Teaching and Research Area Information Theory and Systematic Design of Communication Systems

ISEK

RWTH AACHEN UNIVERSITY

REINFORCEMENT LEARNING BASED DESIGN FOR EDGE COMPUTING NETWORKS

Jannik Vogt

Teaching and Research Area Information Theory and Systematic Design of Communication Systems

RWTH Aachen University

Teaching and Research Area Information Theory and Systematic Design of Communication Systems

ISEK

RWTH AACHEN UNIVERSITY

REINFORCEMENT LEARNING BASED DESIGN FOR EDGE COMPUTING NETWORKS

Jannik Vogt

A thesis submitted to the
Teaching and Research Area Information Theory and Systematic Design of Communication Systems,
RWTH Aachen University,

reviewed by Prof. Dr.-Ing. Anke Schmeink and supervised by Dr. Yulin Hu

Teaching and Research Area Information Theory and Systematic Design of Communication Systems

RWTH Aachen University

Eidesstattliche Versicherung

______________________________	______________________________
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Reinforcement Learning Based Design for Edge Computing Networks

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, den 29. November 2019	______________________________
Ort, Datum	Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, den 29. November 2019	______________________________
Ort, Datum	Unterschrift

Abstract

A a Mobile Edge Computing (MEC) network is considered supporting ultra-reliable low-latency communication based on Finite Blocklength (FBL) to satisfy strict latency constraints. We introduce an optimal framework design selecting servers and allocating time and workload. Due to imperfect knowledge about channel and server conditions in most realistic applications, we propose a deep Reinforcement Learning (RL) design for deciding on offloading decisions. First, we propose a deep Q-Learning based offloading scheme to the analytical framework in different channel and environmental scenarios. Due to scalability issues of action-value based RL methods, we propose a deterministic policy gradient method expanding the action space efficiently. Based on deep neural networks we introduce the Deep Deterministic Policy Gradient (DDPG) offloading scheme considering blocklength, server selection and workload. Via training and simulations we show convergence of the methods and compare the offloading policies to the analytical framework.

Keywords—Deep Q-Learning, Deep Deterministic Policy Gradient, MEC, URLLC.

Acknowledgements

First, I would like to thank Prof.Dr.-Ing Schmeink, for accepting this thesis and providing me with all the necessary resources for the study. Furthermore, I owe my gratitude to Dr. Yulin Hu for giving me the opportunity to work on this thesis as well as to Yao Zhu for giving me a lot of support, very helpful advise and well supervising throughout the process. At last, I wish to thank my family and friends for not only supporting and encouraging me through the whole thesis process, but also during my studies. Thanks.

Contents

Abbreviations

A2C	Advantage Actor Critic
A3C	Asynchronuos Advantage Actor Critic
AdaGrad	Adaptive Gradient Algorithm
CCDF	Conditional Cumulative Distribution Function
CSI	Channel State Information
DDPG	Deep Deterministic Policy Gradient
DNN	Deep Neural Network
DQN	Deep Q-Network
ECN	Edge Computing Networks
FBL	Finite Blocklength
FCN	Fully Connected Network
FIFO	First-In-First-Out
GPD	Generalized Pareto Distribution
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
MICP	Mixed-Integer Convex Problem
MIP	Mixed-Integer Problem
MSE	Mean-Square-Error
MU	Mobile Unit
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSProp	Root Mean Square Propagation
SAC	Soft Actor Critic
SGD	Stochastic Gradient Descent
SNR	Signal-to-Noise Ratio

TD	Temporal-Difference
UE	User Equipment
URLLC	Ultra-Reliable Low-Latency Communication

Notation glossary

$\mathbf{A} \in \mathbb{N}_{+}^{K}$	server selection matrix
$b \in \mathcal{B}$	action
b_{b}	bias of network
$\mathbf{C} \in \mathbb{N}_{+}^{K}$	workload matrix
$\mathcal{C}$	channel capacity
D_{mem}	replay memory
D_k	computing time
D'	rest queue delay
E	environment
f_k	computation power
F_{W_k}	CCDF
G	pareto distribution
h_k	channel gain
$\mathcal{H}$	number of episodes
J	start distribution
$k \in [0, K]$	server
L	loss of training
$m \in [0, M]$	time slot
m_0	latest received timeslot
M_{train}	number of training cycles
$M_{\mathrm{train,tot}}$	total number of training cycles
M_{test}	number of testing cycles
$N_{\wr}$	length of output

N_{hl}	number of hidden layers
$\mathcal{N}$	noise
n	blocklength
$\mathcal{P}_t$	transition probability function
$\mathcal{P}$	probability
Q	Q-Network/ critic network
r_c	coding rate
$r \in \mathcal{R}$	reward
$s \in \mathcal{S}$	state
s'	next state $s^{(m+1)}$
t_1	communication phase time
t_2	computation phase time
T_S	symbol Time
T	time delay
$T^{'}$	intermission time
$\mathcal{T}$	transitions
U	memory size of replay memory
$\mathcal{U}$	uniform distribution
U_{b}	batch size
V	value function
$\mathcal{V}$	channel dispersion
W_{ht}	number of historical timeslots
W_k	queue delay
y	Q-target value
y_{o}	output of network
$y^{'}_{\wr}$	expected output of network
α	learning rate
α_{ac}	learning rate of actor network
α_{crit}	learning rate of critic network

$\varepsilon_{1,k}$	error probability of communication channel k
$\varepsilon_{2,k}$	error probability of computation at server k
ε_O	overall error probability
$\varepsilon_{\max}$	maximal error probability
ϵ_{dec}	exploration decay
ϵ	exploration factor
γ_k	SNR of channel k
γ_{df}	discount factor
μ	actor function
μ_{OU}	mean-reversion-level
π	policy
ρ	channel gain correlation factor
ρ_0	state distribution
σ	scale parameter
σ_{ac}	activation function
$\sigma_{OU} \in (\sigma_{\max,OU}, \sigma_{\min,OU})$	factor of random variable
τ	soft copy factor of target networks
τ_{P}	packet Size
τ_{o}	trajectory of state-action transitions
θ	parameters of network
θ_{OU}	reversion speed
ξ	shape parameter
ω	random variable
$\mathbb{E}_\omega$	expectation over a function of a random variable

1 Introduction

Many of today's applications demand high level data processing putting a lot of computational load on the computer, especially on mobile devices with small processing power. These applications could be in the automotive industry, as in Vehicle-to-Everything or autonomous driving, where the car needs sensed information processed very rapidly [1] as well as in industrial automation, where machinery data needs to be analyzed for maintenance, security and production reasons [2]. Well known use-cases are also virtual video streaming and applications of Internet of Things [3]. Since in particular mobile devices do not have the computational capacity to process higher demanding applications, the need to offload those tasks rises. With Mobile Cloud Computing (MCC), the Mobile Unit (MU) or User Equipment (UE) can offload their tasks to a cloud server [4]. MCC helps to reduce the computational load on the MU, extending the battery lifetime and expanding data storage as well as giving access to high level applications. Because of the long spatial distance as well as logical hops, high delays between MU and cloud computer can occur [5]. In particular, autonomous driving and industrial automation rely on Ultra-Reliable Low-Latency Communication (URLLC), when offloading their tasks. URLLC is the standard defined as greater than 99.999% error probability by 3GPP [6]. A transmission is considered successful when the transmitted data is successful received in a certain time span. One possible solution is the use of Edge Computing Networks (ECN) or MEC for computing offloaded tasks. In ECN the server is located in close proximity to or at the edge of the network. The small spatial and logical distance leads to low transmission delays while having the same computational capacity as in MCC. While it has been shown that ECN can decrease the delay of computational processes, the transmission delay suffers from possibly bad channel conditions. The strict requirements regarding communication and computation invite to take the finite blocklength regime for transmission error probability as well as deadline violation probability due to waiting at the server's buffer into account. FBL, short and medium-block length codes, are suitable for transmitting small packet sizes e.g. in machine-type communication, smart metering networks, remote command links and messaging services [7–10].

For the ultra-reliable low-latency scenario the error probability correlates to the time delay limit [11]. The time delay limit includes both the communication phase between MU and MEC server as well as the computation phase of the offloaded task at the MEC server, a trade-off exists for the time allocation to the communication and computation phase. When increasing the time allocated to the communication phase, the time for the computation decreases as well as an increase of error probability in the computation phase. Between each offloading decision the quality of the communication channel can change. When selecting a channel with a bad condition, the overall error probability suffers, because the offloading process is only successful when every transmission is received error free. Accordingly, the servers computation state depends on the waiting queue at the servers buffer. Hence, selecting a server with a long waiting delay leads to a higher computational error probability. Therefore, a trade-off also exists in server selection where dividing the tasks workload to multiple servers could lead to using a bad channel. But selecting the same server, although having good channel conditions, may also lead to a long waiting delay. It has been shown that there is an optimal solution regarding the time allocation for the communication and computation phase, while dividing the total workload into equal slices [11]. The optimal solution is based on current knowledge of the channel. Since in most cases only outdated information is available we consider using a machine learning approach for deciding on the offloading task based on imperfect environmental knowledge. In particular we apply two deep RL methods to the presented problem and analyze its performance in comparison to the analytical solution. The objective of this thesis is to minimize the error probability in an MEC offloading scenario. Therefore, this thesis follows the following methodology:

- Research regarding the function of reinforcement learning and the different methods for deep RL is presented in Chapter 2. The ability of Deep Neural Network (DNN) to approximate functions is introduced as well. Furthermore, existing work regarding deep RL in offloading scenarios for ECN are summarized.

- Based on [11], the offloading model for the communication and computation phase is presented and numerical results are provided in Chapter 3.

- An offloading scheme based on Deep Q-Network (DQN) for the presented model is provided in Chapter 4. We analyze the methods ability to select servers for offloading in a static and fading channel using perfect state knowledge. We consider setups with different number of servers. Consequently, we look at the ability to learn an offloading policy in an outdated channel for different servers, but also for different channel gain correlations. Afterwards, the limitations of the DQN based design are discussed.

- Due to the limitations of DQN regarding scalability of the action space, we consider in Chapter 5 the DDPG based design to decide on blocklength, server selection and partitioned workload. We test the DDPG agent in a static and fading channel environment based on perfect Channel State Information (CSI) and observe its convergence behavior. Furthermore, we analyze the DDPG agents decisions based on outdated CSI and compare them to the analytical results as a benchmark.

- In Chapter 6, we conclude the presented results and point out challenges and improvements in designing a deep RL method for offloading decisions in MEC. At las we point at future work regarding offloading methods based on deep RL

2 Background

The thesis analyzes RL as a decision method for offloading parameterization in edge computing networks.Section 2.1 introduces the process and main types of RL. Further, two deterministic RL methods are presented. Because deep RL uses DNN the structure, process of learning as well as the parameters of neural networks are explained in Section 2.2. RL has been a focus of research as a solution for offloading decisions in edge computing networks. The last section Section 2.3 provides different existing work regarding Q-Learning and deterministic policy gradients in ECN.

2.1 Reinforcement Learning

RL has its roots in the early days of cybernetics, statistics, psychology, neuroscience and computer science [12]. RL has drawn more interest with the development in artificial intelligence, in particular with the progress in learning speed in deep neural networks. RL has the ability to solve a specific task without exactly knowing of how that goal is going to be achieved. The following section Section 2.1.1 introduce basis functions as well as different characterizations of RL. Section 2.1.2 presents the three main types of converging to an optimal policy: value-functions, policy search and actor-critic methods. At last, two deep RL methods, Deep Q-Learning in Section 2.1.3 and DDPG in Section 2.1.4, are presented.

2.1.1 Process

RL can be seen as a third form of machine learning aside from supervised and unsupervised learning. Other than supervised learning, RL does not compare its own results with the correct output. Instead, the agent receives a reward for its actions, which will be used for evaluating the agents policy. While learning to solve a problem, RL autonomously interacts with an environment and learns the optimal behavior [12]. The programmed agent seen in Fig. 2.1 observes an environment, where it receives an input in each time slot, depending on state s [12]. The state s describes the environment. If the agent has complete access to state s, the state

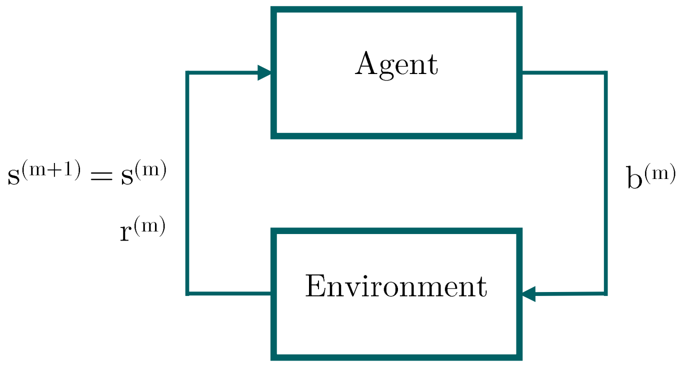


Figure 2.1: Reinforcement Learning Setup

s is completely observed. When the agent receives only the partial state as input, state s is partially observed. Based on the input, the agent chooses an action b. The action b leads to the next state $s^{(m+1)}$. Depending on the satisfaction of the outcome of action b, the agent receives a actual reward or punishment, which is implemented assigning a lower or even a negative value. The satisfaction of the action is described by a reward function. The objective of the agent is to learn the optimal policy π, which maximizes the overall expected reward, which is defined as the following [13]:

$$\pi^* = \arg\max \mathbb{E}[R|\pi] \tag{2.1.1}$$

where R is the overall reward. The policy π needs to be adjusted in order to maximize the long-term reward $\mathbb{E}\sum_{m=0}^{\infty}\gamma_{\text{df}}^{(m)}r^{(m)}$. The $\gamma_{\text{df}}r$ is also called the discounted reward with discount factor $\gamma_{\text{df}}\in [0,1)$. Using a discount factor ensures that the return convergence. The policy π determines how the agent chooses the actions based on the incoming state. The sequence of state and actions is the trajectory $\tau(s^{(0)}, b^{(0)}, s^{(1)}, b^{(1)}, ...)$, which can be represented by a Markov-Decision-Process consisting of 5-tuple$[\mathcal{S}, \mathcal{B}, R, \mathcal{P}_t, J]$:

- set of states $\mathcal{S}$
- set of actions $\mathcal{B}$
- $R : \mathcal{S}, \mathcal{B}, \mathcal{S} \rightarrow R$ is the reward function, with $r^{(m)} = R(s^{(m)}, b^{(m)}, s^{(m+1)})$
- $\mathcal{P}_t : \mathcal{S}, \mathcal{B} \rightarrow \mathcal{P}_t(\mathcal{S})$ is the transition probability function
- is the start state distribution J

In order to make a decision, the agent needs to know all possible actions at all states and which action has the expected reward. There are two different approaches for an agent to optimize its policy. One is the model-based method, where all possible states, actions, state transitions and rewards are calculated before the agent interacts with the environment and the agent acts accordingly to the results [12]. The model-free method explores all action while interacting with the environment. It is a trial and error approach. The agent has no prior knowledge and has to learn about the best policy while interacting with the environment.

Another aspect of reinforcement learning is how the agent adapts its policy in order to reach the optimal solution. On-policy refers to learning procedures which take the policy of past actions into account to generate the new policy [12]. With on-policy learning, the actual policy influences the training samples used for adjusting the policy, which can lead to a local minimum or divergence. Off-policy is ignoring the actual policy, which leads to less local minima.

In particular for model-free RL, the trade-off between exploration and exploitation determines if the agent converges to a suitable solution. In exploration, the agent discovers new state transitions while using different actions, which have not been evaluated yet and the agent can find the optimal policy [12]. An agent being too dependent on exploration can lead to a slow convergence of the agent to the optimal value. Exploitation describes the process where the agent solely relies on its experienced knowledge. State transition, which are memorized as the best actions, will be reevaluated. The downside is that only known transitions are evaluated. So a high exploitation can lead to missing the optimal policy, but to a faster convergence speed. There are different techniques on how to deal with the trade-off. A greedy strategy always chooses the highest valued action at each time step. The randomized strategy chooses the highest valued action, but with probability ϵ the agent takes a randomized action. A hybrid strategy is the ϵ-greedy exploration strategy, where in the beginning the agent chooses a randomized action with a high probability ϵ. During learning, the probability ϵ decreases and the agent turns more to exploitation. How exploration is achieved, depends on the used RL algorithm and will be explained for each used method separately in the following chapters Section 2.1.3 and Section 2.1.4.

2.1.2 Model-Free Methods

The RL agent can be implemented using a deterministic or stochastic solving approach. Further, we focus almost exclusively on deterministic methods. There are three main methods on how to reach the optimal policy for model free reinforcement learning: Policy based, action-value function and actor-critic methods. In the following, the three types of RL are introduced.

Action-Value Function

Value function and action-value functions are both used for deterministic policy representation. Both functions represent the value of being in a state s or using an action b. Value functions or state-value functions estimate the value being in a certain state s. The value function V^π describes the expected return when starting in state s and continuing with policy π. The value function can be formally expressed as the following:

$$V^\pi(s) \quad = \quad \mathbb{E}[R|s,\pi] \tag{2.1.2}$$

To learn from previous state-action transitions, value functions use the Temporal-Difference (TD) method for evaluating the value function, where the TD error is denoted as $r + \gamma_{df} V(s^{m+1}) - V(s^m)$. TD learning updates the value-function iteratively and online as [14]:

$$V(s) \quad = \quad \leftarrow V(s) + \alpha[r + \gamma_{df} V(s^{(m+1)}) - V(s^{(m)})] \max V^\pi(s) \tag{2.1.3}$$

Because state transition are not known, it is too complex to compute the state-values efficiently. An easier implementation is to use state-action-values or the quality function $Q^\pi(s,b)$, which is formulated as:

$$Q^\pi(s,a) \quad = \quad \mathbb{E}[R|s,b,\pi] \tag{2.1.4}$$

The Q-value indicates the expected value of being in state $s$and using action b. The action value will also be updated according to the TD rule. The Q-values and state-action function are used in Q-Learning, which will be introduces more detailed later in Section 2.1.3.

