm{p}}^{\mathrm{o}} = P_i^{\mathrm{s}} T_{i,\mathrm{p}}^{\mathrm{o}} = \frac{P_i^{\mathrm{s}} D_i}{f_i}, \tag{4.12}$$

where $P_i^{\mathrm{s}}$ is defined as the power consumption used by the device when idle. $E_{i,\mathrm{p}}^{\mathrm{o}}$ specifies how much energy is used to solve the computation.

In the end, using a combination of Equation (4.11) and Equation (4.12), we can calculate the total energy consumption according to

$$E_i^{\mathrm{o}} = \frac{P_i B_i}{r_i} + \frac{P_i^{\mathrm{s}} D_i}{f_i} \tag{4.13}$$

for the whole offloading process. The total cost can be calculated according to

$$C_i^{\mathrm{o}} = I_i^{\mathrm{t}} T_i^{\mathrm{o}} + I_i^{\mathrm{e}} E_i^{\mathrm{o}}, \tag{4.14}$$

similar to the cost of local computing and with the same set of weights.

## 4.3 Optimization Problem

This section provides a deeper mathematical description of the optimization problem and corresponding constraints.

### 4.3.1 Constraints

We begin by defining the total cost for all devices in the system according to

$$C_{\mathrm{total}} = \sum_{i=1}^{N} (1 - \alpha_i) C_i^{\mathrm{l}} + \alpha_i C_i^{\mathrm{o}}, \tag{4.15}$$

and the corresponding action as $\alpha_i \in \{0, 1\}$. If the device $i$ decides to offload the task, $R_i$, to the MEC server, we set $\alpha_i = 1$. On the other hand, if the device chooses local computing, we set $\alpha_i = 0$.

By utilizing Equation (4.15) we can define the offloading process as an optimization problem. The objective is to find the optimal action vector, $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_N]$, and resource allocation vector, $\boldsymbol{f} = [f_1, f_2, \ldots, f_N]$, to minimize the total cost of the

entire system as a combination of execution delay and energy consumption according to

$$\min_{\boldsymbol{\alpha},\boldsymbol{f}} \sum_{i=1}^{N} (1-\alpha_i) C_i^{\mathrm{l}} + \alpha_i C_i^{\mathrm{o}}. \tag{4.16}$$

Since the action vector is a vector of binaries that increases rapidly as more devices join, the optimization problem can be described as mixed integer non-linear programming [24].

To find the optimal action and resource allocation vectors, we first define the constraints of the system as:

$$\mathrm{C1} : \alpha_i \in \{0,1\}, \qquad \forall i \in \mathcal{N} \tag{4.17}$$

$$\mathrm{C2} : 0 \leq f_i \leq \alpha_i F, \qquad \forall i \in \mathcal{N} \tag{4.18}$$

$$\mathrm{C3} : \sum_{i=1}^{N} \alpha_i f_i \leq F, \qquad \forall i \in \mathcal{N} \tag{4.19}$$

The first constraint, C1, refers to the decisions the devices can choose between, where each device only can decide between offloading the task as a whole using the offloading computing model or executing the task locally using the local computing model. C2 prohibits the MEC server from allocating more than the total capacity to an offloading device, while C3 states that the sum of resource allocations to offloading devices should not exceed the total capacity of the MEC server. In the end, both [21] and [22] have an additional constraint describing the maximum tolerable delay of an offloading task, where [21] uses the maximum tolerable delay to limit the number of states for their RL and DRL model. However, in this thesis, we aim to explore as many states as possible. Therefore, we neglect introducing a maximum tolerable delay for an offloading task.

## 4.4 Strategies

This section provides a thorough description of implemented strategies in this thesis. Corresponding algorithms will be described in detail with important definitions and theory.

### 4.4.1 Full Local

The full local strategy, Algorithm 1, initializes each device's action to be set to local computation, $\alpha_i = 0$, such that the action vector is $\boldsymbol{\alpha} = [0, 0, \ldots, 0]$, and the total cost of the system is set to $C_{\text{total}} = 0$. After the action vector initialization phase, corresponding parameters such as the size of computation, $B_i$, and the number of CPU cycles, $D_i$, are given as input along with local computation power, $f_i^{\mathrm{l}}$, and the decision weights $I_i^{\mathrm{t}}$ and $I_i^{\mathrm{e}}$.

We begin by calculating the delay, Equation (4.3), the energy consumption, Equation (4.4), and the cost, Equation (4.5), for each device in the system. In the end, the cost for each device is added together with the calculated cost of previous devices. The result is the total cost of $N$ devices in the system only performing local computation.

**Algorithm 1:** Full Local

**Input:** $\mathcal{N}$, $B_i$, $D_i$, $f_i^{\mathrm{l}}$, $z_i$, $I_i^{\mathrm{t}}$, $I_i^{\mathrm{e}}$
**Output:** $C_{\text{total}}$
$C_{\text{total}} \leftarrow 0$ ▷ Set total cost of system
**for** $i \in \mathcal{N}$ **do**
  $T_i^{\mathrm{l}} \leftarrow \frac{D_i}{f_i^{\mathrm{l}}}$ ▷ Calculate delay
  $E_i^{\mathrm{l}} \leftarrow z_i D_i$ ▷ Calculate energy consumption
  $C_i^{\mathrm{l}} \leftarrow I_i^{\mathrm{t}} T_i^{\mathrm{l}} + I_i^{\mathrm{e}} E_i^{\mathrm{l}}$ ▷ Calculate cost
  $C_{\text{total}} \leftarrow C_{\text{total}} + C_i^{\mathrm{l}}$ ▷ Update total cost
**return** $C_{\text{total}}$

### 4.4.2 Full Offload

The full offload strategy, Algorithm 2, sets each device's action to offload, denoted as $\alpha_i = 1$. The action vector is comprised of $\boldsymbol{\alpha} = [1, 1, \ldots, 1]$ (thus, $K = N$) and the total cost is set to $C_{\text{total}} = 0$. After the devices' action vector has been initialized, parameters such as the size of the computation, $B_i$, the number of CPU cycles, $D_i$, and the local computation power, $f_i^{\mathrm{l}}$, are given as inputs. Furthermore, for offloading to the MEC server, several offloading parameters such as the devices' bandwidth, $W$, the devices' distance to the MEC server, $d$, the transmission power consumption for offloading, $P_i$, and power when idle, $P_i^{\mathrm{s}}$, are also given as inputs along with Gaussian channel noise, $N_0$, and the total server capacity $F$.

The offloading strategy begins by calculating the resource allocations for the total number of devices in the system. Here, each device in the system shares the same amount of resources from the server. The resource allocation for each device is calculated as the server's total capacity divided by the number of offloading devices. After this, the channel gain is calculated for each device based on the distance to the MEC server and the uplink rate according to Equation (4.6). When the uplink rate is calculated, the device's delay is calculated according to Equation (4.10) and the energy consumption according to Equation (4.13). In the end, the cost for each device is added together with the calculated cost of previous devices. The result is the total cost of $N$ devices in the system only performing offloading to the MEC server.

### 4.4.3 Random Search

The random search strategy, Algorithm 3, initializes each device's action to be randomly chosen between local computation or offloading to the MEC server, i.e., $\alpha_i$ is picked uniformly at random from $\{0, 1\}$. The action vector can be comprised in several different ways, for example $\boldsymbol{\alpha} = [0, 0, 1, 1, \ldots, 0]$. Furthermore, the other parameters for MEC are given as inputs similar to the full local and offload strategies, with `numSamples` being the number of samples to generate.

The algorithm starts with initializing the current total cost, $C_{\text{total}}$, to zero and the best total cost, $C_{\text{total}}^{\mathrm{b}}$, to infinity. After this initialization step, we check for the number of offloading devices from the action vector and give corresponding devices resource allocations from the server. The resource allocations are created by giving each offloading device a uniformly random number between zero and one, denoted as

**Algorithm 2:** Full Offload

**Input:** $\mathcal{N}$, $B_i$, $D_i$, $I_i^{\mathrm{t}}$, $I_i^{\mathrm{e}}$, $F$, $W$, $N_0$, $d_i$, $P_i$, $P_i^{\mathrm{s}}$
**Output:** $C_{\text{total}}$
$K \leftarrow N$
$C_{\text{total}} \leftarrow 0$ ▷ Set total cost of system
**for** $i \in \mathcal{N}$ **do**
  $f_i \leftarrow \frac{F}{N}$ ▷ Calculate rescource allocations
  $h_i \leftarrow \frac{1}{d_i^2}$ ▷ Calculate channel gain
  $r_i \leftarrow \frac{W}{K} \log\left(1 + \frac{P_i h_i}{\frac{W}{K} N_0}\right)$ ▷ Calculate uplink rate
  $T_i^{\mathrm{o}} \leftarrow \frac{B_i}{r_i} + \frac{D_i}{f_i}$ ▷ Calculate delay
  $E_i^{\mathrm{o}} \leftarrow \frac{P_i B_i}{r_i} + \frac{P_i^{\mathrm{s}} D_i}{f_i}$ ▷ Calculate energy consumption
  $C_i^{\mathrm{o}} \leftarrow I_i^{\mathrm{t}} T_i^{\mathrm{o}} + I_i^{\mathrm{e}} E_i^{\mathrm{o}}$ ▷ Calculate cost
  $C_{\text{total}} \leftarrow C_{\text{total}} + C_i^{\mathrm{o}}$ ▷ Update total cost
**return** $C_{\text{total}}$

uniform($[0, 1]$). Next, the resource allocations are normalized by taking each device's random number and dividing it by the sum of each device's random number before it is multiplied by the server's total capacity. The normalization step ensures that the resource allocation for each device meets the constraints C1 and C2. On the other hand, if the generated action vector is full local, we set the resource allocation vector to be filled with zeros for the total number of devices in the system.

After the offloading devices have been given a resource allocation from the server, we calculate the total cost according to Algorithm 4. Here, the set of offloading devices is denoted as

$$\mathcal{N}^{\boldsymbol{\alpha}} \triangleq \{i \in \mathcal{N} : \alpha_i = 1\}. \tag{4.20}$$

Algorithm 4 takes an action and a resource allocation vector as inputs, in addition to the MEC parameters from the full local and offload strategies, separates devices for local and offloading computation, and outputs the total cost. After the cost is calculated, the minimum of $C_{\text{total}}^{\mathrm{b}}$ and $C_{\text{total}}$ is selected and $C_{\text{total}}^{\mathrm{b}}$ is updated accordingly. In the end, the algorithm creates multiple different resource allocations and returns the result of the allocation that yields the lowest total cost for the number of samples given.