Policy-Based Methods

Other than action-value functions, the policy based methods directly search for the optimal policy π^*. Policy-Based methods have the advantage of executing continuous action spaces. The parameters of the policy will be updated by either a gradient-based or gradient-free methods. For designing a policy for continuous action space probability distributions are mostly used. For discrete actions, individual probabilities of multinomial distributions are more suited. Gradient-free methods on the other hand require heuristic search to find the optimal policy. Gradient-free methods have the advantage to optimize non-differentiable policies. The policy gradients is a good figure of merit on how to update the policy parameters. For computing the gradient the agent needs to average over state transitions of the current policy parameterization. A deterministic approximation requires a model-based setting, where the state transitions are known before. For model-free applications a stochastic approximation is often used. The most common policy evaluator is the policy gradient [15]. A Monte-Carlo approximation can provide the expected return Q. Because of the stochastic nature, a gradient estimator, the reinforce rule is used. The reinforce rule computes the gradient of an expectation over a function f of a random variable ω with respect to parameters θ [16]:

$$\nabla_\theta \mathbb{E}_\omega[f(\omega;\theta)] \quad = \quad \mathbb{E}_\omega[f(\omega;\theta)\nabla_\theta \log \mathcal{P}(\omega)] \tag{2.1.5}$$

In the case of policy based reinforcement learning, θ represents the parameters of the policy and X can be substituted with action b.

Actor-Critic Methods

A combination of the aforementioned methods is the actor-critic method, which combines the ability of using continuous action space as well as a critic allowing a faster convergence. An actor will execute the policy, while being evaluated by a critic. θ is the actors policy parameter, which will be updated by using stochastic gradient descent. The critic estimates the action-value function, while being evaluated by the temporal-difference algorithm. The actor-critic gradient is derived from the policy gradient theorem and is written as [15]:

$$\nabla_\theta J(\pi_\theta) \quad = \quad \mathbb{E}_{s\sim\rho_0^\pi, a\sim\pi_\theta}[\nabla_\theta \log \pi\theta(a|s) Q^{\theta_Q(s,a)}] \tag{2.1.6}$$

where J is the start distribution, θ_Q denotes the critics parameters and $\rho_0^\pi(s)$ is the state distribution, while depending on the policy parameters, the policy gradient does not depend on the gradient of the state distribution.

2.1.3 Deep Q-Learning

Q-Learning and Deep Q-Learning are action-value reinforcement learning methods based on the TD algorithm and the Bellman equation. In Section 2.1.3 both algorithms as well as the reason of using deep neural networks for function approximation is explained. With the development of deep neural networks, Deep Q-Learnings use has been experienced in games from the Atari 2600 collection [17]. It was shown that an agent can quickly learn solving a game with a limited action space. Further in Section 2.1.3, Double Deep Q-Learning is presented as an improvement in convergence speed and overall reward to the previous method.

Q-Learning Using Deep Neural Networks

Q-Learning is a model-free deterministic off-policy reinforcement algorithm. It is used for agents to learn the optimal policy in a stochastic environment. $Q'(s,b)$ is the expected discounted reward for taking action b in state s. $V(s)$ is the value of being in state s, assuming that optimal b is taken initially and $V^*(s) = \max(Q^*(s,b))$, as well as $\pi^*(s) = \arg(\max(Q^*(s,b)))$. So the Q-value can be estimated with s' being the next state and a' the best expected action of state s' using the Bellman equation [18], as:

$$Q(s,a) \quad = \quad R(s,a) + \gamma \sum(\mathcal{T}(s,b,r,s') \max(Q'(s',b'))) \tag{2.1.7}$$

Therefore, the Q-learning rule for multiple iterations can be formulated as:

$$Q(s,a) \quad = \quad Q(s,b) + \alpha(y - Q(s,b)) \tag{2.1.8}$$

where (s,b,r,s') is the experience tuple, α is the learning rate and $y = r + \gamma \max(Q(s',b')$ defines the target value. If each action in each state is used an infinite number of times, Q converges to Q^*.

Because of the high complexity, the problem would become intractable for conventional use of a table for finding the Q-value. Hence, with the advancements of deep neural networks, one is able to use function approximation with a deep neural network [19]. A deep neural network has in contrast to a shallow network, multiple hidden layers besides the input and output layer. The network is parameterized by θ_i, which are the weights between the single neurons. These weights determine the output according to the input and will be changed when adjusting the Q-network. RL can be unstable when a non-linear function for the action-value function is used. Additionally, because the correlation in observed sequences impacts also the change in Q-values, as well as the correlation between action-values and the target function, experienced replay is used when training the agent [19]. As shown in Fig. 2.2, with experienced replay a batch of experience 4-tuple (s,b,r,s') is randomly taken from the agents memory. The batch is used to adjust the weights of the Q-network according the Q-learning rule. The random batch removes high correlation between sequences, which leads to more diversified data. A second network, identical to the Q-network, is used as the target network. Using Q-value and target-value from the same network can lead to a slower convergence or in the worst case to divergence, because the update of the Q-network follows its own target Q value. Hence, Deep Q-Learnings update while using a target network at iteration i uses the following loss function [19]:

$$L_i(\theta_i) = \mathbb{E}_(s,b,r,s') \sim \mathcal{U}(D_{\text{mem}})[(r + \gamma_{\text{df}} \max Q(s',b';\theta_i^-) - Q(s,b;\theta_i))^2] \tag{2.1.9}$$

where γ_{df} is the discount factor, determining the weight of future reward. $\mathcal{U}(D_{\text{mem}})$ represents the uniformly randomly drawn batch samples from the agents memory D_{mem}. θ_i^- are the parameters of the target network, which will be updated every τ time steps and will be held fixed otherwise.

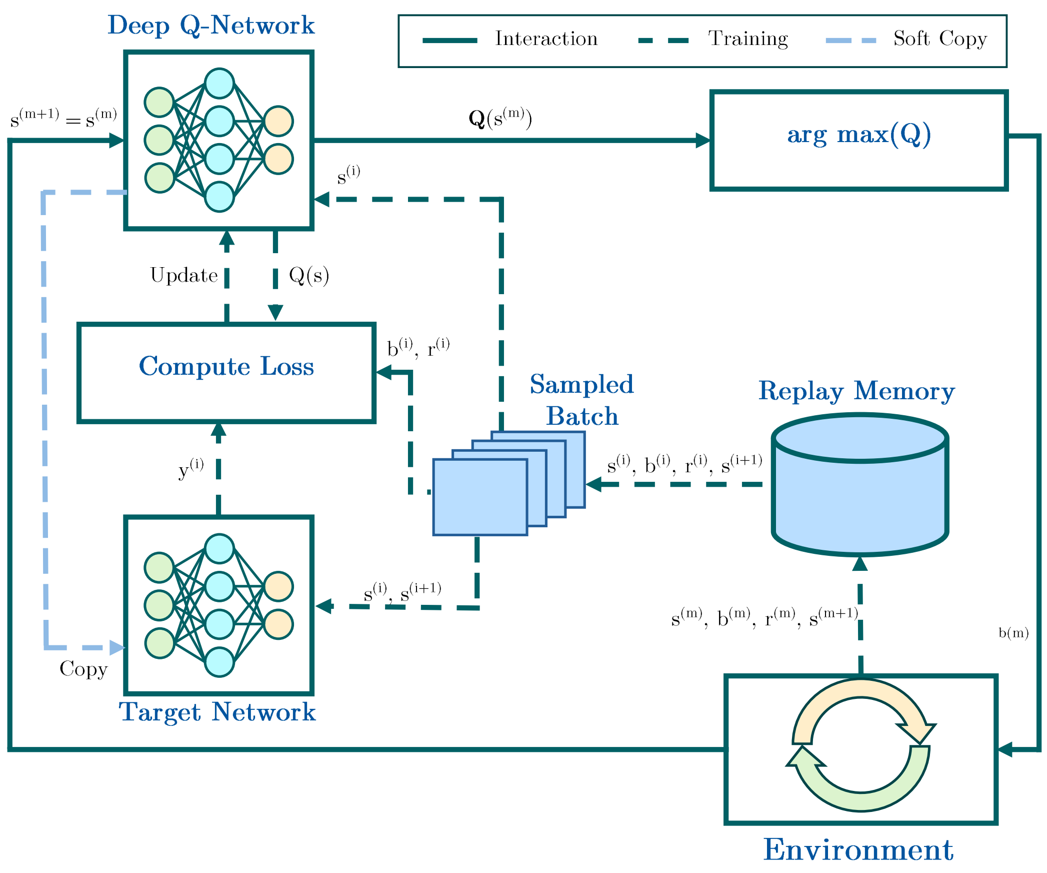


Figure 2.2: Architecture Deep Q-Learning

Double Q-Learning

Conventional Q-Learning using a single estimator can have a poor performance because of the overestimation of the state-action value Q. To avoid this overestimation, double Q-Learning is introduced with two estimators for the state-action value [20]. The double Q-learning uses two Q-values Q^A and Q^B at the same time. When updating the Q function, the other Q function is used. So Q^A updates as $Q^A = \max_b Q^B(s^{'}, b^{*})$. For the method to work, both Q-functions need to use different experience samples when training the functions. While training, both Q-functions can be used, which means that the method should not be less data efficient as in conventional Q-learning when choosing actions. Another way is to take the average of both Q-values, when deciding on the action. It is shown that Double Q-learning converges to the optimal policy π^* [20]. The double Q-learning can be transfered to Deep Q-Networks where the target network is used as the second network. The target network evaluates the action selected for the target value. The DQN selects the action for $s^{(m+1)}$. With the separation of the maximization the Bellman equation into action selection and action evaluation the overestimation of

Algorithm 1 Double Q-Learning Algorithm

```
Initialize $Q^A$, $Q^B$, $s$
for k=1,2,3,... do
    Choose $a$, based on $Q^A(s,.)$, $Q^B(s,.)$, observe $r$, $s^{'}$
    Choose either UPDATE(A) or UPDATE(B)
    if Update(A) then
        Define $a* = argmax_b Q^A(s', b)$
        $Q^A(s,b) \leftarrow Q^A(s,b) + \alpha(s,a)(r + \gamma Q^B(s', b*) - Q^A(s,b))$
    else
        if Update(B) then
            Define $b* = argmax_b Q^B(s', b)$
            $Q^B(s,b) \leftarrow Q^B(s,b) + \alpha(s,b)(r + \gamma Q^A(s', b*) - Q^B(s,b))$
    $s \leftarrow s'$
```

the Q-value is reduced. The target value is therefore [21]:

$$Y_{DD}^{(m)} = r^{(m+1)} + \gamma Q(s^{(k+1)}, \arg(\max Q(s^{(m+1)}, a; \theta^{(m)})), \theta^{-(m)}) \tag{2.1.10}$$

2.1.4 Deep Determinstic Policy Gradient

Value based algorithms have a good convergence for discrete action space, but when learning the Q network for a larger discrete or even a continuous action space, the networks convergence speed as well as the accuracy can suffer due to increase of dimensionality. For most action-value algorithms the common approach is a maximization of the action-value function Q, with $\arg\max Q^{\mu^k}(s,b)$. For a continuous action space a greedy maximization could be too computationally complex, because it requires a global maximization at every time step [22]. It is computationally simpler to lead the policy in the direction of the gradient of Q. Hence, at every time step k the policy parameters θ^{k+1} are updated according to $\nabla_\theta Q^{\mu^k}(s, \mu_\theta(s))$. The Actor-Critic algorithm combines the advantages of a value function with a policy gradient approach. The DDPG method is a model-free, off policy actor critic algorithm, which uses neural networks for function approximation.

DDPG uses a standard reinforcement learning setup with an agent observing an environment E with its state s, deciding on action b, receiving reward r and observing the next state $s^{(m+1)}$ [23]. Opposite indexing in value functions, the actor decides on a real value $b^{(k)} \in \mathbb{R}$ per action b. As in Deep Q-Learning the reward is defined as $R^{(m)} = \sum_m^M \gamma^{(m)} r(s^{(m)}, b^{(m)})$. The agents behavior is implemented in the actor function $\mu(s|\theta^\mu)$, which parameterizes policy μ. Since DDPG is a deterministic method the policy is obtained as μ, instead of π, which is mostly used for a stochastic policy as well as action-value functions. A deep neural network approximates the actor function and θ represent the networks weights. As shown in Fig. 2.3, the network maps state s as inputs to action values b as outputs. The critic $Q(s, a)$ works similarly to Deep Q-learning as the Q-network and is trained using the Bellman equation and the TD method. The actor is learned by applying the chain rule to the expected return from the start distribution J with respect to the actors parameters, which results in the policy gradient [23] [22]:

$$\nabla_{\theta}\mu J \approx \mathbb{E}_{s^m \sim \rho_0}[\nabla_a Q(s, a|\theta^Q)|_{s=s,a=\mu(s^{(m)})} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s^{(m)}}] \tag{2.1.11}$$

The critics Q-network maps the states and actions to a Q-value for every action. As in Deep Q-Learning, using function approximations means that convergence is not guaranteed anymore. Therefore, experienced replay is used where a random mini-batch from a buffer is chosen to train the networks. The benefits from a large replay buffer is to help learn from a large set of uncorrelated state transitions. DDPG uses target networks to prevent instability, but opposite to Q-learning the agent copies a fraction of the networks weight values, also called soft copies, to the target networks. So the DDPG agent has a target actor network $\mu^-(s|\theta^{\mu^-})$ and a target Q-network $Q^-(s, b|\theta^{Q'})$. The target networks weights are slowly updated by using τ as the update parameter for $\theta^- \tau\theta + (1 - \tau)\theta^-$ with $\tau << 1$. The parameter forces the target networks to slowly learn the target value, which greatly improves the convergence of the networks.
The action policy is then composed as $\mu'(s^m) = \mu(s^m|\theta^{\mu,(m)}) + \mathcal{N}$. To explore the whole environment for a continuous action space, noise $\mathcal{N}$ is added to the decided action in the training phase. For the noise process, an Ornstein-Uhlenbeck process is used to produce correlated exploration [24]. For an efficient exploration the mean reversion level μ_{OU}, reversion speed θ_{OU} and the influence σ_{OU} of a random variable $\omega^{(m)}$ have to set by the system designer. The characteristic of the Ornstein-Uhlenbeck process leads to a high noise variance at the beginning until it converges to μ_{OU} with reversion speed θ_{OU}. The Ornstein-Uhlenbeck process is formulated as [24]:

$$X'^{(m)} = \theta_{OU}(\mu - X^{(m)})dt + \sigma\omega'^{(m)} \tag{2.1.12}$$

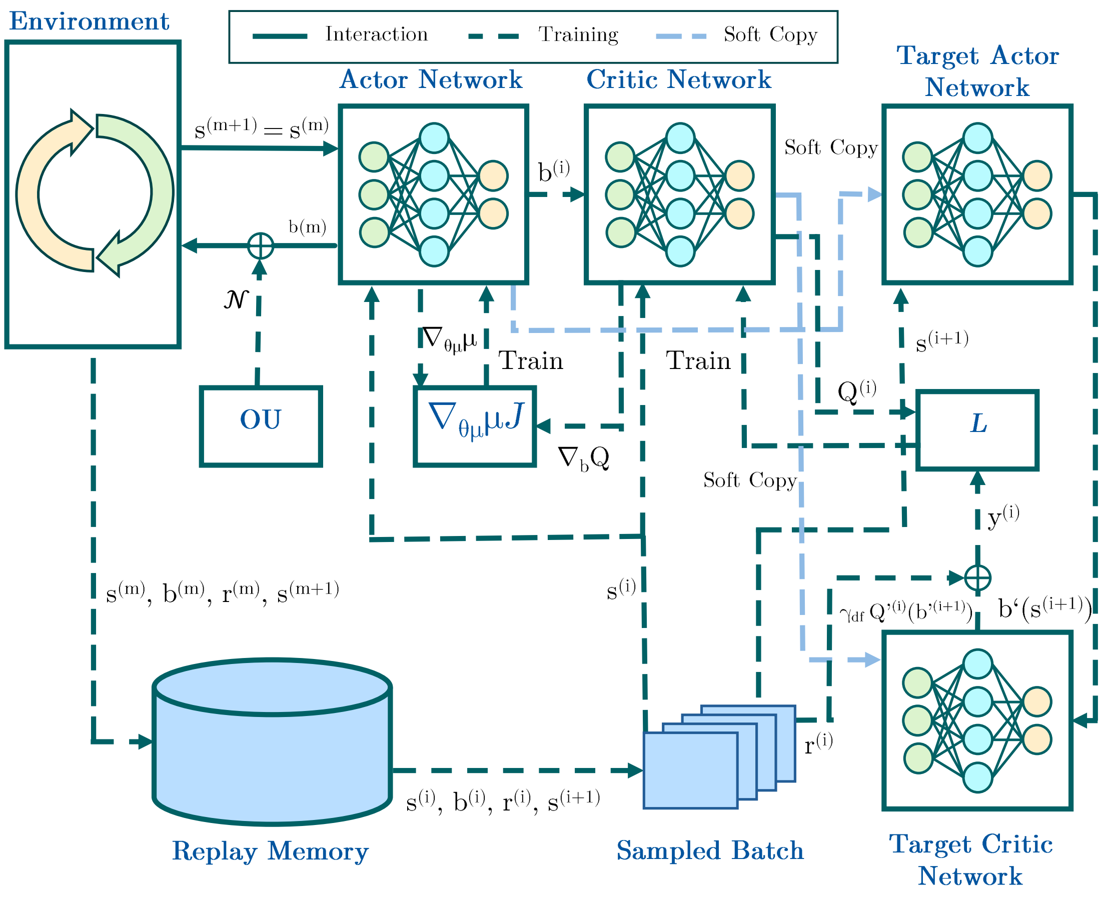


Figure 2.3: Architecture Deep Deterministic Policy Gradient

2.2 Deep Neural Networks

Artificial neural networks are based on the structure of neurons and synapses of real brains. In 1960, scientists found a part of the brain of a cat, which is responsible for vision, neurons, which are activated by receiving certain characteristics in the environment [25]. Based on these findings, biological neural networks are used as a model for building computational networks, which behave similar and create artificial intelligence. In Section 2.2.1, the setup of artificial neural network, activation functions and its hyperparameters. In Section 2.2.2, training algorithms and the advantages of DNN for function approximation are presented.

2.2.1 Neurons and Activation

The most fundamental neural network is a perceptron, which has a arbitrary number of inputs. The weighted sum of these inputs will determine the output of an activation function [26]. The calculation of the activation function is the output y_o of the perceptron, which can be expressed as:

$$y_\mathrm{o} = \sigma_\mathrm{ac}(z_j^l) \tag{2.2.1}$$

with $z_j^l = \sum_j \theta_j x_j + b_\mathrm{b}$. θ_j is the weight of input x_j. Where, in a multi-layer perceptron, x_j could be the output of a prior perceptron. The functionality of the perceptron is influenced by the weight and bias b_b, where the bias is nothing more than an offset. Hence, the behavior of the network is mostly determined by the type of activation function used in the layers.

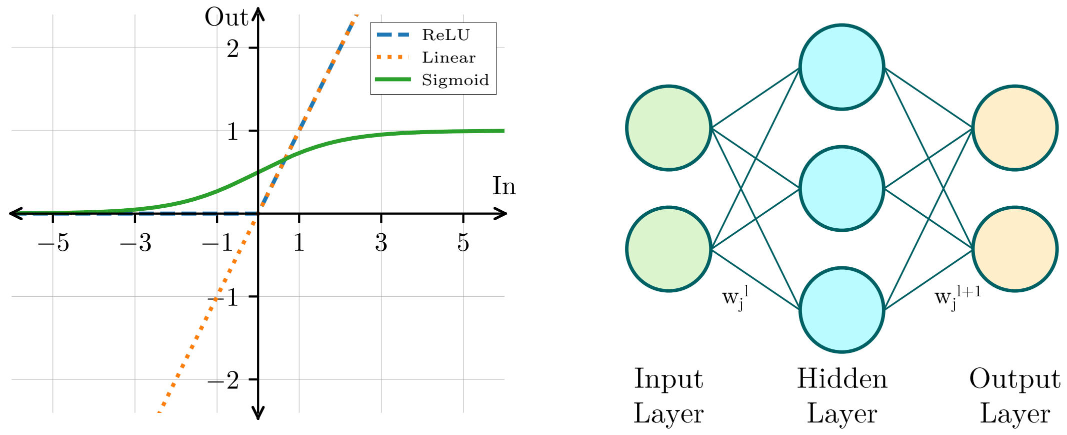
Activation Function

The activation function σ_ac determines how the layer responds to the input. The most important activation functions are shown in Fig. 2.4a. The sigmoid function is a smoothened threshold method, which has the advantage to a normal threshold that it can be differentiated [26], and is formulated as $\sigma_\mathrm{ac} = \frac{1}{1+\exp(-z_j^l)}$. Another function with a threshold is the Rectified Linear Unit (ReLU). ReLU is a linear function for values above zero. Because in Q-Learning function values can reach negative values, a linear function over the real value space has its use as well.

Neural Networks

A neural network consists of multiple layers of connected neurons also called the Fully Connected Network (FCN). The FCN consists of an input layer, hidden layer and an output layer as shown in Fig. 2.4b. Between each layer all neurons of one layer are connected to the neurons of the previous and the next layer [26]. The input layer has the same size as the number of inputs a network should receive. There can be multiple hidden layers and the number of neurons a layer inherits is arbitrary to the designer. It should be noted that the more neurons are used the more detailed and complex information can be extracted by the network. A higher number of neurons and hidden layer can also lead to a longer convergence of the network. The output layer's size depends on the number of different classes needed for solving the problem. In a Q-network the number of output neurons is equal to the size of the action space. When designing a network, multiple parameters have to be set to characterize the network. These parameters are called hyperparameters.

Hyperparameters, which are used in this tesis, include the learning rate, number of hidden layers and batch size.



(a) Activation Function (b) Deep Neural Network

Figure 2.4: Architecture of Deep Neural Network and used Activation Functions

2.2.2 Training via Backpropagation

To update the network, the loss has be determined in order to change the network weights accordingly. The loss L can be calculated using different functions. In this study the Mean-Square-Error (MSE) is used as the loss measure:

$$L(\theta, b_\mathrm{b}) \quad = \quad \frac{1}{2N_\mathrm{o}} \sum_x ||y'_{\rtimes}(x) - y_o(x, \theta, b_\mathrm{b})||^2 \tag{2.2.2}$$

where $y'_{\rtimes}(x)$ is the expected output of the network and $y_o(x)$ the real output of length N_o. Backpropagation is a procedure which calculates the derivative of the loss with the help of backpropagation from layer to layer [26]. First of all, the method determines the change of the loss function with regards to the weighted input, as $\delta L/\delta z_j^l$. Because of the direct impact of changing the activation function of the output on the change of the loss function, the change can be expressed as:

$$\delta_j^l = \frac{\delta L}{\delta z_j^l} = \frac{\delta L}{\delta a_j^l} \sigma'_\mathrm{ac}(z_j^l) \tag{2.2.3}$$

Based on the result of Section 2.2.2, the derivative of the layer can be computed. Accordingly the derivative of the $(l+1)$th layer is:

$$\delta_j^l = \sum_g (\theta_{gj}^{l+1} \delta_g^{l+1}) \sigma'_\mathrm{ac}(z_j^l) \tag{2.2.4}$$

where k is layer $l-1$. Using the backpropagation method, weights of the network can be efficiently adjusted. The main method used in this thesis responsible for the network update is the Adam optimizer. The fundamental algorithm used in the Adam methods is the Stochastic Gradient Descent (SGD). The aim is to minimize the loss function. The weights and biases are adjusted using the gradient of each loss at each weight. The weights are adjusted so that the loss function is minimized the most, as following [26]:

$$\theta_{g,l} \leftarrow \theta'_g = \theta_g - \alpha \frac{\delta L}{\delta \theta_g} \tag{2.2.5}$$

$$b_{\mathrm{b},l} \leftarrow b'_{\mathrm{o},l} = b_l - \alpha \frac{\delta L}{\delta b_{\mathrm{b},l}} \tag{2.2.6}$$

where α is the learning rate, which inherits the step size of the gradient descent. The learning rate is an important parameter, which can determine the performance of the network. If the learning rate is too high, the loss oscillates around the minimum, which can lead to a disconvergence. A lower learning rate can mean that the minimum is found too slowly.
The Adam optimizer is a kind of method to the stochastic gradient descent, because stochastic gradient descent keeps a single learning rate for all weight updates and does not change the learning rate during training [27]. The Adam optimizer maintains a learning rate for each network weight and adapts as learning continues. The Adam optimizer can be separated into two other methods, the Adaptive Gradient Algorithm (AdaGrad) and the Root Mean Square Propagation (RMSProp). The AdaGrad keeps a learning rate for each weight that improves performance on problems with sparse gradients. The RMSProp also maintains learning rates for each weight, which are updated based on the average of current gradients for the weights. The Adam optimizer adapts the learning rates based on the uncentered variance of the gradients. It calculates the exponential moving average of the gradient and the squared gradient. With the adjusted learning rate for each weight, the Adam optimizer improves learning speed in particular for larger data set [27].

Function Approximation

In reinforcement learning, deep neural network are used to map from states to actions or to represent the value of certain actions for a state. The use of conventional functions or Q-tables would become too complex to adapt to the problem. The DNN has to approximate a function, which fits the optimal behavior desired for a certain policy. The advantage from DNN to shallow networks is that a shallow network needs an exponential larger number of neurons to approximate the same function as DNN [28]. For the DNN to approximate a function, it needs a sufficient number of hidden layers and neurons. The performance of the approximation depends also on the type of activation function [29]. A network with many layers can better approximate a function with many oscillations than a network with less layers [30].

2.3 Deep Reinforcement Learning in ECN

Deep Reinforcement Learning has been subject of multiple research studies for Edge Computing Offloading scenarios. Its shown that offloading and resource allocation decision based on deep reinforcement learning outperforms conventional offloading decisions like greedy or local allocations [31]. The agent uses the Deep Q-Learning algorithm, while a deep neural network approximates the Q-function. The model takes the number of arriving tasks, size and channel information into account. The agent has perfect information on the environment and executes its discrete and limited number of actions for resource offloading and allocation to a MEC server. Another study has researched deep RL for MEC in a moving vehicle scenario where the agent not only improves the systems performance by using computation but also integrating network and caching decisions [32]. Another study shows that deep Q-Learning outperforms conventional Q-Learning in offloading and allocation of resources in an MEC environment [33]. In [34], DQN outperforms conventional offloading algorithms, equal-offloading or random-offloading, in an ultra-reliable low latency environment. The model takes communication error probability and end-to-end delay into account. As mentioned Section 2.1.3, double Q-Learning improves convergence speed and overall performance of a RL agent. Research has also shown that a method consisting of double Q-Learning improves a MEC environment system, based on current information about server and channel [35]. Another study examines conventional Q-Learning together with a DNN, where the network computes the action and the Q-function calculates the value of the given action at the current state [36]. The scalability of action and state space is a disadvantage for Deep Q-Learning, because with an increasing number of actions the convergence speed and decision performance decreases. Hence, for applications with a need for higher action space or even continuous values for actions, actor-critic algorithms are more suitable. In DDPG the actor can continuously allocate resources to both local and server computation [37]. The agent works also under the observation of perfect channel information. Mostly the deep RL agents decided on simple offloading and resource allocation. In [38], a DDPG agent decides not only computation and caching resources but also on the bandwidth for offloading tasks to the MEC node in a scenario with moving vehicles.

All of the presented studies use perfect channel state information as well as perfect information about the state of the computation server. They decide on which servers are selected for the best performance. Some also divide the resources to local and external computation. Extensive decisions on the end-to-end performance dependent parameters could not be found during research.

3 End-to-End Error Probability for Multi-Server ECN

The following chapter Section 3.1 presents a model for ultra-reliable low-latency offloading in ECN, which consists of a communication and computation phase. Furthermore, the analytical solution shows the convexity and optimality of the solution in Section 3.2. At last a numerical simulation of the solution is shown in Section 3.2.5.

3.1 Model

A MEC network with K available servers $\mathcal{K} = \{1, .., K\}$ is supporting one user equipment (UE) for processing computer-intensive applications. The setup of the network is presented in Fig. 3.1. Because of the insufficient computation power at the UE, tasks have to be offloaded to a server and computed remotely. After every time-slot m of length T the process has to be finished. The time is slotted into frames, where each frame is divided into three phases: communication phase of length t_1, computation phase of length t_2 and the feedback phase with length $\bar{t}$. During communication phase the UE transmits π to some of the available servers through wireless channels, which includes input data for the tasks as well as the server selection. In the computation phase, the selected servers compute the instructed tasks with workload c_k. After computing the task, the MEC sends the result back to the UE in the feedback phase. But because the transmit power of the servers is generally high and the data size of the computation result is low, the length of $\bar{t}$ can be neglected in comparison to T. We consider in the following model and optimization model $\bar{t}$ to be a very small constant. Hence, the the frame length $T = t_1 + t_2$, is the sum of the length of computation phase and communication phase. When regarding this framework optimization, t_1 and t_2 while satisfying T are determined.

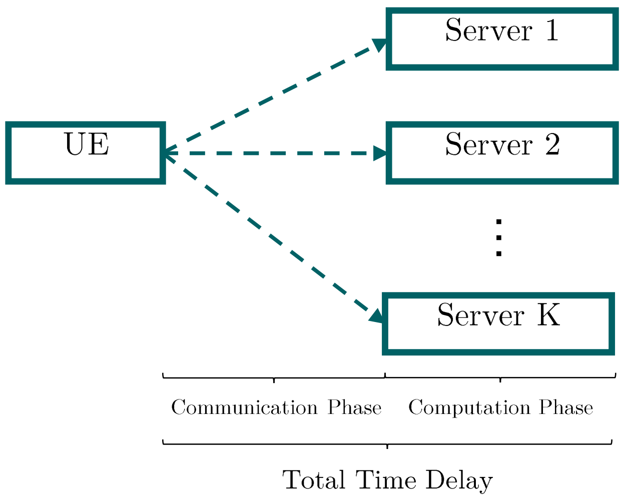


Figure 3.1: ECN Offloading Environment

3.1.1 Communication Error

For communication parameterization, we consider T_S as the duration of one symbol, which leads to the communication blocklength by $n = \frac{t_1}{T_S}$. With π being the packet size transmitted through a wireless channel, the according coding rate is calculated by $r_c = \frac{\pi_P}{n}$(in bits/symbol). When modeling the wireless channels, we consider independent channels to each other as well as Raleigh Fading channels. The average channel SNR is $\bar{\gamma}$, where every for every time slot the signal-to-noise ration of channel k is given by $\gamma_k = \bar{\gamma}\Box h^{(m)}$, with $h^{(m)}$ being the channel coefficient or channel gain. Because of the correlation of neighboring frames, the Jakes model of correlation is used [39]:

$$h^{(m)} = \rho h^{(m-1)} + \sqrt{1-\rho^2}\hat{h}^{(m)} \tag{3.1.1}$$

where ρ is the channel correlation factor and $\hat{h}^{(k)}$ is a complex Gaussian random variable with zero mean and unit variance. When considering the finite blocklength transmission model, the error probability for a channel to server k is given by:

$$\varepsilon_{1,k} = \mathcal{P}(\gamma_k, r, n) \approx Q\left(\sqrt{\frac{n}{V_{c,k}(\gamma_k)}}(\mathcal{C}_k(\gamma_k) - r)\log_e 2\right), \tag{3.1.2}$$

with $\mathcal{C}_k = \log_2(1+\gamma_k)$ being the Shannon capacity. Moreover, $V_{c,k}(\gamma_k)$ is the channel dispersion between the UE and the server k under a complex AWGN channel, $V_{c,k} = 1 - \frac{1}{(1+\gamma_k)^2}$.

3.1.2 Computation Error

In the computation phase, the incoming tasks are only computed for successfully received packets from the UE. The selected servers are then computing the tasks according to the decision vector for the server selection, as:

$$\mathbf{A} = \{a_1, ..., a_K\}, \tag{3.1.3}$$

where a_k is the computation decision of server k with $a_k \in 0, 1$. $a_k = 1$ indicates that the UE is offloading to server k whereas $a_k = 0$ indicates that the UE is not offloading to server k. The workload to be computed at each time slot is denoted as c_o and f_k is the computation power of server k. With $\mathcal{K} = \{k \big| \forall a_k = 1\}$ being the set of selected servers, the size of $\mathcal{K}$ is $\sum_k a_k$. Due to selecting multiple servers for offloading and computation of the task, the workload c_o needs to be divided as well. Hence, we denote by $\mathbf{C} = \{c_1, ..., c_K\}$ the assigned workload vector. c_k is the assigned workload for server k. The task is only finished when the whole workload is computed. Therefore, the workload assignment decision needs to fulfill the following conditions: 1) all workload needs to be assigned to the servers 2) the assigned workload may not exceed the total workload c_o 3) none of the workload may be assigned to the non-selected servers. Hence, the conditions are given by $c_o = \sum_k c_k$ and $c_k \leq a_k c_o$, where $k \in \mathcal{K}$.

For the computation we consider the computing time D_k decomposed as:

$$D_k = \frac{c_k}{f_k} + W_k, \tag{3.1.4}$$

where $W_k = D'_k + Q_k$ is the queuing delay at server k. The queuing delay occurs from the First-Come-First-Serve policy of the computation system. Only when the previous tasks are finished computing, the following task can be computed. Therefore the computing delay is the sum of computing time and queuing delay. Q_k indicates the delay for serving arriving tasks in the current frame k. D' is the rest delay of the previous time slot, which is expressed by $D'_k = \frac{c'_k}{f_k} - T'_k$. Hence, the rest delay depends on the assigned workload to server a_k on the previous time slot as well as intermission time T'_k between two offloading decisions. The computation error probability indicates the probability of finishing computation of workload c_k within time t_2. The probability of this computation error for server a_k can be generally expressed by:

$$\varepsilon_{2,k} = \mathcal{P}(D_k \geq t_2), \tag{3.1.5}$$

where $\mathcal{P}(D_k \geq t_2$ is the probability that the computing time exceed t_2. Because of the First-In-First-Out (FIFO) policy, W_k can be assumed proportional to the current queue length Q_k according to littles Law [40]. $\varepsilon_{2,k}$ is a monotonously decreasing function regarding to t_2. We also assume that the computation frequency is fixed to f_k. Hence, we can describe the error probability as:

$$\varepsilon_{2,k} = \mathcal{P}(D_k \geq t_2) = \mathcal{P}(W_k \geq t_2 - \frac{c_k}{f_k}). \tag{3.1.6}$$

Following we use $\hat{t}_2 = \max\{t_2 - \frac{c_k}{f_k}, 0\}$ as the modified delay. Since the MEC should support high reliability applications, the computation error probability needs to be very low. Therefore, the CCDF of the queue delay satisfies $\bar{F}_{W_k}(\hat{t}_2) = \varepsilon_{2,k} = \mathcal{P}(W_k \geq \hat{t}_2) \ll 1$. Hence, the tail performance of the CCDF is very important when designing the MEC network. Using the extreme value theory, the tail performance of $\varepsilon_{2,k}$ can be characterized [41]. With D_k conditionally exceeding a high threshold d, we denote $X_k = \max\{\hat{t_2} - d, 0\}$ the exceedance of delay tolerance. If the threshold d approaches $F^{-1}_{W_k}(1)$, the CCDF of the exceedance X_k can be expressed as:

$$\begin{aligned} F_{X_k|D_k>d}(x_k) &= \mathcal{P}(D_k - d \leq x_k | D_k > d) \\ &\approx G(x;\sigma,\xi) \\ &= \left\{ \begin{array}{l} e^{-x/\sigma}, \text{ if } \xi = 0, \\ 1 - \left(1 + \frac{\xi x}{\sigma}\right)^{-\frac{1}{\xi}}, \text{ otherwise} \end{array} \right\}. \end{aligned} \tag{3.1.7}$$

with $G(x;\sigma,\xi)$ being the Generalized Pareto Distribution (GPD)) characterized by the scale parameter $\sigma > 0$ and the shape parameter ξ. ξ indicates the tail behavior of the CCDF. For this model we investigate cases with $\xi > -1/2$, which means that the CCDF has a finite upper endpoint. The frame duration is insufficient to completely avoid the occurrence of extreme events in an URLLC scenario. So there is always the possibility of queue length being too long for t_2. The computation error probability for a given time slot of in the ultra-reliable scenario with the threshold d is given by:

$$\varepsilon_{2,k} = (1 - F_{D_k}(d))(1 - G(\max\{t_2 - \frac{c_k}{f_k} - D'_k - d, 0\};\sigma,\xi)). \tag{3.1.8}$$

σ and ξ are influenced by the computing task arrival rate and the computation power of the server k. By empirical data these parameters can be obtained. More importantly, the validity of the expression does not depend on any specific task distribution.