### 4.4.4 Optimal Solution

The optimal solution strategy exhaustively checks all the possible vectors $\boldsymbol{\alpha}$ and for each of them, uses Lagrange multipliers method to find the minimum total cost according to Algorithm 5.

#### Lagrange Multipliers

When $\boldsymbol{\alpha}$ is fixed[1], we want to find the global minimum by finding the optimal resource allocation vector $\boldsymbol{f}$. As previously described in Section 4.3, the optimization problem, Equation (4.16), is a function of the total cost for offloading devices and local devices. We begin by neglecting the total cost for devices performing local

[1]In Lagrange multipliers derivations, we assume that $\boldsymbol{\alpha} \neq \mathbf{0}$, i.e., $\mathcal{N}^{\boldsymbol{\alpha}}$ is not empty.

**Algorithm 3:** Random Search

**Input:** $\mathcal{N}$, $F$, `numSamples`
**Output:** $C^{\mathrm{b}}_{\mathrm{total}}$
$C^{\mathrm{b}}_{\mathrm{total}} \leftarrow \infty$
**for** *numSamples* **do**
  **for** $i \in \mathcal{N}$ **do**
    $\alpha_i \leftarrow \mathrm{uniform}(\{0, 1\})$
  **if** $\boldsymbol{\alpha} \neq \mathbf{0}$ **then**
    ▷ Check for offloading devices
    **for** $i \in \mathcal{N}$ **do**
      ▷ Calculate resource allocations
      $g_i \leftarrow \begin{cases} \mathrm{uniform}([0, 1]) & \text{if } a_i \equiv 1, \\ 0 & \text{otherwise} \end{cases}$
    **for** $i \in \mathcal{N}$ **do**
      $f_i \leftarrow \frac{g_i}{\sum g_i} F$ ▷ Normalize resource allocations
  **else**
    $\boldsymbol{f} \leftarrow \mathbf{0}$ ▷ Fill resource allocation vector with zeros
  $C_{\mathrm{total}} \leftarrow \mathtt{TotalCost}(\boldsymbol{\alpha}, \boldsymbol{f})$ ▷ Calculate cost according to Algorithm 4
  **if** $C_{\mathrm{total}} < C^{\mathrm{b}}_{\mathrm{total}}$ **then**
    $C^{\mathrm{b}}_{\mathrm{total}} \leftarrow C_{\mathrm{total}}$ ▷ Calculate lowest cost
**return** $C^{\mathrm{b}}_{\mathrm{total}}$

**Algorithm 4:** `TotalCost`

**Input:** $\mathcal{N}$, $\boldsymbol{\alpha}$, $\boldsymbol{f}$, $B_i$, $D_i$, $f^{\mathrm{l}}_i$, $z_i$, $I^{\mathrm{t}}_i$, $I^{\mathrm{e}}_i$, $F$, $W$, $N_0$, $d_i$, $P_i$, $P^{\mathrm{s}}_i$
**Output:** $C_{\mathrm{total}}$
$K \leftarrow |\mathcal{N}^{\boldsymbol{\alpha}}|$ ▷ Set number of offloading devices
$C_{\mathrm{total}} \leftarrow 0$ ▷ Set total cost of system
**for** $i \in \mathcal{N}$ **do**
  **if** $\alpha_i = 1$ **then**
    $h_i \leftarrow \frac{1}{d_i^2}$ ▷ Calculate channel gain
    $r_i \leftarrow \frac{W}{K} \log\left(1 + \frac{P_i h_i}{\frac{W}{K} N_0}\right)$ ▷ Calculate uplink rate
    $T^{\mathrm{o}}_i \leftarrow \frac{B_i}{r_i} + \frac{D_i}{f_i}$ ▷ Calculate delay
    $E^{\mathrm{o}}_i \leftarrow \frac{P_i B_i}{r_i} + \frac{P^{\mathrm{s}}_i D_i}{f_i}$ ▷ Calculate energy consumption
    $C^{\mathrm{o}}_i \leftarrow I^{\mathrm{t}}_i T^{\mathrm{o}}_i + I^{\mathrm{e}}_i E^{\mathrm{o}}_i$ ▷ Calculate cost
    $C_{\mathrm{total}} \leftarrow C_{\mathrm{total}} + C^{\mathrm{o}}_i$ ▷ Update total cost
  **else**
    $T^{\mathrm{l}}_i \leftarrow \frac{D_i}{f^{\mathrm{l}}_i}$ ▷ Calculate delay
    $E^{l}_i \leftarrow z_i D_i$ ▷ Calculate energy consumption
    $C^{\mathrm{l}}_i \leftarrow I^{\mathrm{t}}_i T^{\mathrm{l}}_i + I^{\mathrm{e}}_i E^{\mathrm{l}}_i$ ▷ Calculate cost
    $C_{\mathrm{total}} \leftarrow C_{\mathrm{total}} + C^{\mathrm{l}}_i$ ▷ Update total cost
**return** $C_{\mathrm{total}}$

computation since they are not dependent on an allocation from the server to find the optimal $\boldsymbol{f}$, as stated in

$$\arg\min_{\boldsymbol{f}} \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} C_i^{\mathrm{o}}. \tag{4.21}$$

The optimization problem is convex in $\boldsymbol{f}$, meaning that there exists a global minimum.

After this, we show the full equation as a function of delay and energy according to

$$\begin{aligned} &\arg\min_{\boldsymbol{f}} \left[ \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} I_i^{\mathrm{t}} T_i^{\mathrm{o}} + I_i^{\mathrm{e}} E_i^{\mathrm{o}} \right] \\ &\qquad = \arg\min_{\boldsymbol{f}} \left[ \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} I_i^{\mathrm{t}} \left( \frac{B_i}{r_i} + \frac{D_i}{f_i} \right) + I_i^{\mathrm{e}} \left( \frac{P_i B_i}{r_i} + \frac{P_i^{\mathrm{s}} D_i}{f_i} \right) \right]. \end{aligned} \tag{4.22}$$

Since many of the parameters are constants not dependent on an allocation from the server, $f_i$, we can exclude them from the optimization problem and reformulate it as

$$\arg\min_{\boldsymbol{f}} \left[ \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} \left( \frac{D_i}{f_i} + \frac{P_i^{\mathrm{s}} D_i}{f_i} \right) \right] = \arg\min_{\boldsymbol{f}} \left[ \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} \frac{D_i (1 + P_i^{\mathrm{s}})}{f_i} \right]. \tag{4.23}$$

Now, we can utilize Lagrange multipliers to solve the optimization problem. We begin by defining the first constraint according to

$$\sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} f_i = F. \tag{4.24}$$

This constraint states that we always allocate the full capacity of the MEC server for the number of offloading devices in the system. It is better to utilize the whole server capacity to give offloading devices as much resource allocation as possible for minimization of the total cost for the whole system.

The second constraint states that an allocation from the MEC server has to be a positive allocation according to

$$\sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} f_i > 0. \tag{4.25}$$

Since all the variables in this optimization problem are positive, the feasible of the optimization problem is open and we conclude that a global minimum is not at a border region.

First, we define the Lagrangian function as

$$\mathcal{L}(\{f_i : \alpha_i = 1\}, \lambda) = \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} \frac{D_i (1 + P_i^{\mathrm{s}})}{f_i} + \lambda \left( \sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} f_i - F \right). \tag{4.26}$$

Next, we take the partial derivative on Equation (4.26) with respect to $f_i$, and set it to 0:

$$\frac{\partial \mathcal{L}}{\partial f_i} = 0, \quad i \in \mathcal{N}^{\boldsymbol{\alpha}}. \tag{4.27}$$

Furthermore, the partial derivative of Equation (4.26) with respect to $\lambda$ is also set to 0:

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 0. \tag{4.28}$$

We can now solve for $\lambda$ as

$$\lambda = \left( \frac{\sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} \sqrt{D_i(1 + P_i^{\mathrm{s}})}}{F} \right)^2. \tag{4.29}$$

The same formulation is also applied to $f_i$ as

$$f_i = \sqrt{\frac{D_i(1 + P_i^{\mathrm{s}})}{\lambda}}. \tag{4.30}$$

In the end, by combining Equation (4.29) and Equation (4.30) we can calculate the function to find the optimal resource allocation according to

$$f_i = \frac{\sqrt{D_i(1 + P_i^{\mathrm{s}})}}{\sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} \sqrt{D_i(1 + P_i^{\mathrm{s}})}} F. \tag{4.31}$$

### Algorithm

The algorithm starts by generating all possible action vectors for the number of devices in the system. For each action vector, we calculate the optimal resource allocation for a given action vector using Equation (4.31). In the end, we calculate the total cost $C_{\text{total}}$ with the optimal resource allocation vector and return $C_{\text{total}}^{\mathrm{b}}$ of all the generated action vectors and resource allocation vectors.

The optimal solution strategy has a significantly high complexity of $O(N2^N)$. Since the algorithm is computationally expensive, it will only work for up to 20 devices in the system. This will be further elaborated on in Chapter 5.

**Algorithm 5:** Optimal Solution

**Input:** $\mathcal{N}$, $P_i^{\mathrm{s}}$, $D_i$
**Output:** $C_{\text{total}}^{\mathrm{b}}$
$C_{\text{total}}^{\mathrm{b}} \leftarrow \infty$
**for** $\boldsymbol{\alpha} \in \{0,1\}^{|\mathcal{N}|}$ **do**
  **for** $i \notin \mathcal{N}^{\boldsymbol{\alpha}}$ **do**
    $f_i \leftarrow 0$
  **for** $i \in \mathcal{N}^{\boldsymbol{\alpha}}$ **do**
    $f_i \leftarrow \frac{\sqrt{D_i(1+P_i^{\mathrm{s}})}}{\sum_{i \in \mathcal{N}^{\boldsymbol{\alpha}}} \sqrt{D_i(1+P_i^{\mathrm{s}})}} F$
  $C_{\text{total}} \leftarrow \texttt{TotalCost}(\boldsymbol{\alpha}, \boldsymbol{f})$ ▷ Calculate cost according to Algorithm 3
  **if** $C_{\text{total}} < C_{\text{total}}^{\mathrm{b}}$ **then**
    $C_{\text{total}}^{\mathrm{b}} \leftarrow C_{\text{total}}$ ▷ Calculate lowest cost
**return** $C_{\text{total}}^{\mathrm{b}}$

### 4.4.5 Reinforcement Learning

The RL strategy uses Q-learning to find the minimum total cost according to Algorithm 6. Different from the previous strategies, we begin by defining the key components of the RL system first introduced Section 2.2.2.

#### Components of the Reinforcement Learning System

**State:** In our RL system, a state is a combination of an action vector and a resource allocation vector defined[2] as $s_t = (\boldsymbol{\alpha}_t, \boldsymbol{f}_t) = (\alpha_{t,1}, \ldots, \alpha_{t,N}, f_{t,1}, \ldots, f_{t,N})$. Furthermore, we define the terminal state through available capacity on the MEC server defined as

$$\varphi_t = F - \sum_{i=1}^{N} f_{t,i}. \tag{4.32}$$

If $\varphi_t = 0$, we have found a terminal state and will stop the training round.

**Action:** We define an action to transition to the next state as an increase of server capacity allocated to some of the devices. The state is currently defined as $s_t = (\boldsymbol{\alpha}_t, \boldsymbol{f}_t)$ and the action will be a change in $\boldsymbol{f}_t$ (and, if needed, in $\boldsymbol{\alpha}_t$) that leads to the next state $s_{t+1} = (\boldsymbol{\alpha}_{t+1}, \boldsymbol{f}_{t+1})$.

**Reward:** As the agent interacts with the environment, it will receive a reward after performing an action to transition from the current state to the next state. In our system, we define the reward as

$$r_{t+1}(s_t) = \frac{C_{\text{local}} - C(s_t)}{C_{\text{local}}}. \tag{4.33}$$

As previously mentioned, the agent aims to maximize the cumulative reward and minimize the system's total cost. With this in mind, the reward function has to be negatively correlated to the total cost. Here, $C_{\text{local}}$ is the total cost where every device in the system performs local computing and $C(s_t)$ is the total cost of the agent's current state computed from Algorithm 4. We use $C_{\text{local}}$ as an initial benchmark.

#### Environment

The environment the agent interacts with is generated as the agent traverses the states. For each state the agent is in, all possible next states will be generated with the next action and resource allocation. The resource allocation for an offloading device in a given time step is calculated according to

$$f_{t,i} = \frac{\min\left(\frac{\varphi_t \cdot \beta}{F} + 1, \psi\right) \cdot F}{\beta} \tag{4.34}$$

for $i \in \mathcal{N}^{\boldsymbol{\alpha}}$, where the server capacity is converted to several quantities and allocated to offloading devices. Here, $\beta$ is the size of the neighborhood for the agent to explore and $\psi$ is the maximum number of steps the agent can make. For example, if the agent is in the full local starting state with 3 devices in the system, $s_t =$

[2]Note that $\boldsymbol{\alpha}$ is in fact redundant as in every correct state we have that $\alpha_i = 1$ if and only if $f_i > 0$.

$(\boldsymbol{\alpha}_t, \boldsymbol{f}_t) = ((0,0,0),(0,0,0))$, a possible next state can be $s_{t+1} = (\boldsymbol{\alpha}_{t+1}, \boldsymbol{f}_{t+1}) = ((0,1,1),(0,3.25,1.75))$ where devices 2 and 3 choose offloading to the MEC server. Since multiple devices decided to offload, the server will allocate from the available computational resources, $\varphi_t$, to the offloading devices.

All the current states, along with the neighboring states, are stored in $Q_{\text{table}}$. One row in $Q_{\text{table}}$ is composed of a pair of the current state, $s_t$, and a single next state $s_{t+1}$. Every possible next state is stored with the current state in a unique row in $Q_{\text{table}}$. Furthermore, for each row in $Q_{\text{table}}$, we store the hash representation of the state for faster identification of rows, the total cost of the current state, $C(s_t)$, the reward, $r_{t+1}$, the remaining computational capacity of the MEC server, $\varphi_t$, and at the end the Q-value for the current state and action, $Q(s_t, a_t)$. For the next state in the row, we store the total cost of the next state, $C(s_{t+1})$, and the available computational capacity left from the next state, $\varphi_{t+1}$. As more devices join the system, the state space will increase rapidly, and the size of $Q_{\text{table}}$ will be enormous with millions of possible entries.

#### Algorithm

The Q-learning algorithm starts with setting the initial state, $s$, to full local and generating the neighborhood containing all possible next states. Each state is converted to a hash representation for faster identification of rows. Then, the algorithm selects a random state in $Q_{\text{table}}$. If the selected state is a terminal state, we stop and pick another random state. On the other hand, if it is not a terminal state, we choose a random action, receive the reward, and observe the next state. If the next state is a terminal state, the agent stops and selects a new random state. Otherwise, we store the next state and generate the corresponding neighborhood containing all possible next states. After this, we perform a TD update on the Q-value for the current state. After the Q-value is updated, the next state is set to be our current state. The training continues until the number of episodes has ended. In the end, we return the result of the best terminal state that yields the minimum total cost from a selection of multiple terminal states.

### 4.4.6 DQN

The DQN strategy uses DRL to find the minimum total cost according to Algorithm 7. Similar to RL, we begin by defining the key components of the DRL system first introduced Section 2.4 along with the components of the neural network implemented first introduced in Section 2.3.

#### Components of the DQN

In this strategy, we always allocate server capacity in discrete quantities, such that each $f_{t,i}$ is a multiple of $F/\zeta$, where $\zeta$ is a positive integer.

**State:** For the DRL system, the state is defined the same way as for the RL strategy, where $s_t = (\boldsymbol{\alpha}_t, \boldsymbol{f}_t) = (\alpha_{t,1}, \ldots, \alpha_{t,N}, f_{t,1}, \ldots, f_{t,N})$ is the state at a given time step $t$. Furthermore, the definition of a terminal state remains the same for the DRL strategy as for the RL strategy. If a terminal state is found, we reset to a full local starting position.

**Algorithm 6:** Q-learning

**Input:** $\mathcal{N}$, $\eta$, $\gamma$, `numEpisodes`
**Output:** $C^{\text{b}}_{\text{total}}$
Set $s$ to full local
Store $s$ in $Q_{\text{table}}$
**for** $i \in [\textit{numEpisodes}]$ **do**
  Select non-terminal random $s_0 = (\boldsymbol{\alpha}_0, \boldsymbol{f}_0)$ from $Q_{\text{table}}$
  $t \leftarrow 0$
  **while** True **do**
    Select random $a_t$ from $s_t$
    Execute $a_t$, obtain $r_{t+1}$, and observe $s_{t+1}$
    **if** $s_{t+1}$ is terminal **then**
      Break
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
    Store $s_{t+1}$ in $Q_{\text{table}}$
    $t \leftarrow t + 1$
$C^{\text{b}}_{\text{total}} \leftarrow \min\{\texttt{TotalCost}(s) : s \in Q_{\text{table}},\ s \text{ is terminal}\}$
**return** $C^{\text{b}}_{\text{total}}$

Since we allocate to each device a multiple of $F/\zeta$, the total number of states can be calculated using a binomial coefficient for a given number of devices in the system according to

$$\binom{\zeta + N}{N} = \frac{(\zeta + N)!}{N!\zeta!}, \tag{4.35}$$

Furthermore, the total number of terminal states in the system where $\varphi_t = 0$ is calculated according to

$$\binom{\zeta + N - 1}{N - 1} = \frac{(\zeta + N - 1)!}{(N-1)!\zeta!}. \tag{4.36}$$

The number of states and terminal states will increase rapidly as more devices join the system. For example, for 3 devices in the system with $\zeta = 100$, the total number of states will be 176851 with 5151 terminal states. For 7 devices with $\zeta = 100$, the total number of states will increase to 26075972546 with 1705904746 terminal states. Both Equation (4.35) and Equation (4.36) will be used to describe the rapidly increasing state-space further elaborated on in Chapter 5.

**Action:** An action is defined as an alteration of $s_t = (\boldsymbol{\alpha}_t, \boldsymbol{f}_t)$. More precisely, an action allocates additional server capacity of $F/\zeta$ to one of the devices $i_0 \in \mathcal{N}$. The next state $s_{t+1}$ is then defined by

$$f_{t+1,i} = \begin{cases} f_{t,i} & \text{if } i \neq i_0, \\ f_{t,i} + \frac{F}{\zeta} & \text{otherwise} \end{cases} \tag{4.37}$$

and

$$\alpha_{t+1,i} = \begin{cases} 1 & \text{if } f_{t+1,i} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.38}$$

The allocation is a part of the MEC server's total capacity, where $\zeta$ determines how many discrete allocation quantities should be given to an offloading device from the server capacity.

**Policy:** The agent follows an $\epsilon$-greedy policy as previously described in Section 2.4. The agent begins by selecting random actions for a number of episodes before selecting $\epsilon$-greedy actions from the neural network.

**Reward:** The reward function is

$$r_{t+1}(s_t, s_{t+1}) = \frac{C(s_t) - C(s_{t+1})}{C_{\text{local}}}. \tag{4.39}$$

Instead of comparing the total cost of the current state with the total cost of local computing done in the RL strategy, the reward function implemented in this strategy uses the total cost of the current state and that of the next state for a more direct evaluation of the agent's path. The goal is to determine if the agent is able to find the best course of action while interacting with the environment.

### Environment

The environment the agent interacts with is stored in a replay buffer of a fixed length $b$, which depends on the number of devices in the system. Before training initiates, the replay buffer is filled up with uniformly random experiences from the agent. By filling up the buffer with random experiences, the bias between the agent's experiences will be reduced, and the neural network will generalize better, as previously discussed in Section 2.3.6. Furthermore, for each time step, $t$, we store the agent's current state, $s_t$, the agent's action, $a_t$, the reward, $r_{t+1}$, the next state $s_{t+1}$, and a Boolean value $T_t$ indicating if the next state is a terminal state or not. The experience is denoted as $e_t = (s_t, a_t, r_{t+1}, s_{t+1}, T_t)$. After this, we utilize MGD to sample a random mini-batch of size $M$ from the buffer. Through the use of MGD, we can fit exceedingly large state spaces for a batch size of a respectable size.