3.1.3 End-to-End Error Probability

The communication error for data transmission and computation error for computing offloaded tasks influence the end-to-end error probability for each server a_k at each time slot. The overall service will be successful if both phases finished successfully. Obviously, an error occurs when a transmission to a server fails or the computation delay exceeds t_2. If one server is not selected it will not contribute to the overall error probability, as $\varepsilon_k = 0, \forall k \notin \mathcal{K}$. The end-to-end probability for each server a_k can be given as:

$$\begin{aligned} \varepsilon_k &= a_k(\varepsilon_{1,k} + (1 - \varepsilon_{1,k})\varepsilon_{2,k}) \\ &= a_k(\varepsilon_{1,k} + \varepsilon_{2,k} - \varepsilon_{1,k}\varepsilon_{2,k}). \end{aligned} \tag{3.1.9}$$

The single end-to-end error probability compose to the overall error probability ε_O over all servers as obtained by:

$$\varepsilon_O = 1 - \prod_k (1 - \varepsilon_k). \tag{3.1.10}$$

3.2 Optimal Design of Offloading Transmission

First of all, the optimal analytical solution for transmission based offloading is presented. We consider a scenario with perfect knowledge of the channel state as well as independent server computation performance between each time slots, obtained by $D'_k = 0$. The objective is to minimize the overall error probability ε_O. Under the sum of t_1 and t_2, as well as workload c_k and server selection a_k the optimal decision is made with regards to the end-to-end to probability. First of all, the problem will be formulated in Section 3.2.1, which then will be decomposed to subproblems. We will show the convexity of t_1 and t_2, as well as the optimality of $t_1 + t_2 = T_{\max}$ in Section 3.2.2. In Section 3.2.3 convexity of t and $\mathbf{C}$ in ε_O is shown. At last we reformulate the problem in Section 3.2.4

3.2.1 Problem Formulation

The whole problem is formulated with parameters t_1, t_2, $\mathbf{C}$ and $\mathbf{A}$ as:

$$\begin{aligned}
\underset{\mathbf{t_1}, \mathbf{t_2}, \mathbf{C}, \mathbf{A}}{\text{minimize}} \quad & \varepsilon_O && \text{(3.2.1a)}\\
\text{subject to} \quad & \mathbf{A} \in \{0,1\}^K, && \text{(3.2.1b)}\\
& t_1 + t_2 \le T_{\max}, && \text{(3.2.1c)}\\
& \varepsilon_k \le \varepsilon_{\max}, \qquad \forall k \in \mathcal{K}, && \text{(3.2.1d)}\\
& c_k \le a_k c_O, \qquad \forall k \in \mathcal{K}, && \text{(3.2.1e)}\\
& \textstyle\sum_{k=1}^{K} c_k = c_O, && \text{(3.2.1f)}\\
& \textstyle\sum_{k=1}^{K} a_k \ge 1 && \text{(3.2.1g)}
\end{aligned}$$

where the constraints making sure that all given conditions are met for the offloading transmission system. The sum of communication time and computation time shall not exceed the total time delay T. The computation size shall not exceed the total workload c_o as well as at least 1 server to be selected.

3.2.2 Decomposition to Subproblem

The main problem is decomposed to the following subproblem:

$$\begin{aligned}
\underset{\mathbf{t_1}, \mathbf{t_2}, \mathbf{C}}{\text{minimize}} \quad & \varepsilon_O && && (3.2.2\text{a})\\
\text{subject to} \quad & \varepsilon_k \leq \varepsilon_{\max}, && \forall k \in \hat{\mathcal{K}}, && (3.2.2\text{b})\\
& a_k = 1, && \forall k \in \hat{\mathcal{K}}, && (3.2.2\text{c})\\
& c_k \leq a_k c_O, && \forall k \in \mathcal{K}, && (3.2.2\text{d})\\
& \textstyle\sum_{k=1}^{K} c_k = c_O. && && (3.2.2\text{e})
\end{aligned}$$

With K available servers, the main problem can be decomposed into $2^K - 1$ subproblem as formulated before. The subproblem aims at minimizing the end-to-end error probability while choosing the optimal t_1 and t_2 as well as workload c. Further, the time distribution to communication and computation phase with a fixed workload c_k, is observed:

$$\begin{aligned}
\underset{\mathbf{t_1}, \mathbf{t_2}}{\text{minimize}} \quad & \varepsilon_O && && (3.2.3\text{a})\\
\text{subject to} \quad & \varepsilon_k \leq \varepsilon_{\max}, && \forall k \in \hat{\mathcal{K}}, && (3.2.3\text{b})\\
& a_k = 1, && \forall k \in \hat{\mathcal{K}}, && (3.2.3\text{c})\\
& c_k = c{\circ}_k, \forall k \in \mathcal{K}, && && (3.2.3\text{d})\\
& \textstyle\sum_{k=1}^{K} c_k = c_O. && && (3.2.3\text{e})
\end{aligned}$$

The following lemma investigates two characterization regarding the subproblem, where we will show the convexity of the error probability e_k.

Lemma 1. *The end-to-end error probability ε_k is convex in both t_1 and t_2.*

Proof. If servers are not selected with $k \ni \hat{\mathcal{K}}$, we have $e_k = 1$, which is convex for t_1 and t_2. For the selected servers, the second derivative regarding t_1 is as:

$$\begin{aligned}
\frac{\partial^2 \varepsilon_k}{\partial t_1^2} &= \frac{\partial^2 \varepsilon_{1,k}}{\partial t_1^2} + 0 - \frac{\partial^2 \varepsilon_{1,k}}{\partial t_1^2} \varepsilon_{2,k} \\
&= \frac{\partial^2 \varepsilon_{1,k}}{\partial n_1^2} \left(\frac{\partial n_1}{\partial t_1} \right)^2 (1 - \varepsilon_{2,k}) \qquad (3.2.4)\\
&= \frac{\partial^2 \varepsilon_{1,k}}{\partial n_1^2} \frac{(1 - \varepsilon_{2,k})}{T_s^2}.
\end{aligned}$$

It is shown that $\frac{\partial^2 \varepsilon_{1,k}}{\partial n_1^2} \geq 0$. Because the error probability of the computation phase is $\varepsilon_{2,k} \leq 1$, it means that $\frac{\partial^2 \varepsilon_k}{\partial t_1^2} \geq 0$ as well as that the error probability for the selected servers is convex for t_1. Hence, the second derivative of t_2 can be given based on the past solution as:

$$\begin{aligned}\frac{\partial^2 \varepsilon_k}{\partial t_2^2} &= \frac{\partial^2 \varepsilon_{2,k}}{\partial t_2^2} + 0 - \frac{\partial^2 \varepsilon_{2,k}}{\partial t_1^2}\varepsilon_{1,k} \\ &= \frac{\partial^2 \varepsilon_{2,k}}{\partial t_2^2}(1-\varepsilon_{1,k})\end{aligned} \tag{3.2.5}$$

Since we consider a ultra-reliable scenario with a maximum error probability of $\varepsilon_{\max} < 0.01$ as well as that each phase error probability is lower than the threshold, we assume that t_2 is sufficient to apply the EVT for the selected servers as given in previous section:

$$\begin{aligned}\frac{\partial^2 \varepsilon_{2,k}}{\partial t_2^2} &= F_{D_k}(d)\frac{\partial^2 G(t_2 - \frac{c_k}{f_k}d_k;\sigma,\xi)}{\partial t_2^2} \\ &= F_{D_k}(d)\frac{(1+\xi)}{\sigma^2} \\ &\qquad \cdot \left(1 - G(t_2 - \frac{c_k}{f_k} - d;\sigma,\xi)\right)^{-\frac{2+\xi}{\xi}} \\ &\geq 0.\end{aligned} \tag{3.2.6}$$

After showing the convexity of ε_1 and ε_2, we present the convexity of the total end-to-end probability ε_o. Furthermore, the second derivative of ε_O is expressed as:

$$\frac{\partial^2 \varepsilon_O}{\partial t_1^2} = -\sum_k \frac{\partial^2 v_k}{\partial t_1^2}\prod_{l\neq k} v_l + \sum_k \sum_{l\neq k} \frac{\partial v_k}{\partial t_1}\frac{\partial v_l}{\partial t_1}\prod_{p\neq k, p\neq l} v_p \tag{3.2.7}$$

where $v_k = 1 - \varepsilon_k$ denotes the reliability of the server k, which means that if a server is not selected it is always reliable. $\varepsilon_k \leq 1$ is a convex and monotonic function with respect to t_1. $v_k = 1 - \varepsilon_k \geq 0$ is a concave and monotonic function, because $\varepsilon_k \leq 1$ is a convex and monotonic function with respect to t_1. We have $\frac{\partial^2 v_k}{\partial t_1^2} \geq 0, \forall a_k = 1$ and $\frac{\partial^2 v_k}{\partial t_1^2} = 0, \forall a_k = 0$. Moreover, we have $sgn\left(\frac{\partial v_k}{\partial t_1}\right) = sgn\left(\frac{\partial v_l}{\partial t_1}\right), \forall a_k = a_l = 1$ and $\frac{\partial v_k}{\partial t_1} = 0, \forall a_k = 0$, where $sgn(\cdot)$ is the sign function. Hence, it holds $\frac{\partial^2 \varepsilon_O}{\partial t_1^2} \geq 0$. Analog to t_1, we can show that $\frac{\partial^2 \varepsilon_O}{\partial t_2^2} \geq 0$ since ε_k is also a convex and monotonic function with respect to t_2 according to (3.2.5). As a result, the end-to-end error probability ε_k is convex in both t_1 and t_2. □

Lemma 2. *The optimal solutions to Problem (3.2.2), given by t_1^* and t_2^*, satisfy $t_1^* + t_2^* = T_{\max}$.*

Proof. Further, we show that $t_1^* + t_2^* = T_{\max}$ satisfies the optimal solution to problem 12 given t_1 and t_2. The optimal solution is shown via a contradiction, which assumes that the optimal solution t_1' and t_2' satisfies the strict inequality of constraint (21d), as $T_{\max}-(t_1'+t_2') = \alpha > 0$. Therefore, $\varepsilon_O'(t_1', t_2')$ has to be always the global minimum. Contradictory $(t_1'' = t_1' + \alpha, t_2'' = t_2') \in \{t_1, t_2 | t_1 + t_2 \leq T_{\max}$ is a feasibly solution, which also results in a lower error probability ε_O, because ε_O is an decreasing function in regards to the blocklength n as well as to t_1. Hence, we can conclude that solution (t_1'', t_2'') results a lower error probability comparing to (t_1', t_2'), i.e., $\varepsilon_O''(t_1'', t_2'') < \varepsilon_O'(t_1', t_2')$. Therefore, the assumption of the optimal solution (t_1', t_2') is violated. $t_1^* + t_2^* = T_{\max}$ satisfy the optimal solution to problem (12). □

3.2.3 Optimal Time Allocation and Workload

Following, the subproblem minimizing ε_O under the consideration of t_1 and c_k is analyzed. The subproblem is expressed as:

$$\begin{aligned} \underset{\mathbf{t_1}, \mathbf{C}}{\text{minimize}} \quad & \varepsilon_O && (3.2.8a) \\ \text{subject to} \quad & t_1 + t_2 = T_{\max}, && (3.2.8b) \\ & \varepsilon_k \leq \varepsilon_{\max}, \quad \forall k \in \mathcal{K}, && (3.2.8c) \\ & c_k \leq a_k c_O, \quad \forall k \in \mathcal{K}, && (3.2.8d) \\ & \textstyle\sum_{k=1}^{K} c_k = c_O && (3.2.8e) \end{aligned}$$

Further, we show that ε_O is jointly convex in t_1 and $\mathbf{C}$, if it holds $\varepsilon_k \leq \varepsilon_{\max} \ll 1$, $\forall k \in \mathcal{K}$.

Lemma 3. *The end-to-end error probability ε_O is jointly convex in t_1 and $\mathbf{C}$, if it holds $\varepsilon_k \leq \varepsilon_{\max} \ll 1$, $\forall k \in \mathcal{K}$.*

Proof. If the error probability of each link k is sufficiently small, the end-to-end error probability ε_O can be approximated as:

$$\varepsilon_O = 1 - \prod(1 - \varepsilon_k) \approx \sum_k \varepsilon_k. \tag{3.2.9}$$

It also implies that both ε_1 and ε_O need to be lower than $\varepsilon_{\max}$. The error probability in link k can also be approximated as $\varepsilon_k \approx \varepsilon_{1,k} + \varepsilon_{2,k}$. To determine the convexity of ε_O, we use the Hessian matrix of ε_O with regards to t_1 and $\mathbf{C}$, which can be written as:

$$H = \begin{bmatrix} \frac{\partial^2 \varepsilon_{1,k}}{\partial t_1^2} + \frac{\partial^2 \varepsilon_{2,k}}{\partial t_1^2} & \frac{\partial^2 \varepsilon_{2,k}}{\partial t_1 \partial c_k} \\ \frac{\partial^2 \varepsilon_{2,k}}{\partial t_1 \partial c_k} & \frac{\partial^2 \varepsilon_{2,k}}{\partial c_k^2} \end{bmatrix} \tag{3.2.10}$$

After manipulations according to (3.2.4), (3.2.5) and (3.1.6), the determinate of H is given by

$$\det H = \frac{\partial^2 \varepsilon_{1,k}}{\partial t_1^2} \frac{\partial^2 \varepsilon_{2,k}}{\partial c_k^2} \tag{3.2.11}$$

where $\frac{\partial^2 \varepsilon_{1,k}}{\partial n_1^2} \frac{1}{T_S^2} \geq 0$ and $\frac{\partial^2 \varepsilon_{2,k}}{\partial c_k^2} = \frac{\partial^2 \varepsilon_{2,k}}{\partial t_2^2} \frac{1}{f_k^2} \geq 0$. Hence, it holds $\det H \geq 0$. As a results, ε_k is jointly convex in t_1 and C. As shown, the error probability of each link k is joint convex. Further, the overall end-to-end error probability is the sum of error probabilities in every link. Hence, the overall end-to-end error probability is also joint convex in t_1 and C. $\square$

3.2.4 Reformulation of the problem

According to the above lemmas characterizing the subproblem Problem-c1, the original problem in problem-mix can be reformulated as:

$$\begin{aligned}
&\underset{\mathbf{t_1}, \mathbf{C}, \mathbf{A}}{\text{minimize}} && \varepsilon_O = 1 - \prod_k (1 - \varepsilon_k) && && \text{(3.2.12a)}\\
&\text{subject to} && \mathbf{A} \in \{0,1\}^K, && && \text{(3.2.12b)}\\
& && t_1 + t_2 = T_{\max}, && && \text{(3.2.12c)}\\
& && \varepsilon_k \leq \varepsilon_{\max}, && \forall k \in \mathcal{K}, && \text{(3.2.12d)}\\
& && c_k \leq a_k c_O, && \forall k \in \mathcal{K}, && \text{(3.2.12e)}\\
& && \textstyle\sum_{k=1}^{K} a_k \geq 1, && && \text{(3.2.12f)}\\
& && \varepsilon_k \leq a_k, && \forall k \in \mathcal{K}, && \text{(3.2.12g)}\\
& && a_k(\varepsilon_{1,k} + \varepsilon_{2,k} - \varepsilon_{1,k}\varepsilon_{2,k}) \leq \varepsilon_k, \forall k \in \mathcal{K} && && \text{(3.2.12h)}
\end{aligned}$$

where (3.2.12g) and (3.2.12h) are the constraints for the linearization of objective function, which helps us to avoid the multiplication of variable a_k and t_1. Because the objective function and constraints of problem (3.2.12) are either affine or convex the reformulated problem in (3.2.12) becomes a Mixed-Integer Convex Problem (MICP), which can be solved efficiently via the recently developed algorithm in [42].

3.2.5 Numerical Simulation of Problem

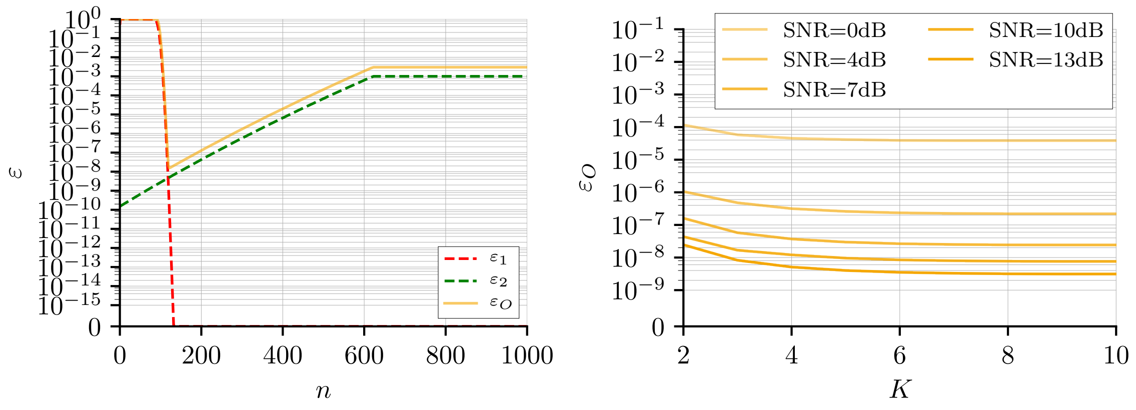
Furthermore, we present numerical results of implementing the model. The used parameters for communication and computation phase are shown in Table 3.1. We consider a static channel with $\rho = 1$. Every channel has the same channel gain $h_k = 1$ and SNR $\gamma = 10dB$. Because all channels have the same state we can assume equally assigned workload c_k over all selected servers a_k. The setup contains three servers k handling the offload of one user.