### Neural Network

The neural network implemented is a linear neural network with 5 layers, 3 hidden layers, and a leaky ReLU activation function as illustrated in Figure 4.2. The input layer contains the agent's current state. The number of neurons for the hidden layers in the network is set to 192, 256, and 64, respectively, while the size of the output layer is equal to the number of actions the agent may select from the input state. The network also utilizes the Adam optimizer for improved optimization and handling of bias.

The neural network architecture design is completed through hyper-parameter testing and trial and error. Multiple different networks have been tested with different hyper-parameters and number of devices. While a less complex neural network was more stable for our problem formulation, the more complex network implemented was overall better at the main objective of finding a lower total cost.

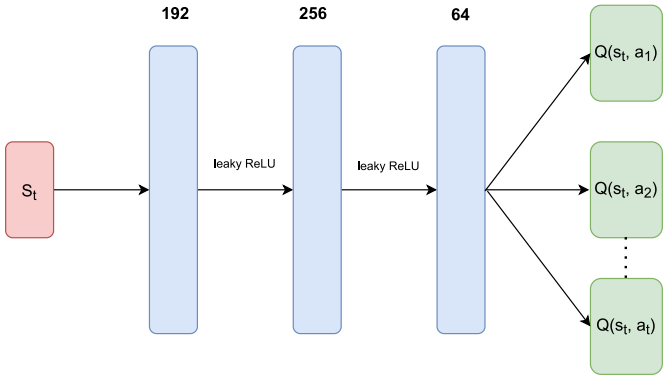


Figure 4.2: The architecture of the implemented neural network.

### Algorithm

The first stage of the algorithm starts by pre-filling the buffer with random experiences up to the set buffer capacity. By pre-filling the buffer, the DQN algorithm will converge more rapidly due to the batch being filled up with experiences as training initiates rather than at a later stage. After this, the starting state is set to full local. The training starts with the agent performing an action while following an $\epsilon$-greedy policy and transitioning to a new state, $s_{t+1}$, before the reward, $r_{t+1}(s_t, s_{t+1})$, from the transition is calculated. Then, after the reward is calculated, the algorithm checks if the new state is a terminal state or not. If $s_{t+1}$ is a terminal state, the Boolean terminal state flag, $T_t$, is set to true and false if not. The algorithm then stores the experience, $e_t$, in the replay buffer.

The second stage of the algorithm begins with sampling $M$ random experiences from the replay buffer $\mathcal{E} = \{e_{t^{(1)}}, \ldots, e_{t^{(b)}}\}$, where the $i$-th sampled experience is denoted as $e_{t^{(j_i)}} = (s_{t^{(j_i)}}, a_{t^{(j_i)}}, r_{t^{(j_i)}+1}, s_{t^{(j_i)}+1}, T_{t^{(j_i)}})$. First, we calculate the target vector, $\hat{\boldsymbol{y}}_t = (\hat{y}_{t^{(j_1)}}, \ldots, \hat{y}_{t^{(j_M)}})$, by transmitting the states from the mini-batch through the target network and selecting the maximum Q-value according to

$$\hat{y}_{t^{(j_i)}} \leftarrow \begin{cases} r_{t^{(j_i)}+1} + \gamma \max_a \widehat{Q}(s_{t^{(j_i)}+1}, a; \boldsymbol{\Theta}^-) & \text{if } T_{t^{(j_i)}} = 0, \\ r_{t^{(j_i)}+1} & \text{otherwise.} \end{cases} \tag{4.40}$$

Next, the algorithm predicts the Q-values, $\boldsymbol{Q}_t^{\text{pred}} = \left(Q_{t^{(j_1)}}^{\text{pred}}, \ldots, Q_{t^{(j_M)}}^{\text{pred}}\right)$, on the model $Q$ according to

$$Q_{t^{(j_i)}}^{\text{pred}} \leftarrow \begin{cases} Q(s_{t^{(j)}}, a_{t^{(j)}}; \boldsymbol{\Theta}) & \text{if } T_{t^{(j_i)}} = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.41}$$

Next, we calculate the loss on $\hat{\boldsymbol{y}}_t$ and $\boldsymbol{Q}_t^{\text{pred}}$ with the Huber loss function on the whole mini-batch according to Equation (2.18), first presented in Section 2.3.4. The gradient descent step is then performed on the loss where each gradient entry is clipped to the range of $(-1, 1)$. The weights from $Q$ are then loaded onto $\widehat{Q}$ for more stable training when

$$t \bmod C \equiv 0, \tag{4.42}$$

where $t$ is the current time step and $C$ is the update frequency. In the end, after an episode is finished, the $\epsilon$ is updated for an $\epsilon$-greedy policy action selection according to

$$\epsilon \leftarrow \max\{\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}}\}, \tag{4.43}$$

where $\epsilon_{\text{min}}$ is the minimum allowed value of $\epsilon$ and $\epsilon_{\text{decay}}$ is the rate to decay $\epsilon$. Both $\epsilon_{\text{min}}$ and $\epsilon_{\text{decay}}$ are hyper-parameters set before training initiates. Furthermore, the total cost, $C_{\text{total}}$, is calculated for the given terminal state and compared to the current best total cost $C^{\text{b}}_{\text{total}}$. If the algorithm is not able to find a better total cost after 500 episodes, we terminate from further training. When that occurs, the terminal state that yields the minimum total cost from a selection of multiple terminal states is returned.

**Algorithm 7:** DQN

**Input:** $\mathcal{N}$, $\eta$, $\epsilon$, $\epsilon_{\text{min}}$, $\epsilon_{\text{decay}}$, $\gamma$, $M$, $C$, `numEpisodes`
**Output:** $C^{\text{b}}_{\text{total}}$
$Q \leftarrow \text{Model}(N, N, \eta)$ with weights $\boldsymbol{\Theta}$ ▷ Initialize model
$\widehat{Q} \leftarrow \text{Model}(N, N, \eta)$ with weights $\boldsymbol{\Theta}^{-}$ ▷ Initialize target model
$C^{\text{b}}_{\text{total}} \leftarrow \infty$
Pre-fill buffer with random experiences
**for** $i \in$ [*numEpisodes*] **do**
  Set $s_0 = (\boldsymbol{\alpha}_0, \boldsymbol{f}_0)$ to full local
  $t \leftarrow 0$
  **while** $s_t$ *not terminal* **do**
    Select random $a_t$ with probability $\epsilon$ according to the $\epsilon$-greedy policy
    Alternatively, $a_t \leftarrow \arg\max_a Q(s_t, a; \boldsymbol{\Theta})$
    Perform $a_t$ from $s_t$, receive $r_{t+1}$, and transition to $s_{t+1}$
    Check if $s_{t+1}$ is terminal and set boolean flag $T_t$
    Store experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1}, T_t)$ in replay buffer
    Sample $M$ experiences $(s_{t^{(j_i)}}, a_{t^{(j_i)}}, r_{t^{(j_i)}+1}, s_{t^{(j_i)}+1}, T_{t^{(j_i)}})$ from replay buffer
    Calculate $\hat{\boldsymbol{y}}_t$ according to Equation (4.40)
    Calculate $\boldsymbol{Q}^{\text{pred}}_t$ accroding to Equation (4.41)
    Perform gradient descent on $L(\hat{\boldsymbol{y}}_t, \boldsymbol{Q}^{\text{pred}}_t)$ where gradient entries are clipped to be in the range $(-1, 1)$
    **if** $t \bmod C \equiv 0$ **then**
      Load weights from $Q$ to $\widehat{Q}$
    $t \leftarrow t + 1$
  Update $\epsilon$ according to Equation (4.43)
  $C_{\text{total}} \leftarrow \texttt{TotalCost}(s_t)$ ▷ Calculate cost according to Algorithm 3
  **if** $C_{\text{total}} < C^{\text{b}}_{\text{total}}$ **then**
    $C^{\text{b}}_{\text{total}} \leftarrow C_{\text{total}}$
**return** $C^{\text{b}}_{\text{total}}$

## 4.5 Optimal Action-Value Function $Q_*$

The optimal action-value function $Q_*$ is given by a fixed point of the iterative process in Equation (2.10) presented in Section 2.2.3, i.e.,

$$Q_*(s_t, a_t) = Q_*(s_t, a_t) + \eta[r_{t+1} + \gamma \max_a Q_*(s_{t+1}, a) - Q_*(s_t, a_t)]. \tag{4.44}$$

This can be further simplified to

$$Q_*(s_t, a_t) = r_{t+1} + \gamma \max_a Q_*(s_{t+1}, a). \tag{4.45}$$

For a small number of devices, the number of states and actions is not too large. Then, we can directly tabulate the optimal $Q_*$ for all states and actions, using the boundary condition $Q_*(s, \cdot) = 0$ for any terminal state $s$.

### Algorithm

Algorithm 8 tabulates $Q_*$. It works with all the states organized in layers. Layer $l$ consists of the states with total allocated server capacity ${}^{lF}/_{\zeta}$. In particular, layer $l = 0$ consists of the states where every device performs local computation, and layer $l = \zeta$ consists of all the terminal states. If state $s_t$ belongs to layer $l$, then after performing action $a_t$ the resulting state $s_{t+1}$ belongs to layer $l + 1$.

Initially, $Q_*(s, \cdot) = 0$ for all terminal states $s$. Further, if all the Q-values have been calculated for layer $l + 1$, then the values for the states in layer $l$ can be calculated according to Equation (4.45).

Tabulating $Q_*$ is computationally expensive and only works for a small number of devices. However, we can utilize the optimal $Q_*$ as a benchmark to evaluate the convergence of the DQN. This will be further discussed in Chapter 5.

**Algorithm 8:** Optimal Action-Value Function

**Input:** $\gamma$, $\zeta$
**Output:** Tabulated $Q(s, a)$ for all $s$ and $a$
**for** $s$ in layer $\zeta$ **do**
  $Q(s, \cdot) \leftarrow 0$
**for** $l = \zeta - 1, \zeta - 2, \ldots, 0$ **do**
  **for** $s$ in layer $l$ **do**
    **for** every action $a$ from state $s$ **do**
      Calculate reward $r$ for state $s$ and action $a$
      Let $s'$ be the result of performing $a$ from $s$
      **if** $l \neq \zeta - 1$ **then**
        $Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$
      **else**
        $Q(s, a) \leftarrow r$
**return** $Q(s, a)$ for all $s$ and $a$

# Chapter 5

# Results and Findings

This chapter presents the results from each strategy. Each result will be described in detail. The first section, Section 5.1, provides the results of the total cost for each strategy for different sets of devices. After this, Section 5.2 presents the total cost for each strategy with both limited and increased server capacity. In Section 5.3, the performance of the DQN are presented, and in Section 5.4 a discussion about the convergence of the network. In the end, a conclusion about the results and possible improvements will be further elaborated on in Chapter 6.

## 5.1 Total Cost

The main objective of this thesis is to examine the use of RL techniques, such as Q-learning and DRL, to heuristic solutions in the form of full local and offload, random search, and the optimal solution. For each strategy presented, we analyze the performance to the optimal solution and determine if the strategy is able to scale as more devices join the system. To simulate the results of each strategy, we begin by generating 50 different sets of $N$ devices. We then take the average of each result, where the number of devices is increased from 3 up to 50. Since the total cost is an enormous number, we divide it by $10^9$ for better visualization. The parameters for the MEC server and each device are generated according to Table A.1. Each strategy presented is simulated on the same set of generated device parameter settings for accurate performance estimation. For the random search strategy, we set `numSamples` = 10.000 while the parameters for Q-learning and the DQN can be found in Table A.2 and Table A.3, respectively. In the end, as the number of devices in the system changes, the parameters are re-generated.

From Figure 5.1 we can observe the total cost, $C_{\text{total}}$, of each strategy as the number of devices in the system increases. For 3 to 7 devices, the full local strategy has the overall worst performance with a linear increase in total cost. The cost of random search and Q-learning is equal to that of the optimal solution for 3 devices, while the cost of DQN is very close, with only a 0.1 difference. For 4, 5, 6, and 7 devices, the performance of random search, Q-learning, and DQN is incredibly close to the optimal solution with only an 0.001 to 0.2 difference. As previously mentioned, the state space increases rapidly as more devices join. However, the increase in the state space from 3 to 7 devices has little impact on the performance of the DQN and Q-learning algorithm. At the same time, the random search strategy is able to consistently generate close-to-optimal solutions.

The results between each strategy become a little more apparent for 7 to 20 devices. We can observe that the total cost for the full offload strategy increases exponentially as more devices join the system. The Q-learning, DQN, and random search strategies are consistently close to the optimal solution, with minor changes in cost between each strategy. As previously mentioned in Section 4.4.4 and Section 4.4.5, both the optimal and the Q-learning strategy have significant scaling limitations. The computational complexity difference between 20 and 50 devices is tremendous for the optimal strategy. As already reported, the complexity of the optimal strategy is $O(N2^N)$, meaning that for 20 devices, the optimal solution needs $20 \cdot 2^{20} = 20971520$ iterations to find optimal resource allocations. For 50 devices, the number of iterations to calculate the optimal increases to $50 \cdot 2^{50} = 56294995342131200$ and would not be solvable. The Q-learning strategy has other scaling limitations. As an increasing number of devices join the system, the size of $Q_{\text{table}}$ increases to include millions to billions of possible states. The result of the rapid growth of the size $Q_{\text{table}}$ is a failure in the algorithm where the agent cannot iteratively update the Q-value through TD updates for each state and action pair.

For 50 devices, the performance difference between each strategy increases. As more devices join the system, the random search strategy becomes increasingly worse compared to the full local and DQN strategies. Since the number of different resource allocations increases, the number of samples needed to achieve good performance also increases. Furthermore, we can observe that the DQN has worse performance compared to full local. For 50 devices in the system, the state space increases to $2.013 \times 10^{40}$ with $6.708 \times 10^{39}$ different terminal states, where both numbers are rounded to three decimal points and calculated from Equation (4.35) and Equation (4.36). For the DQN algorithm to achieve good performance with such a gigantic state space, the algorithm would have to train for an extensive period of time where the size of the replay buffer, currently with 1.000.000 experiences, would have to be increased considerably to successfully fit the increased state space. Additionally, the architecture of the neural network should be modified with an increased number of neurons and hidden layers to better accommodate the changes in the state space in addition to other parameters utilized for the DQN. In the end, while the full offload strategy can accommodate 50 devices, we can observe that the total cost for the full offload strategy increases beyond the scale.

## 5.2 Increased Server Capacity

For the next simulation, we are interested in analyzing the performance of each strategy with respect to both limited and increased server capacity for a set of devices. The simulation is similar to the first simulation, where 50 different sets of $N$ devices are generated. We set $N = 20$ to visualize the performance of each strategy better and generate the device parameter settings according to Table A.1 over different server capacities $F$. For the Q-learning strategy, we only change the size of the neighborhood to be

$$\beta = 2 \cdot F, \tag{5.1}$$

while the number of resource allocations for the DQN, $\zeta$, is kept linear in $F$. For example, where we set $F = 5$ and $\zeta = 100$ in the previous simulation, we now set

$F = 1$ and $\zeta = 20$ or $F = 10$ and $\zeta = 200$ to keep the resource allocation for each offloading device equal to changes in $F$. Otherwise, the other parameters in Q-learning and DQN are kept equal.

From Figure 5.2 we can observe that the total cost decreases as the server capacity is increased, with the exemption of the full local strategy that does not depend on a resource allocation from the server. For server capacities from 1 to 3, the total cost for the DQN and Q-learning strategies are very close to the optimal, while the random search strategy starts off slightly worse but is able to improve. The performance of the DQN strategy stagnates for server capacities between 3 and 12 compared to the strategies full offload, random search, Q-learning, and optimal. Here, the random search and Q-learning strategies are close to the optimal strategy but become slightly worse as the server capacity increases. For server capacities between 12 and 15, the performance of the DQN strategy is improving. However, it is not close to the optimal and is worse than random search and Q-learning. In the end, the full offload strategy improves significantly for increasing server capacities, while the random search and Q-learning strategies have the best overall performance compared to the optimal strategy.

## 5.3 Performance of DQN

In this section, we present the performance of the DQN. We begin by displaying the average loss and reward for $N = 20$ devices. For simulation purposes, we generate 50 different sets of $N$ devices and take the average reward and loss for each training simulation, where the loss is calculated using the Huber loss function from Equation (2.18). The device parameter settings are generated equally for this simulation as the simulation in Section 5.1. Furthermore, we increase the number of training episodes to 10.000 to better visualize the convergence of the network.

For the average loss plot, Figure 5.3, we can observe that for the first 2000 episodes, the loss becomes worse, and the agent diverges toward a worse policy. Furthermore, for each 1000 episodes, we update the target network, $\widehat{Q}$, with the weights from the $Q$ network. Once this update occurs, the loss increases tremendously before the network is able to reduce it. As previously discussed in Section 2.3.4, the goal of the agent is to minimize the loss between the target and the predicted actions. From episode 2000 to 4000, the loss remains at the same level as for the first 1000 episodes and is yet to converge. In the end, after 4000 training episodes, the agent is finally able to reduce the loss for each weight update between $Q$ and $\widehat{Q}$. However, a loss of about 0.1 still remains at a high level after 8000 episodes, signifying that the predictions made are still relatively poor. For comparison, the results of a good prediction typically lead to a loss of about 0.005.

The goal of the agent is to maximize the long-term reward. From Figure 5.4, we can observe that the overall reward increases for the first 2000 episodes and stables out for the remaining training period. However, the reward fluctuates where the reward for some of the later episodes is worse than for the first 2000 episodes. The wavering performance of the DQN from this observation leads us to believe that the network may suffer from catastrophic forgetting. There are two reasons why catastrophic forgetting may occur. The first reason is due to the instability of the neural network when approximating Q-values over large state spaces, as stated in [25] by Roderick *et al.* As previously discussed, the state space increases rapidly

when more devices join the system. The result of the increasing complexity makes stable convergence more challenging. Furthermore, since the neural network learns an approximation compared to a direct evaluation of the Q-function done in RL, the network may increase the accuracy of the approximator at the expense of following a worse policy. The result may lead to reduced loss at the expense of not choosing the optimal action in a given state.

Catastrophic forgetting is a recurring problem for neural networks when learning an approximation. Currently, no universal solution to the problem exists and remains an active area of research. In [26], Kirkpatrick *et al.* proposes to use elastic weight consolidation (EWC) to enable former tasks to remain unchanged when generating new predictions. This will limit the occurrence of catastrophic forgetting. However, the results of the DQN with the EWC resulted in worse performance compared to training 10 separate DQNs on Atari games. Kirkpatrick *et al.* believes the reduced performance occurs due to the weights for each game becoming a tractable approximation of parameter uncertainty. As previously mentioned, we train 50 separate DQNs and take the average over each result. Since training separate DQNs yielded better results, we neglected to utilize EWC. Other measures for limiting catastrophic forgetting in neural networks exist. However, these papers use SL instead of DRL.

## 5.4 Convergence of DQN

In this section, we seek to further discuss the convergence and behavior of the DQN. The predicted action from the DQN in a given state will be compared to the optimal action from the optimal action-value function (computed using Algorithm 8). As previously mentioned in Section 4.4.4, calculating the optimal action-value function is computationally expensive and only works for a small number of devices. Therefore, we generate one set of $N = 3$ devices with device parameter settings from Table A.1. Furthermore, we set the number of training episodes to 10.000 to better visualize the convergence and behavior of the network.

We begin by analyzing the behavior of the DQN on 3 different runs with the same set of devices to see if the behavior of the network is different between the runs. From Figure 5.5, we can observe that the runs are almost identical, where the loss between each run only differs by about 0.001 to 0.01. This means that the DQN will make many of the same predictions for the same set of devices with equal device parameter settings.