The results show that there is a optimal solution for blocklength n and server selection $\mathbf{A}$. As shown in Fig. 3.2a, with an increasing blocklength the communication

Model Parameters			
SNR	10 dB	Computation Power f	3 GHz
Symbol Time	0.025×10^{-3}	Workload c_O	24 Mcycles
Packet Size τ_P	320 bits	Shape Parameter ξ	-0.0214
Delay Tolerance T	25 ms	Scale Parameter σ	3.4955×10^6
Intermission T'	4×10^6	Threshold d	2.0384×10^7
ε_{max}	0.011	Task Poisson Arriving Rate λ	3 Mcycles/s

Table 3.1: Configuration of Environment

phase error probability $\varepsilon_{1,k}$ decreases. Accordingly, the computation phase error probability decreases as well with $t_2 = T - t_1$. The optimal point of the end-to-end error probability is close to $\varepsilon_{1,k} = \varepsilon_{2,k}$. Since the maximal possible tolerated error probability is to be considered at 0.001, the offloading procedure can be seen as erroneous. Fig. 3.2b shows for an increasing SNR ε_O decreases for the same number of servers K. It is also shown that there is a certain number of K that ε_O converges for equal channel gain and SNR over all channels.



(a) Communication, Computation and Total Error probability

(b) Overall erorr probablity ε_O versus number of servers with different SNR

Figure 3.2: Error probability for communication, computation and overall (a), as well as overall error probability over K for different SNR

4 Deep Q-Learning Based Offloading

Because for some RL agents an increasing action becomes computationally too complex. In particular for Deep Q-Learning, we first consider a simpler model where the workload is equally assigned to each selected server. The model introduced in [11], is used for the presented Deep Q-Learning method. The model is formulated and its optimality is similarly introduced as the previously presented model. The constraint of $c_k < \frac{c_O}{\sum_k a_k}$ replaces the previous constraint of $c_k a <= a_k c_O,\ \forall k \in \mathcal{K}$. The changed problem formulation regarding only server selection and blocklength is written as:

$$
\begin{aligned}
\underset{\mathbf{t_1}, \mathbf{t_2}, \mathbf{A}}{\text{minimize}} \quad & \log(\varepsilon_O) && (4.0.1\text{a}) \\
\text{subject to} \quad & \mathbf{A} \in \{0,1\}^K, && (4.0.1\text{b}) \\
& t_1 + t_2 \leq T_{\max}, && (4.0.1\text{c}) \\
& \varepsilon_k \leq \varepsilon_{\max}, \forall k \in \mathcal{K}, && (4.0.1\text{d}) \\
& \textstyle\sum_{k=1}^{K} a_k \geq 1 && (4.0.1\text{e})
\end{aligned}
$$

where t_1 and t_2 are the time of communication and computation phase respectively as well as $\mathbf{A}$ being the server selection vector. The added constraint ensures an equally assigned workload and that the sum of the assigned workload is exactly c_O. It is shown that the problem has an optimum, which can be solved using exhaustive search and equations (3.1.8) and (3.1.2).

The formulated problem is a Mixed-Integer Problem (MIP), with n being an integer and $\mathbf{A}$ being a binary variable. In particular, for a larger number of servers and states, the problem can become too computationally complex to solve efficiently. In realistic cases the agent may not have perfect knowledge about the condition of the channel or even about the server state. The UE still has to decide on server selection k as well as resource allocation in form of workload c_k. Currently, there does not exist a computationally sufficient analytically solution for offloading based on outdated information, while satisfying the large amount of states, in form of channel condition, i.e.: SNR γ_k or channel gain h_k, or server state in form of computation delay D_k. Since the problem can be formulated as a Markov-Decision-Process, we propose a deep reinforcement learning approach to solve the presented problem. In particular, we use DQN for selecting servers based on the channel information, using SNR γ. As mentioned prior, DQN works well for a limited discrete action space. Whereas for a larger action space, or even for continuous actions can lead to a long training time and high computation effort to learn the optimal behavior for all state transitions. Hence, the agent decides on $\mathbf{A}$ and based on $\mathbf{A}$ we use an exhaustive search to find the optimal blocklength n.

First of all, the reinforcement learning setup with its states, actions, rewards and transitions, the implementation of the Q-network with its hyperparameters as well as the parameterization of the model is presented in Section 4.1. In the following, we consider a static channel with perfect CSI for the agent in Section 4.2. We show the performance results regarding ε_O and compare them to the analytical solution. Further, we use a Rayleigh fading channel, where the channel gain $h_k^{(m)}$ correlates to $h^{(m+1)}$. Based on perfect CSI, we present the server selection decision performance of DQN and the optimal solution also in Section 4.2. Furthermore, we analyze in Section 4.3 the parameterization of the DQN agent as well as the offloading decision performance based on outdated CSI, while using perfect CSI for computing the blocklength n based on the selected $\mathbf{A}$. The section also provides the learning results using outdated CSI for both the agent's server selection decision and for searching blocklength n.

4.1 RL Setup and Network Configuration

The reinforcement learning environment E inherits the used offloading model with the overall error probability ε_O. The UE contains the agent to decide on the offloading decision. The agent receives state $s$and makes the server selection $\mathbf{A}$ with $[a_1, ..., a_K]$. State s formulates to $s^{(m)} = [X^{(m_0)}, X^{(m_0-1)}, X^{(m_0-W_{\text{ht}})}]$, where m_0 is the latest received time slot and W_{ht} indicates the number of historical time slot information used for state creation. The state s for every scenario will be introduced before the simulation results. After receiving action b the environment computes the reward r. Since the goal is to minimize the overall error probability ε_O and the Q-Learning rule aims at maximizing the reward r, we introduce the reward r for time slot m as:

$$r^{(m)} \quad = \quad -\log(\varepsilon_O^{(m)}) \tag{4.1.1}$$

while we assume $r^{(m)} = 1,\ \forall \varepsilon_O \leq 10^{-100}$ for numerical calculations. Accordingly, the agent is rewarded from the environment E with a higher value for a lower overall error probability ε_O. The environment E consists of communication phase with error probability ε_1 and computation phase with error probability ε_2. Figure Table 3.1 shows the parameters for each phase used in the simulation.

DQN uses two deep neural networks, one for the Q-network and its target network. Both networks have the same structure. After $\tau = 10$ training steps, the Q-network copies its weights to the target network. The size of the input layer equals the size of the state vector. For the hidden layer we use the ReLU as a activation function. The output layer has the size of the length of the action space $\mathcal{B}$ or the length of $\mathbf{A}$, which corresponds to $2^K - 1$ and uses the linear activation function. The output, presenting the expected discounted reward, of each layer is computed using a linear activation function. The action with the highest Q-value is then chosen. As shown in Algorithm 2, in the training phase each transition tuple $(s,\, b,\, s^{(m+1)},\, r)$ is stored in a memory of size $U = 500$. The samples of the last 500 time slots are stored and used for random batch selection. The batch of size $U_b = 16$ with random samples from the memory is used for training the Q-network. While training, DQN choses with exploration factor ϵ a random action. Otherwise the agent decides on the bases of the Q-values provided by the Q-network. The exploration factor decreases with $\epsilon^{(m+1)} = \epsilon^{(m)} \times \epsilon_{dec}$, where ϵ_{dec} is the exploration decay.

The simulation is separated into episodes. In each episode the network is trained $M_{\text{train}} = 50$ times. Afterwards the network is tested $M_{\text{test}} = 1000$ times. Note that for testing the exploration rate ϵ is equal to zero and the network solely relies on its learned Q-values. After each testing phase the error probability for each cycle is averaged over M_{test}. Because the values of ε_O can lie in different magnitudes of 10, we chose to take the logarithmic average, formulated as: $\varepsilon_{O,log} = \frac{\sum_0^{M_{\text{test}}} -\log(\varepsilon_O)}{\text{test}}$. The average loss L is presented similarly as $L_{O,log} = \frac{\sum_0^{M_{\text{train}}} -\log(L_O)}{M_{\text{train}}}$.

Algorithm 2 Deep Q-Learning algorithm

```
Initialize the replay memory $D_{\text{mem}}$ to capacity U
Initialize Q network $Q(s, b|\theta^Q)$ and the target network $Q^{'}(s, b|\theta^{Q^{'}})$
for m=1,2,3,... do
    Observe current state $s^{(m)}$
    Select random action $b^{(m)}$ with probability $\epsilon$
    Otherwise select $b^{(m)} = \arg\max_b Q(s^{(m)}, b^{(m)}; \theta^Q)$
    Execute $b^{(m)}$ and observe $r^{(m)}$ and $s^{(m+1)}$
    Store transition $(s^{(m)}, b^{(m)}, r^{(m)}, s^{(m+1)})$ in $D_{mem}$
    Sample a random minibatch of size $U_{\text{b}}$ transitions $(s^{(i)}, b^{(i)}, r^{(i)}, s^{(i+1)})$
    Set $y^{(i)} = r^{(i)} + \gamma_{\text{df}} \max Q(s^{(i+1)}, (b^{(i+1)}|\theta^Q)$
    Perform update using Adam optimizer on Q-network by minimizing the loss:

    $L_i(\theta_i) = \mathbb{E}_(s, b, r, s') \sim U(D)[(r + \gamma_{\text{df}} \max Q(s', b'; \theta_i^-) - Q(s, b; \theta_i))^2]$   (4.1.2)
```

4.2 Perfect CSI

First, we consider a static channel in Section 4.2.1 and a Rayleigh fading channel in Section 4.2.2 while having perfect knowledge about the channel condition. We consider an input $x^{(m)}$ with $W_{ht} = 0$ and $m_0 = m$ composed as:

- $\gamma^{(m)} = [\gamma_1^{(m)}, \gamma_2^{(m)}, ..., \gamma_K^{(m)}]$: vector of length K containing SNR of every link

We design two neural networks with a learning rate $\alpha = 0.0001$, exploration decay $\epsilon = 0.9$ and discount factor $\gamma_{df} = 0.0$. We also use only one hidden layer for the network. Since there is no impact of the decision from one time slot to the state of the next time slot, we use the current input $x^{(m)}$ as the receiving state, which formulates to $s^{(m)} = x^{(m)}$ with $W_{ht} = 1$. So, the environment is fully observed. We also consider the queuing delay $D' = 0$. Hence, there is no dependence of computation efforts between two time slots.

4.2.1 Static Channel

For the static channel, we consider equal $\gamma_k = 10dB$ over all channels as well as $\rho = 1$. The environment parameters are shown in Table 3.1. The agent learns for $\mathcal{H} = 200$ episodes. Because of the static channel, the agent receives the same state $s$at every time slot.

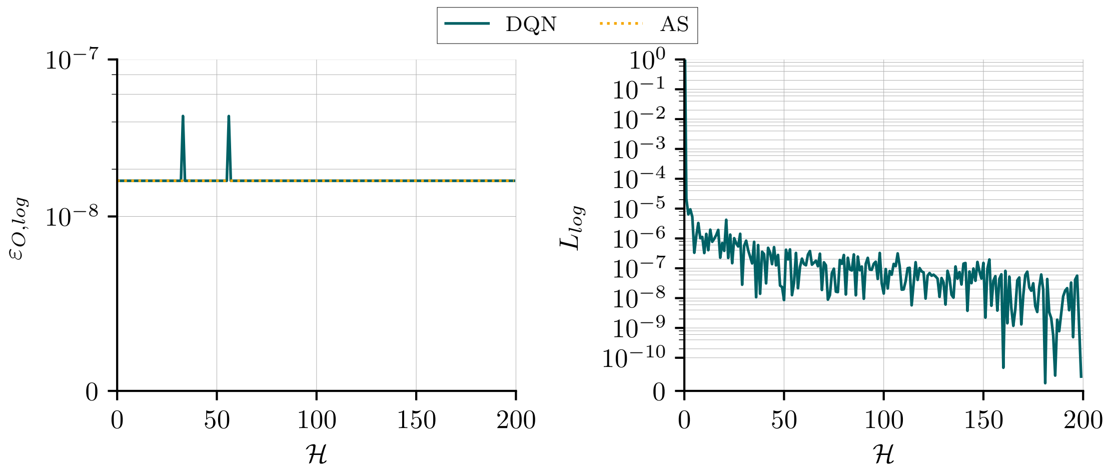


Figure 4.1: Error and loss for training with $K = 3$ in a static channel

First of all, we analyze the results for setup of $K = 3$. The results show in Fig. 4.1 that the DQN quickly converges, which is observed from the equal error probability over almost all episodes. Because of the low number of states, number of servers and hence, low number actions $2^3 - 1$ relative to the number of training cycles $M_{\text{train}} = 50$, we do not see a decrease in error probability. The learning of the network can be recognized from the decrease of loss over the episodes. Although the ε_O does not decrease anymore, the Q-values are still being adjusted. The loss's decrease shows that the Q-network learned the optimal policy and converges.

When consulting Fig. 4.2, $\varepsilon_{O,log}$ of the DQN method and the optimal solution is equal. It is also shown that DQN for two to 7 servers performs equal to the optimal solution. We also differentiate between two curves for the training process of the Q-network: the 'DQN-Last' as the performance of the network at the last episode and 'DQN-Best' as the best performance of the network over all episodes. Since we set the training process to a fixed number of episodes and do not stop the process when reaching a specific condition like falling below a gradient threshold or error probability threshold, we use the 'DQN-Best' to find the capabilities of the specific setup. For larger K, more training cycles are needed to reduce the loss in a way that the Q-network saturates at the optimal solution. Another factor could be the number of hidden layers compared to the size of the action space A. As shown in Fig. 3.2b, ε_O decreases for an increasing number of servers.

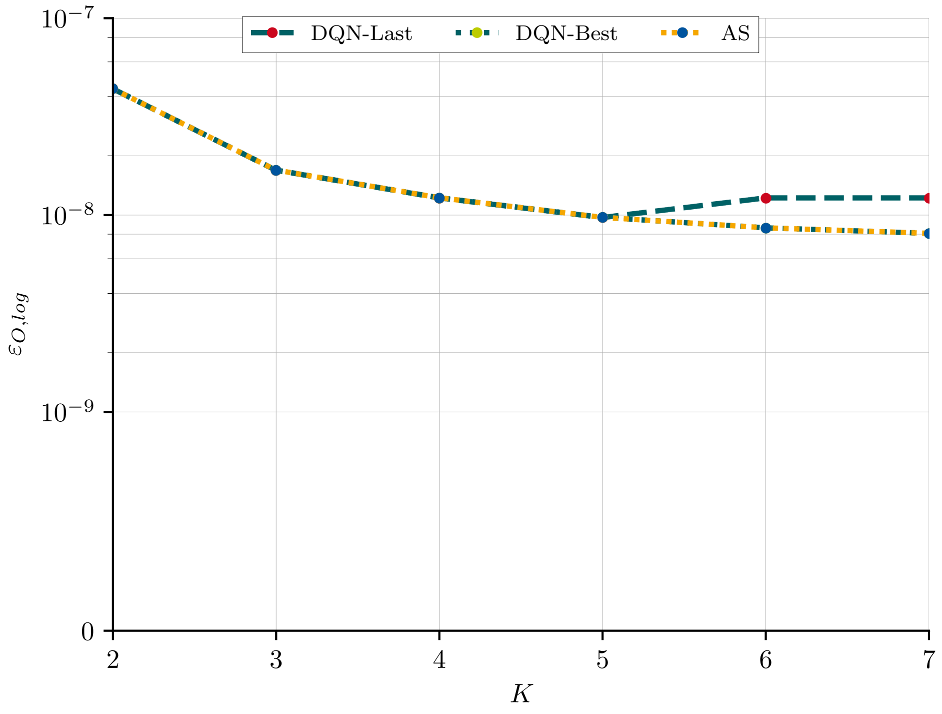


Figure 4.2: Error for different number of servers K in a static channel

4.2.2 Fading Channel

For the fading channel, we consider a correlation factor of $\rho = 0.9$. We use the same environment parameters as in the previous section, shown in Table 3.1. The agent also learns for $\mathcal{H} = 200$ episodes. We assume no waiting delay for computation efforts, hence $D' = 0$.

In Fig. 4.3, we can observe a steady decrease of error probability $\varepsilon_{O,log}$ and loss L_{log}. The curve saturates after approximately 80 episodes. When comparing the learning process to the static channel scenario, it can be seen that the agent needs more time to reach convergence. Compared to the static channel, which has a state space size of 1, the fading channel state space is $\mathbb{R}^K$. Because of the larger state space the Q-network needs longer to converge and near the optimal solution at just under 4×10^{-7}. Accordingly, the loss needs more training time to minimize.

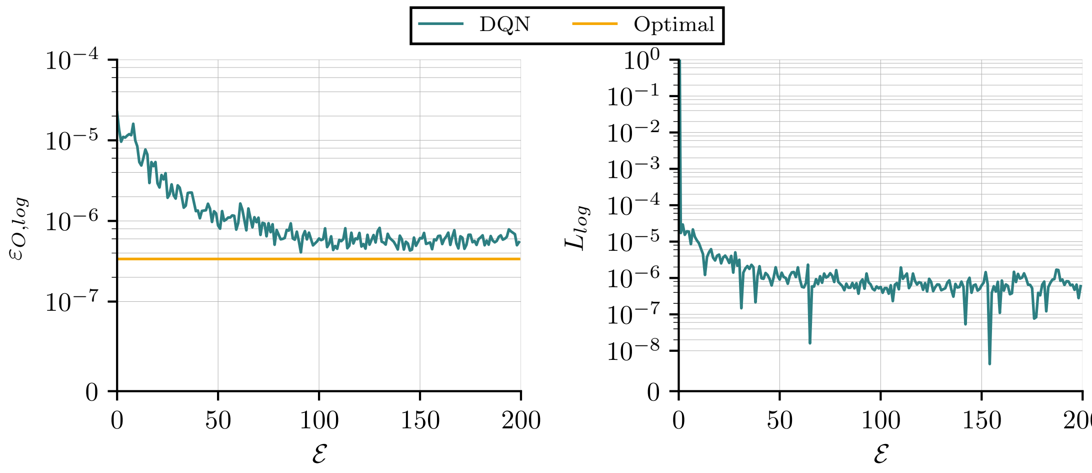


Figure 4.3: Error and loss for training with $K = 3$ in a fading channel based on perfect CSI

When increasing the number of servers in a fading channel, we can observe in Fig. 4.4 that the error probability should decrease as in the optimal solution. The optimal solution also decreases with an increase over K. Regarding DQN, at first $\varepsilon_{O,log}$ decreases as well. For $K \geq 5$ the error probability increases. Except for the number of neurons per hidden layer, the parameterization, number of hidden layers and training time of the DQN stays the same. The performance decreases with an increase of server K. The increase leads to a larger state space $\mathcal{S}$ and action space $\mathcal{B}$. For a larger action space more actions needs to be explored, which could be fastened using a higher exploration decay ϵ_{dec}. Also more hidden layers could lead to a better extraction of information for a larger input. At $K = 2$ shows also a better performance for the DQN method compared to the optimal solution, because of the runtime of the testing phase. The testing phase is set to 1000 cycles to speed up the training process and to still receive a reliable performance and convergence measure.

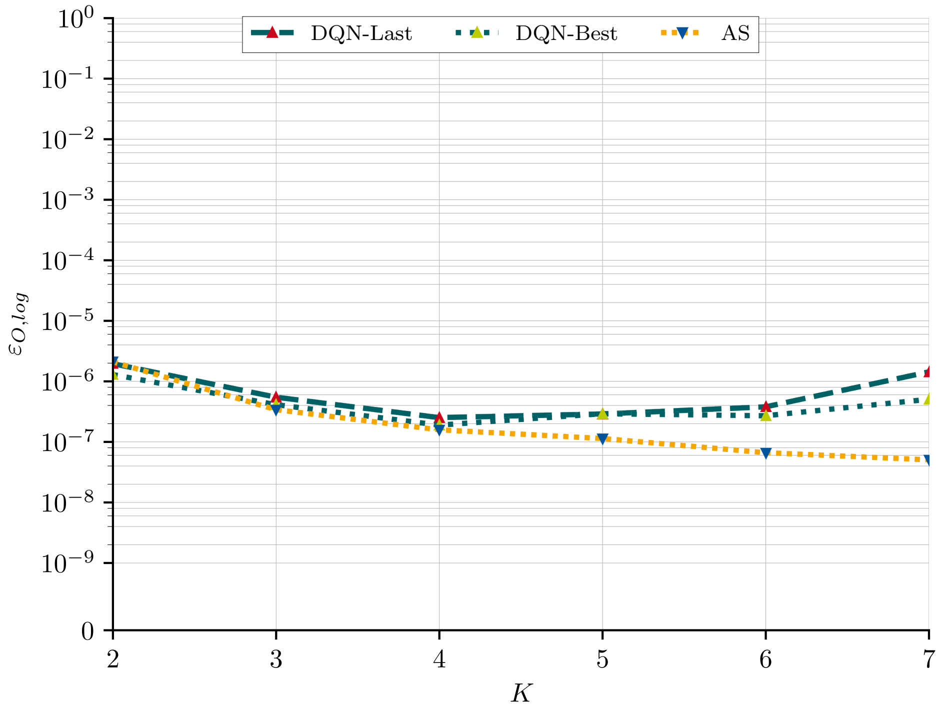


Figure 4.4: Error for different number of servers K in a fading channel

4.3 Outdated CSI

DQN can successfully learn the the optimal sever selection $\mathbf{A}$ in a static and fading channel based on perfect channel state knowledge. Because in application scenarios the channel is not only changing but mostly only imperfect information is available, we now consider a Rayleigh fading channel while having outdated knowledge about the channel condition. We also assume that there is a queuing delay, which is influenced by the previous workload at server k as well as by the intermission time T'_k. The rest queue from the previous time slot is described as $D_k^{(m-1)} = \frac{c_k^{(m-1)}}{f_k} - T'_k$. Consequently, the computing time at time slot m is $D_k^{(m)} = \frac{c_k^{(m)}}{f_k} + D^{(m-1)} + Q_k$. Therefore the previous time slot impacts the computation performance of the following time so. We consider an input $x^{(m)}$ with and $m_0 = m - 1$ composed as:

- $\gamma^{(\mathbf{m})} = [\gamma_1^{(m-1)}, \gamma_2^{(m-1)}, ..., \gamma_K^{(m-1)}]$: vector of length K containing SNR of every link
- $\mathbf{c}^{(\mathbf{m})} = [c_1^{(m-1)}, c_2^{(m-1)}, ..., c_K^{(m-1)}]$: vector of length K containing size of previous task loads

The action $a^{(m)}$ impacts state $s^{(m+1)}$, which introduces a correlation from actions $\mathcal{B}$ on to $\mathcal{S}$. Because of the impact from each time slot to the other we consider $s^{(m)} = [x^{(m)}, x^{(m-1)}, ..., x^{(m-W_{\text{ht}})}]$ with $W_{\text{ht}} = 3$.

In order to reach the best performance regarding ε_O as well as convergence speed, we find the best possible combination of the selected hyperparameters. In Section 4.3.1, we analyze the impact of different values for the hyperparameters. Based on the best combination of hyperparamters, we investigate the performance of the agent for different numbers of servers and channel gain correlation factor ρ in Section 4.3.2. In the last Section 4.3.3 we observe the results based only on imperfect channel state knowledge.

4.3.1 Hyperparameters

After some trial runs, we have selected different values for parameters α, γ_{df}, N_{hl} and ϵ_{dec}. We analyze the case of $K = 3$ servers with an average SNR. The complete environment parameters for communication and compution phase are shown in Table 3.1.

Learning Rate

The learning rate α is set to 0.1, 0.01, 0.001, 0.0001 and 0.00001. We set the other hyperparameters to $\gamma_{\text{df}} = 0.0$, $\epsilon_{\text{dec}} = 0.9$ and $N_{\text{hl}} = 1$. As shown in Fig. 4.5, for learning rates of $\alpha = 0.1$ and $\alpha = 0.01$ $\varepsilon_{O,log}$ alternates very strongly. While for $\alpha = 0.1$, the loss is much higher and does not decrease. For $\alpha = 0.001$ and $\alpha = 0.0001$, $\varepsilon_{O,log}$ decreases while $\alpha = 0.0001$ decreases faster and converges to a lower error value. The strong oscillation of higher learning rates comes from over adjusting to the loss L_{log}. The network is never able to adjust its weights so that it can minimize its generated loss during training. Even when lowering the learning rate as for $\alpha = 0.001$, the weight adjustment is too strong. For $\alpha = 0.00001$, the learning is too slow. The network does not adjust its weights as strongly, hence it takes longer to train to the same performance.

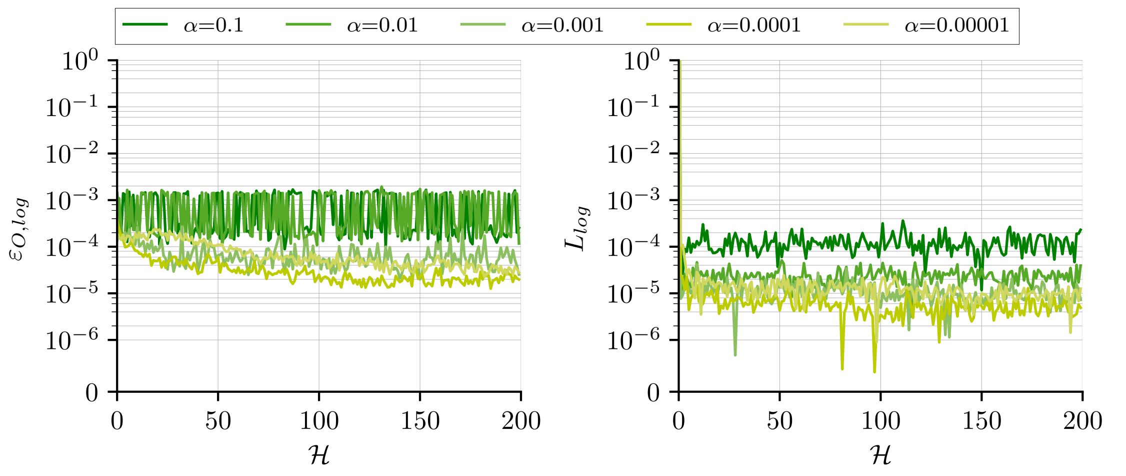


Figure 4.5: Error and loss for training with $K = 3$ in a fading channel based on outdated CSI for different α

Discount Factor

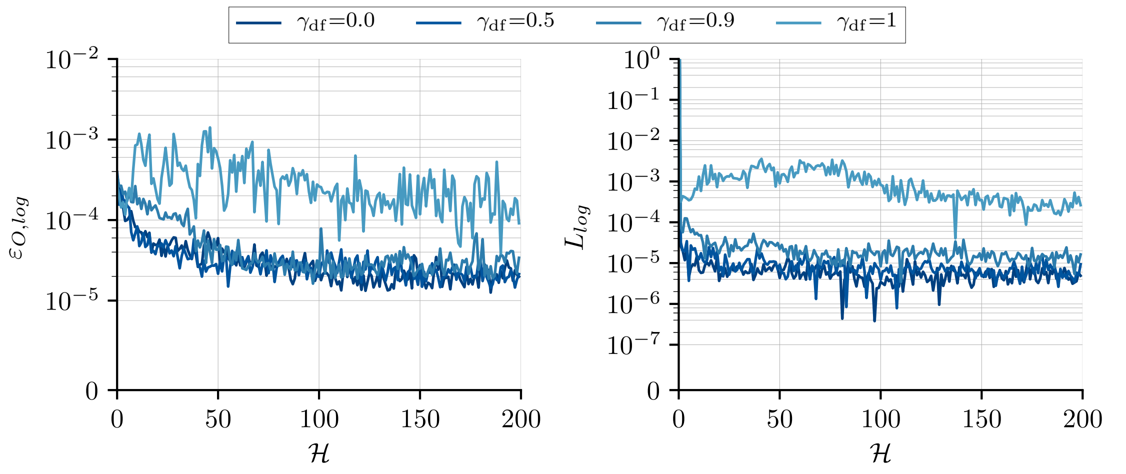


Figure 4.6: Error and loss for training with $K = 3$ in a fading channel based on outdated CSI for different γ_{df}

The discount γ_{df} fixed to 0.0, 0.5, 0.9 and 1. We set the other hyperparameters to $\alpha = 0.0001$, $\epsilon_{\mathrm{dec}} = 0.9$ and $N_{\mathrm{hl}} = 1$. As shown in Fig. 4.6, discount factors of $\gamma_{\mathrm{df}} = 0.0$, $\gamma_{\mathrm{df}} = 0.5$ and $\gamma_{\mathrm{df}} = 0.9$ converge much faster and to a lower error probability. $\gamma_{\mathrm{df}} = 1$ does not converge, which is shown by the loss curve. The loss curve does also show the best performance for $\gamma_{\mathrm{df}} = 0.0$, which converges to the lowest error probability. The discount factor was introduced for the reason of securing convergence. The factor limits the impact of the expected future reward represented by the Q-value in the Bellman equation.

Network Size

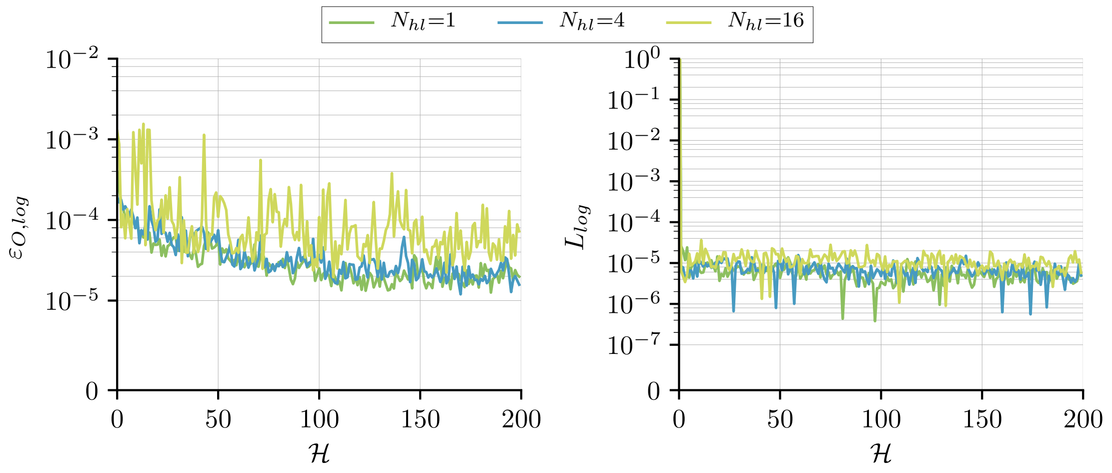


Figure 4.7: Error and loss for training with $K = 3$ in a fading channel based on outdated CSI for different number of hidden layers N_{hl}

For the network size we investigate three different scenarios. The Q-network and target network are designed with $N_{\mathrm{hl}} = 1$, $N_{\mathrm{hl}} = 4$ and $N_{\mathrm{hl}} = 16$ hidden layers. The size of the hidden layers is always the same and is equal to the size of the state, which for $K = 3$ and $W_{\mathrm{ht}} = 3$ corresponds to 18 neurons. As shown in Fig. 4.7, the Q-network with $N_{\mathrm{hl}} = 1$ hidden layers outperforms the other networks with more hidden layers. In regards to convergence speed, the smaller network decreases faster. All networks converge to roughly the same $\varepsilon_{O,log}$. It shows that a smaller network decreases its loss faster. It is also important to note that one hidden layer is sufficient to learn the best behavior in this scenario of given parameters.

Exploration Factor

The exploration factor states the probability of choosing a random action or using the learned Q-network for making an action. We investigate the use of different exploration decays $\epsilon_{\mathrm{dec}} = 0.5$, $\epsilon_{\mathrm{dec}} = 0.9$, $\epsilon_{\mathrm{dec}} = 0.99$ and $\epsilon_{\mathrm{dec}} = 0.999$. ϵ_{dec} determines the discussed trade-off between exploration and exploitation. As seen in Fig. 4.8, the Q-network learns faster when using a lower exploration. For $\epsilon_{\mathrm{dec}} = 0.999$ the network relies too long on random actions to explore a relatively small action space. $\epsilon_{\mathrm{dec}} = 0.5$ is quick to rely on its Q-network, where it does not decrease as fast its error probability, because it does not explore the whole action space $\mathcal{A}$ as fast.

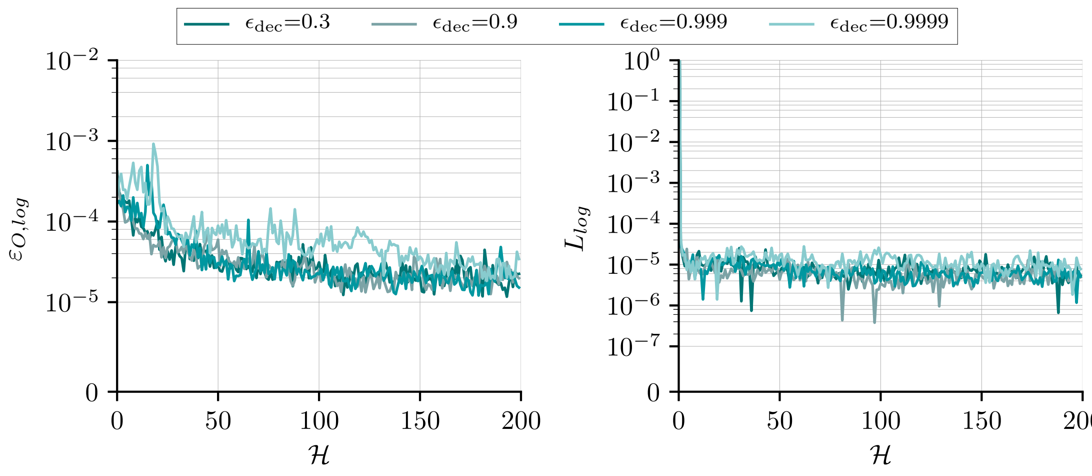


Figure 4.8: Error and loss for training with $K = 3$ in a fading channel based on outdated CSI for different exploration decay ϵ_{dec}

4.3.2 Performance of Offloading Outdated CSI

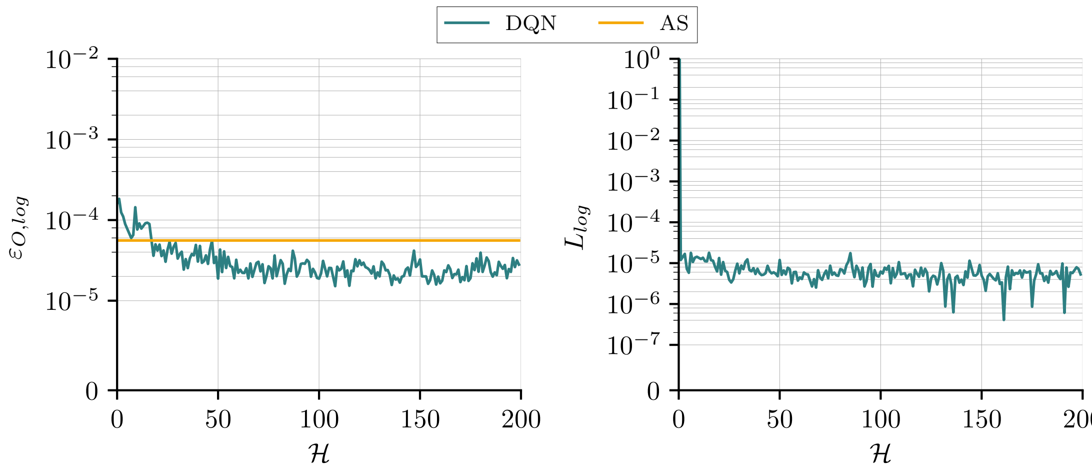


Figure 4.9: Error and loss for training with $K = 3$ in a fading channel based on outdated CSI for $K = 3$ servers

For the performance simulation, we use a setup with $K =$ servers and an average SNR of 10 dB in a Rayleigh-Fading channel with channel gain correlation of $\rho = 0.9$. The complete simulation parameters are shown in Table 3.1. For the Q-network we use the hyperparameter combination of learning rate $\alpha = 0.0001$, discount factor $\gamma_{\text{df}} = 0.0$, number of hidden layer $N_{\text{hl}} = 1$ and exploration factor $\epsilon_{\text{dec}} = 0.9$. Further, we analyze the convergence of the setup. In the following, we look at the performance for the same network setup for channel correlation $\rho \in [0, 1)$.