For the next simulation, we want to analyze the predictions from the DQN compared to the optimal action-value function $Q_*$. From Figure 5.5, we can observe that the network is able to reduce the loss for each weight update from the Q-network to the target network $\widehat{Q}$. In Figure 5.6, the loss is calculated between the $Q_t^{\text{pred}}$ and $Q_*$. For the first 3000 episodes, the network is able to reduce the loss before it stables out from episode 3000 to 5000. The loss between $Q_t^{\text{pred}}$ and $Q_*$ is incredibly high and gives a good indication that the neural network is poor at predicting good actions. From episode 5000 to 10.000, the network diverges, and the loss is equal to that of the first 1000 episodes.

Even though the target network ensures more stability, it is not sufficient enough to prevent divergence, where no solution currently exists. In [27], Hasselt *et al.* introduces an empirical study to help identify the underlying problems causing di-

vergence, where function approximation, bootstrapping, and off-policy learning are referred to as the deadly triad of occurring learning divergence. For simulation, Hasselt *et al.* ran 3 distinct simulations on Atari games with a DQN to observe if the action-value function, previously explained in Section 2.2.3, reached unrealistic values with equal hyper-parameters, referred to as soft divergence. The simulation result showed that the DQN overestimates the results of the action-value function when the value of $\epsilon$ tends toward a greedy action selection, even if the target network can limit the problem from occurring at an earlier training stage. The result of this overestimation leads to unreasonable high values for $Q(s_t, a_t)$. For example, if the optimal action from the optimal action-value function in a given state is $Q_* = 10$, an overestimated value from the neural network might be $Q_t^{\text{pred}} = 20$. When this occurs, the agent will select an action from the prediction made by $Q_t^{\text{pred}}$ instead of $Q_*$ and transition to a different state. This will continue for each training episode where the agent continues to deviate, leading to divergence. Hasselt *et al.* believe the divergence becomes more apparent for more complex neural networks compared to simpler networks. In our case, when the divergence begins at episode 5000, the value of $\epsilon$ has been at a greedy action selection for around 3000 episodes. This leads us to believe that the network may also overestimate the action-value function, leading to divergence.

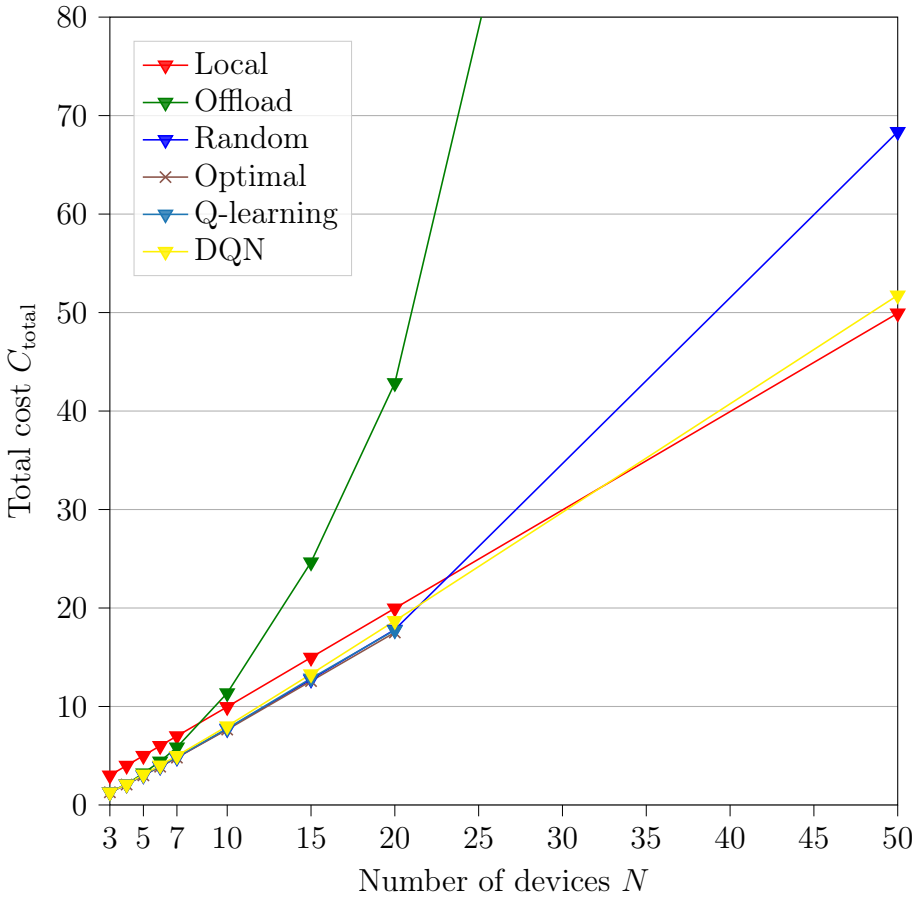


Figure 5.1: Total cost, $C_{\text{total}}$, for different number of devices, $N$. Numerical results can be found in Table B.1.

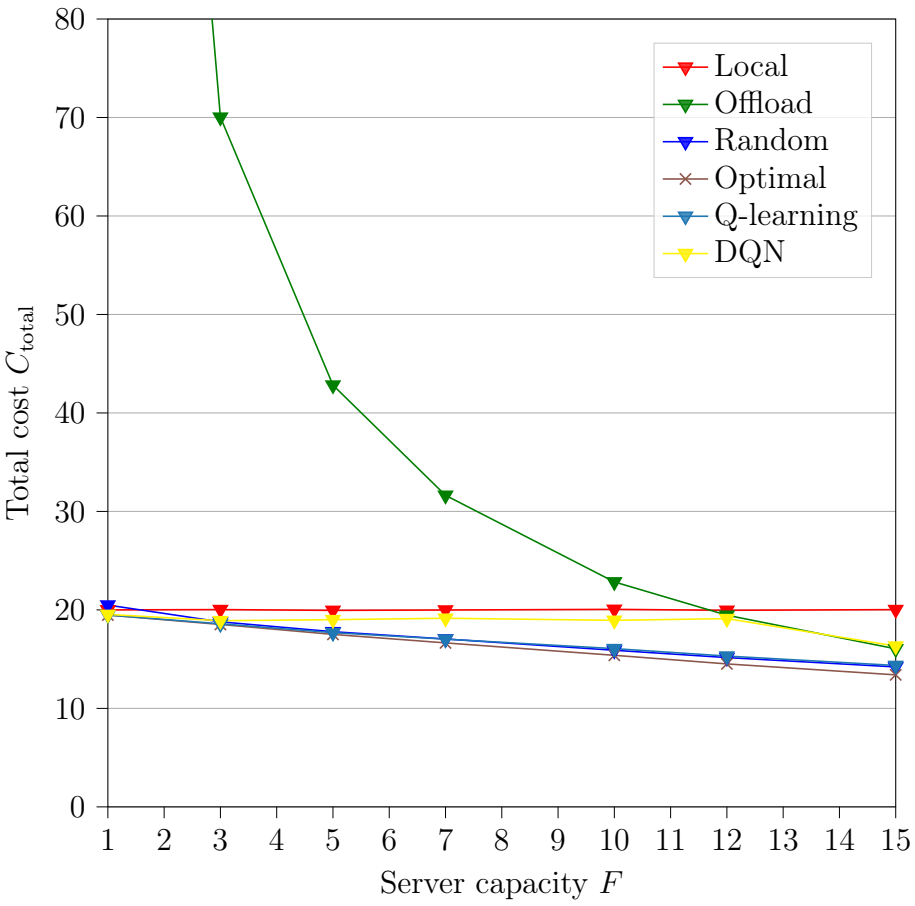


Figure 5.2: Total cost, $C_{\text{total}}$, for different server capacities, $F$, with $N = 20$ devices. Numerical results can be found in Table B.2.

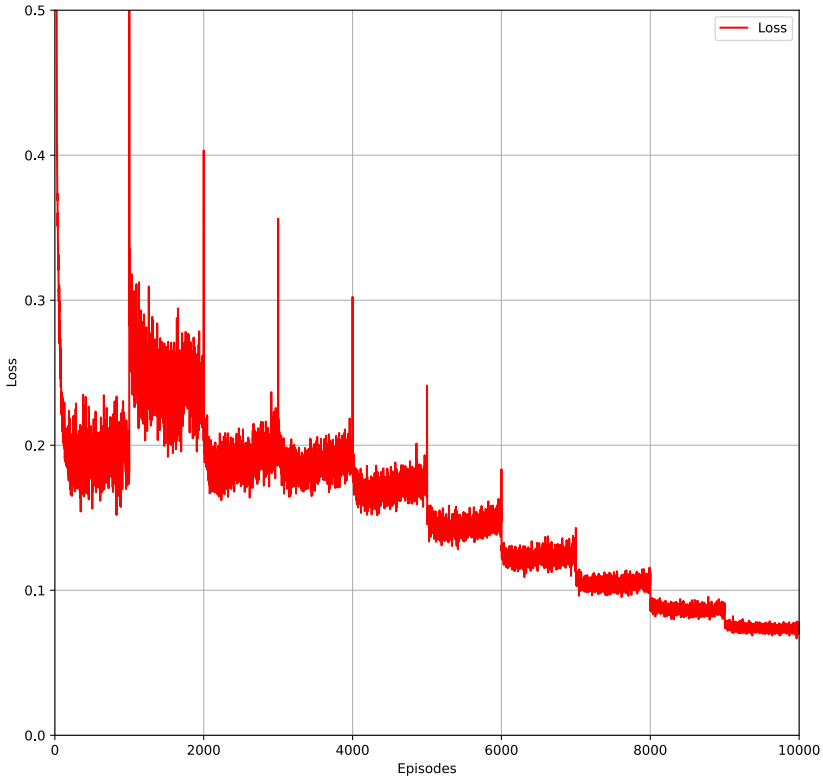


Figure 5.3: Huber loss over 10.000 training episodes for $N = 20$ devices.

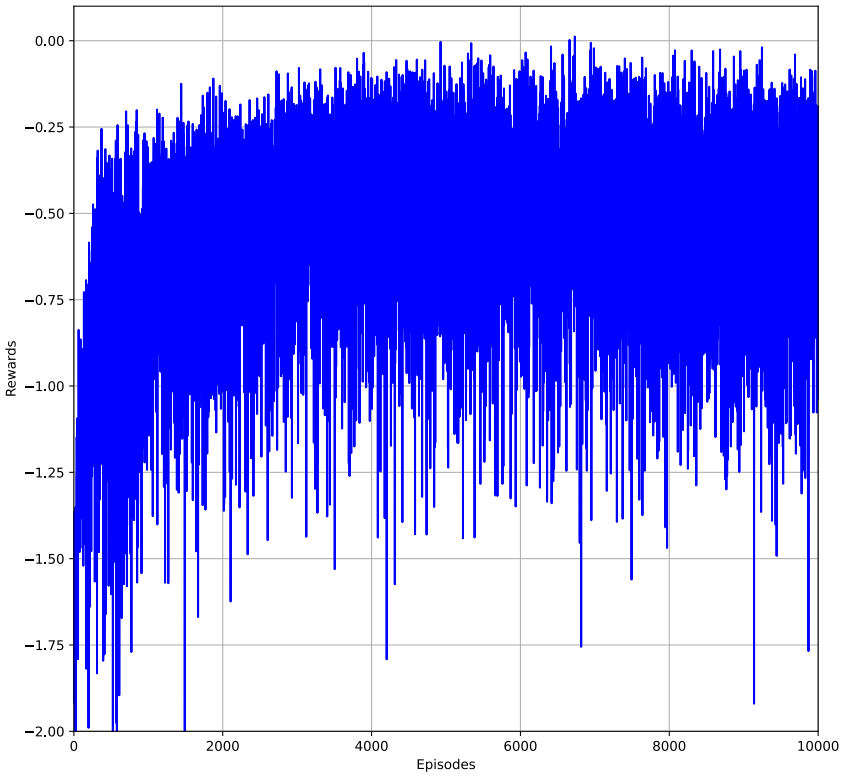


Figure 5.4: Rewards over 10.000 training episodes for $N = 20$ devices.

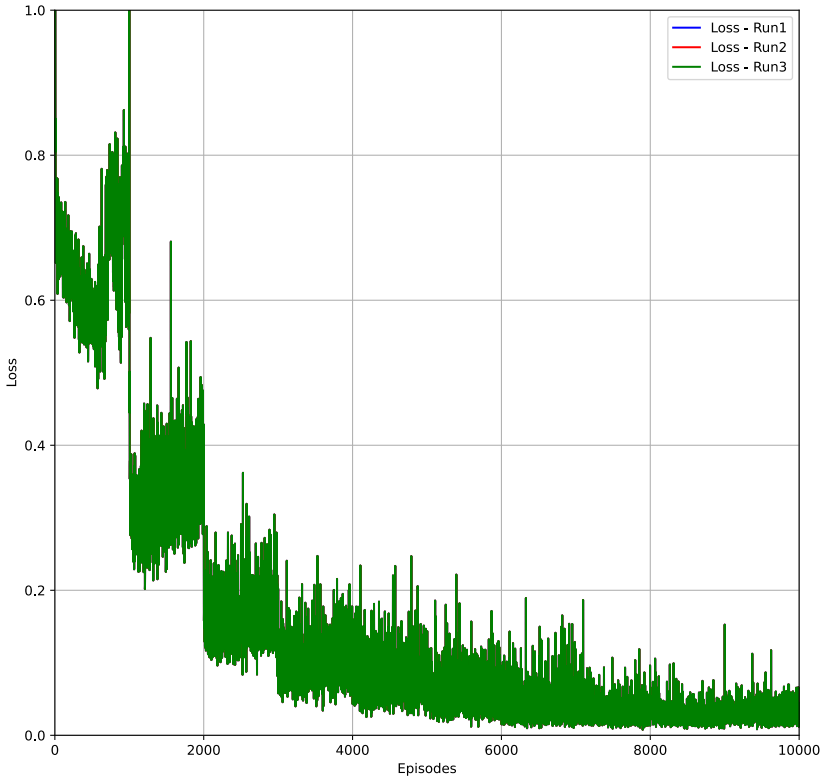


Figure 5.5: Huber loss between 3 different runs.

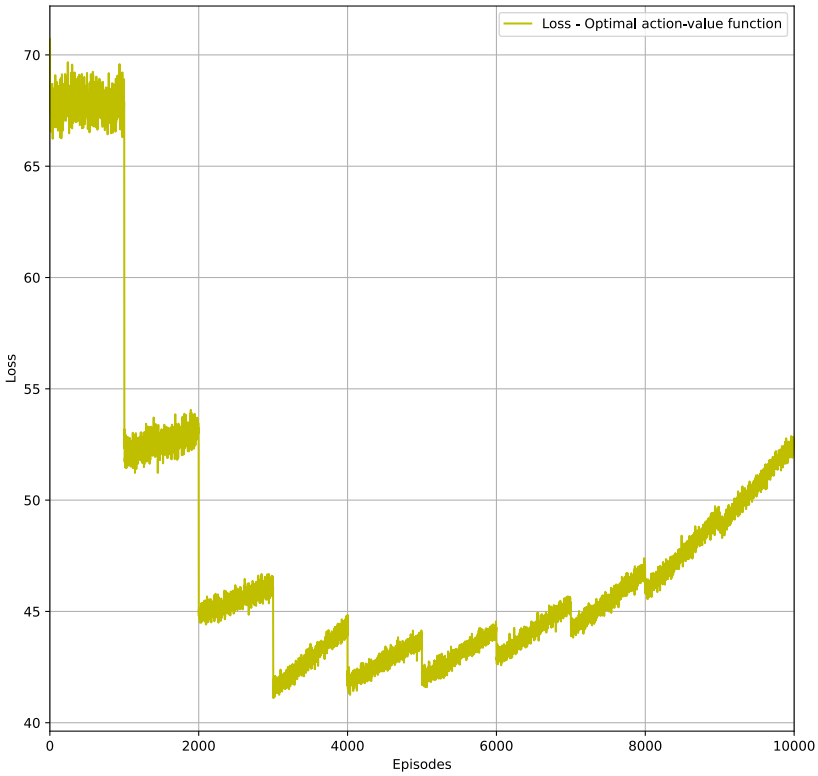


Figure 5.6: Huber loss between predictions from the DQN and the optimal action-value function.

# Chapter 6

# Conclusions and Future Work

This chapter concludes the overall results described in Chapter 5 and presents suggested future work. Further discussion about the positive aspects and limitations of each strategy will be presented, specifically RL and DRL. The suggested future work focuses on how the limitations of Q-learning and the DQN can be reduced and different problem formulations.

## 6.1 Conclusions

In this thesis, we implemented different strategies in addition to solving the optimization problem introduced in [21]. While the optimal strategy does not scale for a growing number of devices, it acts as an essential benchmark for the other strategies utilized. The performance of random search, Q-learning, and DQN strategies are very close to the optimal strategy for up to 20 devices. However, the results show generally poor performance for the strategies that can address more than 20 devices. Furthermore, for limited and increased server capacity with 20 devices, we can observe that the random search and Q-learning strategies adapt well to changes in server capacity, where the performance of full offload is only suitable for high server capacities.

In Section 5.3 and Section 5.4, we discussed the performance and convergence of the DQN. While several papers in Section 3.2 and Section 3.3 use DQN and different variants of a DQN, there is little discussion of the general limitations and convergence of a DQN in MEC. In this thesis, a more sophisticated benchmark in the form of the optimal action-value function is implemented to evaluate the convergence of the DQN better. From the results in Section 5.4, we can observe that the DQN diverges when $\epsilon$ tends toward a greedy action selection. The result of this divergence may lead to the agent not being able to find the minimum total cost. We also believe that the DQN suffers from catastrophic forgetting due to the fluctuating performance, as discussed in Section 5.3. Furthermore, the DQN generalizes poorly when the number of devices in the system changes or for different server capacities. As previously discussed, key parameters such as the size of the replay buffer, the number of training episodes, and the architecture of the neural network should be modified when the number of devices or server capacity change.

## 6.2 Future Work

While the DQN suffers from both poor performance, convergence, and lack of generalization for the different number of devices in the system, several techniques can be utilized to limit the problems that are not too common in research on MEC with DRL. The first technique is called prioritized experience replay and was introduced in [28]. Instead of uniform sampling from the replay buffer, prioritized experience replay sample those experiences with high TD error to be retrained for faster convergence. Furthermore, prioritized experience replay also introduces bias and stochastic prioritization for improved diversity of experiences from each sample.

As previously discussed in Section 5.4, the DQN suffers from overestimation leading to a higher loss for large-scale problems. In [29], Hasselt *et al.* propose to use a double-deep Q-network (DDQN) to reduce overestimation by replacing

$$\hat{y}_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \boldsymbol{\Theta}^-) \tag{6.1}$$

with

$$\hat{y}_t^{\text{DDQN}} = r_{t+1} + \gamma Q\Big(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \boldsymbol{\Theta}^-), \boldsymbol{\Theta}^+\Big), \tag{6.2}$$

where $\boldsymbol{\Theta}^+$ denotes a second set of weights. Instead of a single max operation, DDQN splits the operation into two separate operations, one for action selection and one for evaluation. To the best of our knowledge, little research in MEC has been done with a DDQN for more stable convergence and improved performance. Additionally, it is worth mentioning that double Q-learning can be achieved without a neural network by utilizing two $Q_{\text{table}}$, as discussed in [30]. However, double Q-learning is still limited by exponential growth in the size of the $Q_{\text{table}}$, where the algorithm may fail in updating the Q-values for each state and action pair.

Another possible improvement that can be utilized is a dueling-deep Q-network as presented in [31] by Wang *et al.* In [10], Mnih *et al.* utilize a fully connected convolutional neural network for the lower layers of the network. On the other hand, Wang *et al.* propose to change the lower layers of the neural network to a stream for separation of value and advantage functions. This can be converted to fit the linear neural network utilized in this thesis. By changing Equation (6.1) into

$$\hat{y}_t^{\text{Duel}} = r_{t+1} + \gamma \left(\nu(s_t; \boldsymbol{\Theta}^-, \rho) + (A(s_t, a_t; \boldsymbol{\Theta}^-, \eta) - \max_a A(s_t, a; \boldsymbol{\Theta}^-, \eta))\right), \tag{6.3}$$

the agent's policy evaluation is improved over a series of actions where divergence is less likely to occur. Here, $\nu(s_t; \boldsymbol{\Theta}^-, \rho)$ is an estimate of the value function where $\rho$ is the learning parameter for $\nu$, and $A(s_t, a_t; \boldsymbol{\Theta}^-, \eta)$ is the advantage function with $\eta$ as the learning parameter. The advantage function measures the importance of an action.