As seen in Fig. 4.9, $\varepsilon_{O,log}$ decreases with more training of the Q-network. After roughly 15 episodes it outperforms the analytical solution. After around 100 episodes the learning converges at around 2×10^{-5}. Note that the difference between the analytical solution and the learning is just determined by the server selection decision. There is also no channel initiliazation and random pseudo sequence used between the two methods, which could lead the differences. It is shown in Fig. 4.10

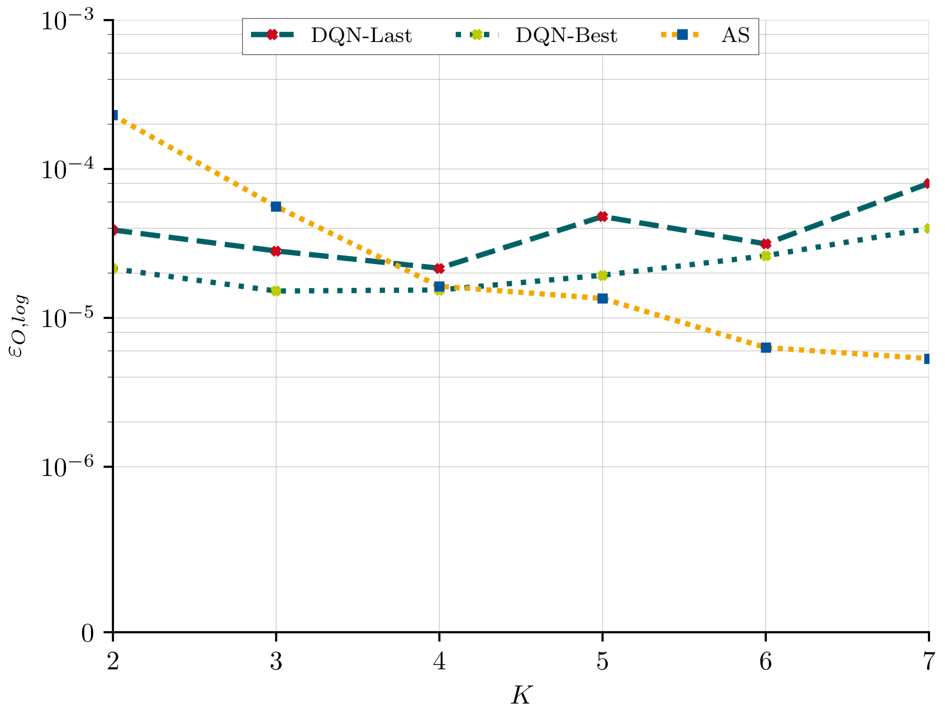


Figure 4.10: Error for different number of servers K in a fading channel based on outdated CSI

that $\varepsilon_{O,log}$ decreases with the number of K for the setup with $K = 2$, $K = 3$ and $K = 4$. Afterwards, $\varepsilon_{O,log}$ increases with K. Although, the layer size increases with the state size, the network cannot find the best server selection anymore. The other parameters as the number of hidden layer, length of training and exploration have to be changed accordingly as well in order to achieve a lower error probability for higher K. Hence, the analytical solution provides a better solution for scenarios with $K > 4$.

With an increase of the correlation factor ρ, we observe an increase of error probability for both the analytical solution as well as the DQN. Because the agent decides on server selection based on outdated CSI and the blocklength is calculated based on the current CSI, the performance gets worse for a high channel correlation. For low ρ the channel is sufficiently random that the overall error probability improves. The increase of $\varepsilon_{O,log}$ for the analytical solution could be due to the random channel initialization. With a fixed channel initialization and channel sequence, a decreasing curve over ρ is expected for the analytical solution. Otherwise an increase of the runtime M_{test} would be necessary.

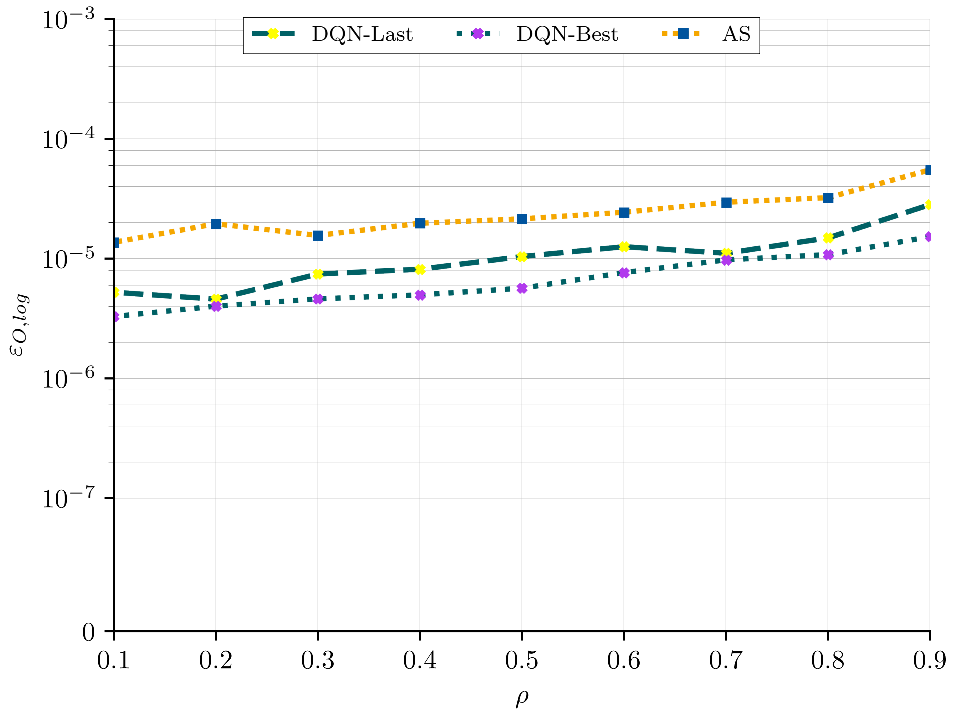


Figure 4.11: Error for different channel correlation factors ρ for a fading channel based on outdated CSI

4.3.3 Server Selection and Blocklength Based on Outdated CSI

As mentioned, in realistic cases there is no complete knowledge about channel condition or server state at the disposal, be it for deciding on servers or computing the blocklength. Therefore, we observe the case where only outdated CSI is provided to the agent. We now consider the case where not only the action selection is based on outdated CSI, but also the blocklength n. After the agent selects the server combination $\mathbf{A}$, the optimal blocklength for the previous time slot is calculated. We consider the same parameterization of environment and Q-network for outdated CSI as in the previous sections. The setup has 3 servers and an average SNR of 10dB.

As shown in Fig. 4.12, the error probability ε_O stays around 9×10^{-3} for all episodes. The loss decreases slightly, but oscillates a lot. The lack of loss reduction and improvement of error probability is due to high impact of blocklength n on the performance as well as training. The numerical results of the model introduced in Chapter 3 show only a small margin of blocklength n that generates a low error probability. Because of the dependency on the blocklength, the Q-network is not able to learn efficiently from the reward $r = -log(\varepsilon_O)$. When the margin between the optimal blocklength n^* and the chosen blocklength n is too great, the server selection does not impact the overall error.

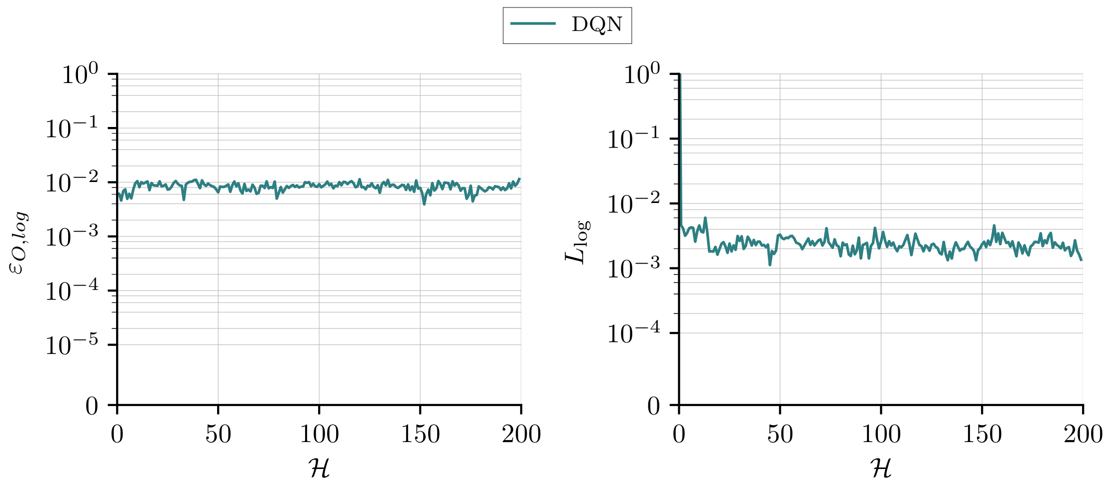


Figure 4.12: Error and loss for setup based on outdated CSI for both server selection and computing blocklength

5 Deep Deterministic Policy Gradient Based Offloading

In most application cases the offloading decision can only be based on outdated CSI. Because of the lack of analytical solutions for offloading decisions based on prior knowledge, we consider a deep RL agents to decide on blocklength n, server selection $\mathbf{A}$ and workload $\mathbf{C}$. The decision on $\mathbf{C}$ allows to adjust the workload load to the need of the conditions of the servers state. With assigning workload a lower error probability can be achieved. We consider the problem formulated in Section 3.2.4 as:

$$\begin{aligned}
\underset{\mathbf{t_1}, \mathbf{C}, \mathbf{A}}{\text{minimize}} \quad & \log\left(\varepsilon_O = 1 - \prod_k (1-\varepsilon_k)\right) && && (5.0.1a)\\
\text{subject to} \quad & \mathbf{A} \in \{0,1\}^K, && && (5.0.1b)\\
& t_1 + t_2 = T_{\max}, && && (5.0.1c)\\
& \varepsilon_k \le \varepsilon_{\max}, && \forall k \in \mathcal{K}, && (5.0.1d)\\
& c_k \le a_k c_O, && \forall k \in \mathcal{K}, && (5.0.1e)\\
& \textstyle\sum_{k=1}^{K} a_k \ge 1, && && (5.0.1f)\\
& \varepsilon_k \le a_k, && \forall k \in \mathcal{K}, && (5.0.1g)\\
& a_k(\varepsilon_{1,k} + \varepsilon_{2,k} - \varepsilon_{1,k}\varepsilon_{2,k}) \le \varepsilon_k, \forall k \in \mathcal{K} && && && (5.0.1h)
\end{aligned}$$

where t_1 is proportional to n. We assume a partition of c_o, where each server can take a workload of $c_k = v_k c_o$, where v_k is the partition factor from the set of $V = \{0, \frac{1}{8}, \frac{1}{3}, \frac{2}{3}, 1\}$. We can formulate the problem for the server selection and workload assignment to:

$$\begin{aligned}
\underset{\mathbf{C}, \mathbf{A}}{\text{minimize}} \quad & \log(\varepsilon_O) && && (5.0.2a)\\
\text{subject to} \quad & \varepsilon_k \le \varepsilon_{\max}, && \forall k \in \mathcal{K}, && (5.0.2b)\\
& c_k \le a_k c_O, && \forall k \in \mathcal{K}, && (5.0.2c)\\
& \textstyle\sum_{k=1}^{K} c_k = c_O, && && (5.0.2d)\\
& c_k = v_k c_o, && \forall k \in \mathcal{K}, && (5.0.2e)\\
& \varepsilon_k \le a_k, && \forall k \in \mathcal{K} && (5.0.2f)
\end{aligned}$$

The constraints ensure that the total workload over all servers is equal to c_o and that the workload c_k is in $v_k c_o$. The limitations of the analytical solution regarding the performance based on outdated CSI, makes the proposed approach in Chapter 4 as a combination of DQN and analytical solution inappropriate. Since for DQN, the Q-networks output size is equal to the size of action space $\mathcal{B}$, the network would increase rapidly when adding workload $\mathbf{C}$ and blocklength n to the action space. Therefore, we chose the actor-critic RL method DDPG. With DDPG larger action spaces or even continuous action values can be implemented, without necessarily increasing the networks. We now consider a case where the server selection $\mathbf{A}$, workload $\mathbf{C}$ and blocklength n is chosen based on a DDPG.

First of all, the reinforcement learning setup with its states, actions, rewards and transitions as well as the implementation of the DDPG agent with its hyperparameters and the parameterization of the environment is described in Section 5.1. Further in Section 5.2, we consider the case with perfect state information for the agent. We observe the training performance for both a static and a Rayleigh fading channel and compare the results to the optimal solution. Then in Section 5.3, we consider offloading based only on outdated CSI, where agents decides on blocklength, server selection and workload. First, we find a suitable hyperparameter combination for training the networks. Afterwards, the section shows the training performance regarding ε_O for scenarios with different number of servers K and correlation factors ρ as well as the behavior of a trained policy in a different environment.

5.1 RL setup and network configurations

The RL environment E contains the presented offloading model with the overall error probability ε_O. The UE uses the agent to decide on the offloading decisions. The agent receives state $s$and and chooses blocklength n, servers $\mathbf{A}$ and workload $\mathbf{C}$. State s formulates to $s^{(m)} = [X^{(m_0)}, X^{(m_0-1)}, X^{(m_0-W_{\text{ht}})}]$, where m_0 is the latest received time slot and W_{ht} indicates the number of historical time slot information used for state creation. The state $s$for every scenario will be introduced before the simulation results. The action space $\mathcal{B}$ of the DDPG contains: $[n, \mathbf{C}]$. While $n = b_1 \times \frac{T}{T_S}$, with $b_1 \in (0, 1]$ where from $\mathbf{C}$ $\mathbf{A}$ is generated as:

$$a_k = \begin{cases} 1, & \text{if } c_k > 0, \\ 0, & \text{if } c_k = 0 \end{cases} \tag{5.1.1}$$

The size of the action space $\mathcal{B}$ is equal to two regardless of number of servers. The server selection and workload assignment is achieved through the real valued $b_2 \in [0, 1]$, which maps the output to an index of a matrix of all possible workload assignments $\mathbf{C}$. With the index, the workload assignment and the server selection is made. For every action the environment E returns a reward r. Since the goal is to minimize the overall error probability ε_O and the Q-Learning rule aims at maximizing the reward r, we introduce the reward r for time slot m as:

$$r^{(m)} \;=\; -\log(\varepsilon_O^{(m)}) \tag{5.1.2}$$

while we assume $r^{(m)} = 1,\ \forall\varepsilon_O \leq 10^{-100}$ for numerical calculations. Accordingly, the agent is rewarded from the environment E with a higher value for a lower overall error probability ε_O. The environment E consists of communication phase with error probability ε_1 and computation phase with error probability ε_2. Figure Table 3.1 shows the parameters for each phase used in the simulation.

As shown in **??**, the DDPG agent consists of an actor network and a critic network, where each network has a target network. The actor network receives the state $s$as input and chooses action $b$as output. Therefore, the input layer size equals to the size of the state vector and the output layer size equals the size of action space, with $\mathcal{B} = [n, \mathbf{C}]$. The critic network represents the Q-network, and evaluates the action chosen by the actor network for the received state. The critics inputs is a vector of $X_c = [\mathcal{S}, \mathcal{B}]$, consisting of both state and action. The critic network receives $s$and $b$separately and concatenates them after two hidden layers. The output is the best expected value for action b. After every training cycle the transition tuple of $(s^{(m)}, b^{(m)}, r^{(m)}, s^{(m+1)})$ is stored in replay memory D_{mem} of size $U = 5000$. For every batch of size $U_{\text{b}} = 512$, random samples are taken from the memory to train the networks. While training the agent uses noise from the Ornstein-Uhlenbeck process to explore the state space A. The processes exploration and exploitation ability depends on $\sigma_{\text{max,OU}}$, $\sigma_{\text{min,OU}}$, θ_{OU} and μ_{OU}. The parameters are given for every scenario separately. The networks will be updated with a soft copy factor of $\tau = 0.001$. We chose the learning rate of the actor network as $\alpha_{\text{ac}} = \frac{\alpha_{\text{crit}}}{2}$. The simulation is separated into $\mathcal{H}$ episodes. In each episode $\mathcal{H}$ the network is trained $M_{\text{train}} = 1000$ times. Afterwards the network is tested $M_{\text{test}} = 1000$ times. Note that for testing the noise is equal to zero and the agent solely relies on its trained actor network. After each testing phase the error probability for each cycle is averaged over M_{test}. Because the values of ε_O can lie in different magnitudes of 10, we chose to take the logarithmic average, formulated as:$\varepsilon_{O,log} = \frac{\sum_0^{M_{\text{test}}} - \log(\varepsilon_O)}{\text{test}}$. The average loss L is presented similarly as $L_{O,log} = \frac{\sum_0^{M_{\text{train}}} - \log(L_O)}{M_{\text{train}}}$. For the analytical solution we also use exhaustive search.