While different techniques can be utilized to improve the DQN presented in this thesis, none of these techniques solves the problem of a rapidly increasing state space. One solution is to change the problem formulation where the offloading decision and resource allocation from the server is not stored as vectors dependent on the number of devices in the system. One can, for example, change the problem formulation to multi-agent RL similar to Lu *et al.* in [19], where the devices in the system act as independent agents making their own decisions compared to a single agent presented in this thesis. However, many of the same problems still persist if an RL strategy is to be utilized in a real-world scenario.

# Appendices

# Appendix A

# Utilized Parameters

## A.1 Parameters for MEC

| Parameters | Values | Descriptions |
|---|---|---|
| $B_i$ | uniform($[300, 500]$) | Size of computation (Kbit/s) |
| $D_i$ | uniform($[900, 1100]$) | Number of CPU cycles (megacycles) |
| $F$ | 5 | Server capacity (GHz/sec) |
| $W$ | 10 | Bandwidth for offloading devices (MHz) |
| $N_0$ | 1 | Gaussian channel noise |
| $f_i^{\mathrm{l}}$ | 1 | Local CPU capacity (GHz/sec) |
| $z_i$ | $10^{-27}(f_i^{\mathrm{l}})^2$ | Energy consumption per CPU cycle |
| $d_i$ | uniform($[1, 200]$) | Distance from MEC server (meters) |
| $P_i$ | 500 | Transmission power (mW) |
| $P_i^{\mathrm{s}}$ | 100 | Idle power (mW) |
| $I_i^{\mathrm{t}}$ | 0.5 | Decision weight delay |
| $I_i^{\mathrm{e}}$ | 0.5 | Decision weight energy |

Table A.1: Table of parameters utilized in MEC with values and descriptions.

## A.2 Parameters for Q-learning

| Parameters | Values | Descriptions |
|---|---|---|
| $\eta$ | 0.8 | Learning parameter |
| $\gamma$ | 0.95 | Discount factor |
| $\psi$ | 3 | Maximum number of steps |
| $\beta$ | 10 | Size of neighborhood |
| `numEpisodes` | 3000 | Number of training episodes |

Table A.2: Table of parameters utilized with values and descriptions for the Q-learning strategy.

## A.3 Parameters for DQN

| Parameters | Values | Descriptions |
|---|---|---|
| $\eta$ | 0.00025 | Learning parameter |
| $\gamma$ | 0.99 | Discount factor |
| `numEpisodes` | 3000 | Number of training episodes |
| $\epsilon$ | 1 | Starting epsilon value |
| $\epsilon_{\text{decay}}$ | 0.995 | Decay epsilon per episode |
| $\epsilon_{\text{min}}$ | 0.1 | Minimum epsilon value |
| $M$ | 512 | Batch size for sampling |
| $\zeta$ | 100 | Number of discrete allocation quantities of the server capacity |
| $b$ | 1000000 | Total buffer capacity |
| $C$ | 100000 | Frequency to update target network |

Table A.3: Table of parameters utilized with values and descriptions for the DQN strategy.

# Appendix B

# Numerical Results

## B.1 Total Cost

| $N$ | Local | Offload | Random | Optimal | Q-learning | DQN |
|---|---|---|---|---|---|---|
| 3 | 3.011 | 1.290 | 1.290 | 1.290 | 1.303 | 1.290 |
| 4 | 4.003 | 2.110 | 2.078 | 2.075 | 2.109 | 2.094 |
| 5 | 5.003 | 3.226 | 3.013 | 3.007 | 3.046 | 3.101 |
| 6 | 6.009 | 4.428 | 3.899 | 3.888 | 3.974 | 4.025 |
| 7 | 6.998 | 5.859 | 4.823 | 4.806 | 4.869 | 4.997 |
| 10 | 9.964 | 11.362 | 7.716 | 7.655 | 7.772 | 8.002 |
| 15 | 14.973 | 24.651 | 12.746 | 12.582 | 12.856 | 13.288 |
| 20 | 19.969 | 42.852 | 17.811 | 17.515 | 17.750 | 18.697 |
| 50 | 49.929 | 258.842 | 68.358 | – | – | 51.751 |

Table B.1: Numerical results between the different strategies divided by $10^9$ for different number of devices $N$.

## B.2 Server Capacity

| $F$ | Local | Offload | Random | Optimal | Q-learning | DQN |
|---|---|---|---|---|---|---|
| 1 | 20.002 | 204.496 | 20.492 | 19.479 | 19.479 | 19.543 |
| 3 | 20.033 | 70.030 | 18.790 | 18.522 | 18.591 | 18.903 |
| 5 | 19.951 | 42.828 | 17.786 | 17.512 | 17.690 | 19.010 |
| 7 | 19.990 | 31.636 | 17.047 | 16.650 | 17.034 | 19.152 |
| 10 | 20.057 | 22.842 | 15.901 | 15.390 | 16.069 | 18.492 |
| 12 | 19.959 | 19.463 | 15.159 | 14.517 | 15.313 | 19.115 |
| 15 | 20.030 | 16.049 | 14.210 | 13.405 | 14.359 | 16.327 |

Table B.2: Numerical results between the different strategies divided by $10^9$ for different server capacities $F$ with $N = 20$ devices.

# Bibliography


[1] Y. Zhang, *Mobile Edge Computing*. Simula SpringerBriefs on Computing, Springer, Cham, 2022.

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5G," *ETSI white paper*, pp. 1–16, Sept. 2015.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series, Cambridge, MA, USA: MIT Press, second ed., 2018.

[4] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Adaptive Computation and Machine Learning Series, Cambridge, MA, USA: MIT Press, 2016.

[6] P. J. Huber, "Robust estimation of a location parameter," in *Breakthroughs in Statistics: Methodology and Distribution* (S. Kotz and N. L. Johnson, eds.), pp. 492–518, New York, NY: Springer, New York, 1992.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations* (D. E. Rumelhart and J. L. McClelland, eds.), vol. 1, pp. 318–362, Cambridge, MA: MIT Press, 1986.

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." arXiv:1412.6980v9 [cs.LG], Dec. 2014.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning." arXiv:1312.5602v1 [cs.LG], Dec. 2013.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[11] R. Adams and C. Essex, *Calculus: A Complete Course*, ch. 14. Pearson, 2013.

[12] J. Wang, K. Liu, M. Ni, and J. Pan, "Learning based mobility management under uncertainties for mobile edge computing," in *Proceedings IEEE Global Communications Conference (GLOBECOM)*, (Abu Dhabi, UAE), pp. 1–6, Dec. 2018.

[13] Y. Hao, Y. Jiang, M. S. Hossain, M. F. Alhamid, and S. U. Amin, "Learning for smart edge: Cognitive learning-based computation offloading," *Mobile Networks and Applications*, vol. 25, pp. 1016–1022, 2020.

[14] K. Zhang, Y. Zhu, S. Leng, Y. He, S. Maharjan, and Y. Zhang, "Deep learning empowered task offloading for mobile edge computing in urban informatics," *IEEE Internet of Things Journal*, vol. 6, pp. 7635–7647, Oct. 2019.

[15] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digital Communications and Networks*, vol. 5, pp. 10–17, Feb. 2019.

[16] M.-H. Chen, B. Liang, and M. Dong, "Multi-user multi-task offloading and resource allocation in mobile cloud systems," *IEEE Transactions on Wireless Communications*, vol. 17, pp. 6790–6805, Oct. 2018.

[17] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance optimization in mobile-edge computing via deep reinforcement learning," in *Proceedings IEEE Vehicular Technology Conference (VTC-Fall)*, (Chicago, IL, USA), pp. 1–6, Aug. 2018.

[18] J. Li, Q. Liu, P. Wu, F. Shu, and S. Jin, "Task offloading for uav-based mobile edge computing via deep reinforcement learning," in *Proceedings IEEE/CIC International Conference on Communications in China (ICCC)*, (Beijing, China), pp. 798–802, Aug. 2018.

[19] H. Lu, C. Gu, F. Luo, W. Ding, S. Zheng, and Y. Shen, "Optimization of task offloading strategy for mobile edge computing based on multi-agent deep reinforcement learning," *IEEE Access*, vol. 8, pp. 202573–202584, 2020.

[20] Y. He, N. Zhao, and H. Yin, "Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach," *IEEE Transactions on Vehicular Technology*, vol. 67, pp. 44–55, Jan. 2018.

[21] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for MEC," in *Proceedings IEEE Wireless Communications and Networking Conference (WCNC)*, (Barcelona, Spain), pp. 1–6, apr 2018.

[22] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang, "Energy-efficient offloading for mobile edge computing in 5G heterogeneous networks," *IEEE Access*, vol. 4, pp. 5896–5907, 2016.

[23] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *Proceedings IEEE International Conference on Computer Communications (INFOCOM)*, (Orlando, FL, USA), pp. 2716–2720, Mar. 2012.

[24] J. Lee and S. Leyffer, *Mixed Integer Nonlinear Programming*, ch. 3. The IMA Volumes in Mathematics and its Applications, New York, NY: Springer, New York, 2011.

[25] M. Roderick, J. MacGlashan, and S. Tellex, "Implementing the deep Q-network." arXiv:1711.07478v1 [cs.LG], Nov. 2017.

[26] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks." arXiv:1612.00796v2 [cs.LG], Dec. 2016.

[27] H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep reinforcement learning and the deadly triad." arXiv:1812.02648v1 [cs.AI], Dec. 2018.

[28] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay." arXiv:1511.05952v4 [cs.LG], Nov. 2015.

[29] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning." arXiv:1509.06461v3 [cs.LG], Sept. 2015.

[30] H. van Hasselt, "Double Q-learning," in *Proceedings International Conference on Neural Information Processing Systems (NIPS)*, (Vancouver, BC, Canada), pp. 1–9, Dec. 2010.

[31] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning." arXiv:1511.06581v3 [cs.LG], Nov. 2015.