5.2 Perfect CSI

First, we consider a static channel in Section 5.2.1 and a Rayleigh fading channel in Section 5.2.2 while having perfect knowledge about the channel condition. We analyze the convergence of DDPG in both scenarios. We consider an input $x^{(m)}$ with $W_{\text{ht}} = 0$ and $m_0 = m$ composed as:

- $\gamma^{(m)} = [\gamma_1^{(m)}, \gamma_2^{(m)}, ..., \gamma_K^{(m)}]$: vector of length K containing SNR of every link

We design two neural networks for the actor with a learning rate $\alpha = 0.00005$ and two neural network for the critic with a learning rate $\alpha = 0.0001$. We consider a actor network with four hidden layers with three of size 16 and one of size 64. The critic network uses two hidden layers for the state and action input. While the state hidden layers have a size of 16 and the action hidden layers have a size of 32. The discount factor is chosen separately for the static and fading channel case. We use batch normalization for each input in the actor and critic network as well as for the output in the critic network. For noise $\mathcal{N}$ generation we choose $\theta_{OU} = [0.5, 0.5]$, $\mu_{OU} = [0, 0]$, $\sigma_{\max,OU} = [0.8, 0.8]$ and $\sigma_{\min,OU} = 0.1, 0.05]$ as the parameters of the Ornstein-Uhlenbeck process. Since there is no impact of the decision from one time slot to the state of the next time slot, we use the current input $x^{(m)}$ as the receiving state, which formulates to $s^{(m)} = x^{(m)}$ with $W_{\text{ht}} = 1$. So the environment is fully observed. We also do not consider the queuing delay $D' = 0$. Hence, there is no dependence of computation efforts between two time slots.

5.2.1 Static Channel

For the static channel, we consider equal $\gamma_k = 10dB$ over all channels as well as $\rho = 1$. The environment parameters are shown in Table 3.1. The agent learns for $\mathcal{H} = 400$ episodes. Because of the static channel, the agent receives the same $s$at every cycle. Since there is no impact from one action to the next state we set the discount factor $\gamma_{df} = 0$. We also the runtime to $\mathcal{H} = 100$ Episodes.

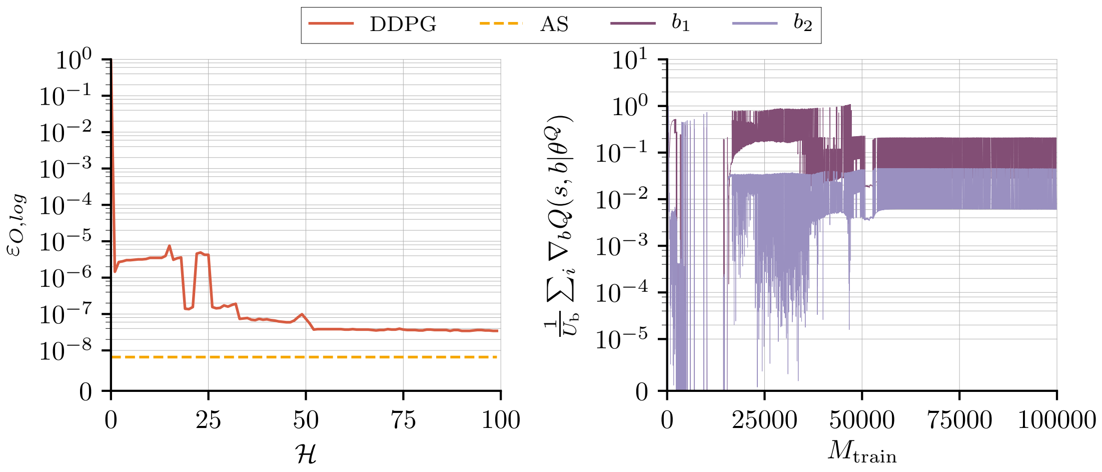


Figure 5.1: Error and action gradient for training with $K = 3$ in a static channel based on perfect CSI

As seen in Fig. 5.1, the agent oscillates and converges after roughly 50 episodes around 4×10^{-8}. The agent needs a relatively long time to converge and does not reach the analytical solution as in comparison to Section 4.2.1, where the agent quickly converges to the optimal decision. It is also shown via the action gradient that both action converge after a some oscillation. The convergence of the gradient coincides with the convergence of $\varepsilon_{O,log}$.

5.2.2 Fading Channel

For the fading channel, we consider a correlation factor of $\rho = 0.9$ with $\bar{\gamma} = 10dB$. We use the same environment parameters as in the previous section, shown in Table 3.1. The agent also learns for $\mathcal{H} = 400$ episodes. We assume no waiting delay for computation efforts, hence $D' = 0$.

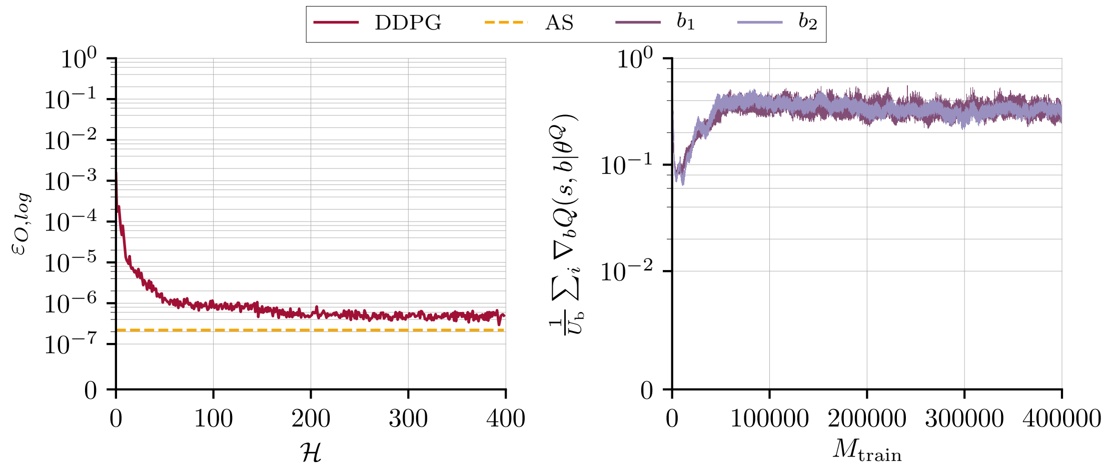


Figure 5.2: Error and action gradient for training with $K = 3$ in a fading channel based on perfect CSI

As seen in Fig. 5.2, $\varepsilon_{O,log}$ decreases over the first 70 episodes. Afterwards, $\varepsilon_{O,log}$ converges to around an error probability of 5×10^{-7}. The error still oscillates after a long training, probably because of the low number of testing cycles for the random channel or the relative a high added noise with $\theta_{OU} = 0.5$. The action gradient for both the blocklength and workload assignment also converges. The gradient for both action increase until they converge. It shows the reason for the oscillating $\varepsilon_{O,log}$. Further, we observe that the same network configuration for different K in Fig. 5.3 shows a lower error probability for $K = 2, 3$, since the network is designed for $K = 3$. Although the number of outputs stays the same for all K, the state size respectively the input size increases with K. The analytical solution on the other hand decreases with K. Therefore, the DDPG agent performance decreases for $K > 3$, which could be due to the agents hyperparameters or learning time.

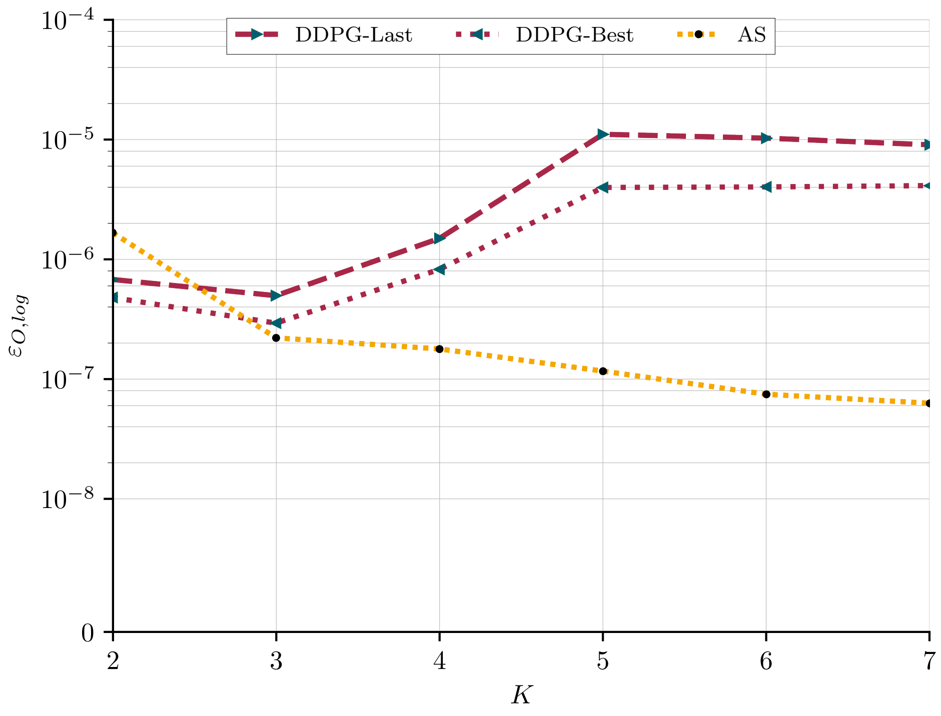


Figure 5.3: Error for different number of servers K in a fading channel based on perfect CSI

5.3 Outdated CSI

DDPG can learn the blocklength n, sever selection $\mathbf{A}$ and workload $\mathbf{C}$ in a static and fading channel based on perfect channel state knowledge. Because in application scenarios the channel is not only changing, but mostly only imperfect information is available, we now consider a Rayleigh fading channel while having outdated knowledge about the channel condition. As in Section 4.3 assume that there is a queuing delay, which is influenced by the previous workload at server k as well as by the intermission time T_k'. The rest queue from the previous time slot is described as $D_k^{(m-1)} = \frac{c_k^{(m-1)}}{f_k} - T_k'$. Consequently, the computing time at time slot m is $D_k^{(m)} = \frac{c_k^{(m)}}{f_k} + D^{(m-1)} + Q_k$. Therefore the previous time slot impacts the computation performance of the following time slot. We consider an input $x^{(m)}$ with $m_0 = m - 1$ composed as:

- $\gamma^{(m)} = [\gamma_1^{(m-1)}, \gamma_2^{(m-1)}, ..., \gamma_K^{(m-1)}]$: vector of length K containing SNR of every link

- $\mathbf{c}^{(m)} = [c_1^{(m-1)}, c_2^{(m-1)}, ..., c_K^{(m-1)}]$: vector of length K containing size of previous task loads

The action $b^{(m)}$ impacts state $s^{(m+1)}$, which introduces a correlation from actions $\mathcal{B}$ on to $\mathcal{S}$. Because of the impact from each time slot to the other we consider $s^{(m)} = [x^{(m)}, x^{(m-1)}, ..., x^{(m-W_{\text{ht}})}]$ with $W_{\text{ht}} = 3$.

In order to reach the best performance regarding ε_O as well as convergence speed, we find the best possible combination of the selected hyperparameters. In Section 5.3.1, we analyze the impact of different values for the hyperparameters. Based on the best combination of hyperparamters, we investigate the performance of the agent for different number of servers and channel gain correlation factor ρ in Section 5.3.2. The actor network consists of 4 hidden layers with 32 and 64 neurons. The critic network consists of 2 hidden layers for the action input with 32 and for the state input with 16 neurons.These two layer streams are connected and lead into a layer with 64 neurons. We apply batch normalization for the input of both the actor and critic networks, which improves the training stability in DDPG.

5.3.1 Hyperparameters

We now observe the network for different hyperparameters. The hyperparameters are the learning rate α, the discount factor γ_{df}, the random variable factor σ_{OU} and the size of the network N_{hl}, which are fixed at a certain value and one parameter is changed throughout the simulations. We chose $\alpha_{\text{crit}} = 0.0001$ so that $\alpha_{\text{act}} = 0.00005$, $\gamma_{\text{df}} = 0.0$, $\sigma_{\text{max},OU} = 0.8$ and $N_{\text{hl}} = 4$. The network consists of multiple hidden layers of size 32 and 64 depending on N_{hl}. The setup has $K = 3$ servers and and average SNR $\bar{\gamma} = 10$dB. For the analysis we consider the error probability and the action gradient. We consider the same environmental random sequence over all scenarios. While the random batch selection is different from each scenario due to different random sequences.

Learning Rate

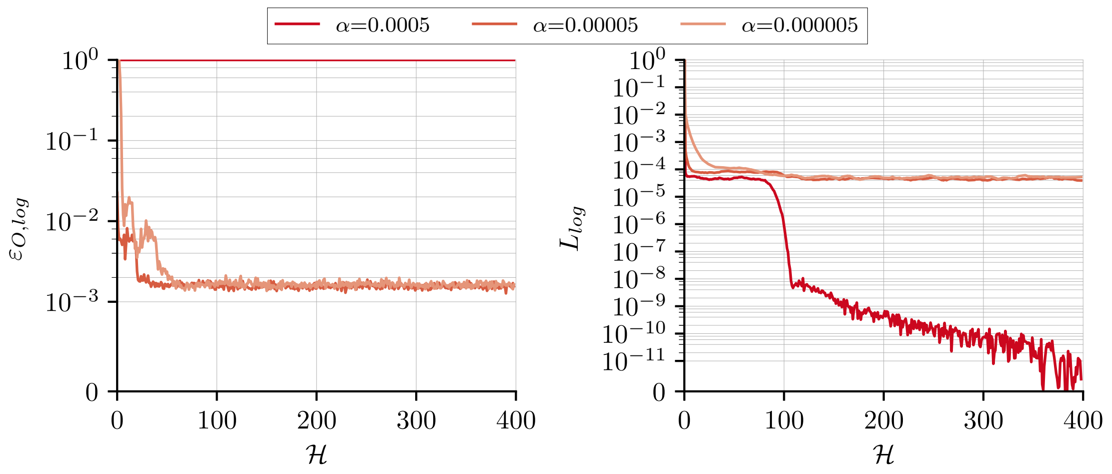


Figure 5.4: Error and action gradient for training with $K = 3$ in a fading channel based on outdated CSI for different α_{act} and α_{crit}

For the learning rate we consider $\alpha_{\text{crit}} = 0.0001$, $\alpha_{\text{crit}} = 0.00001$ and $\alpha_{\text{crit}} = 0.000001$, so that $\alpha_{\text{act}} = 0.00005$, $\alpha_{\text{act}} = 0.00005$ and $\alpha_{\text{act}} = 0.000005$. As shown in Fig. 5.4, for $\alpha_{\text{act}} = 0.000005$ the error probability does not improve during training the networks. For higher learning rates, $\varepsilon_{O,log}$ decreases, while for $\alpha_{\text{act}} = 0.00005$ the network needs more episodes for converging around the same value as for $\alpha_{\text{act}} = .0005$. The loss of the critic network shows a convergence for $\alpha_{\text{act}} = 0.00005$ and $\alpha_{\text{act}} = 0.000005$, where for $\alpha_{\text{act}} = 0.0005$ the loss of the critic network decreases.

Discount Factor

Now, we observe the behavior of the DDPG agent for discount factors of $\gamma_{\text{df}} = 0.0$, $\gamma_{\text{df}} = 0.5$, $\gamma_{\text{df}} = 0.9$ and $\gamma_{\text{df}} = 1$. It is shown in Fig. 5.5 that for $\gamma_{\text{df}} = 0.5$ and $\gamma_{\text{df}} = 0.9$ $\varepsilon_{O,log}$ decreases to a lower saturation than for $\gamma_{\text{df}} = 0$ and $\gamma_{\text{df}} = 1$. The loss of the critic network shows that for $\gamma_{\text{df}} = 1$ the loss oscillates the most and does not really saturates at a certain value. The agent with the lowest discount factor at $\gamma_{\text{df}} = 0$ converges the fastest to the smallest loss, which its also shown for the Q-network in Fig. 4.6. For $\gamma_{\text{df}} = 0.9$ the loss increases steadily, while for $\gamma_{\text{df}} = 0.5$ the loss converges as well. The reason for $\gamma_{\text{df}} = 0.0$ not decreasing in $\varepsilon_{O,log}$ as much could lie in a different channel realization or different batch selection.

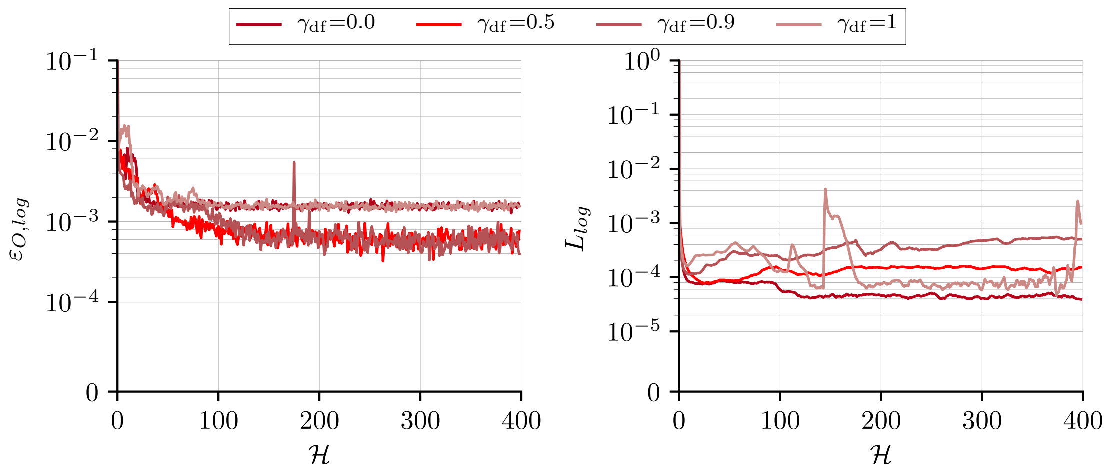


Figure 5.5: Error and action gradient for training with $K = 3$ in a fading channel based on outdated CSI for different γ_{df}

Random Noise Factor

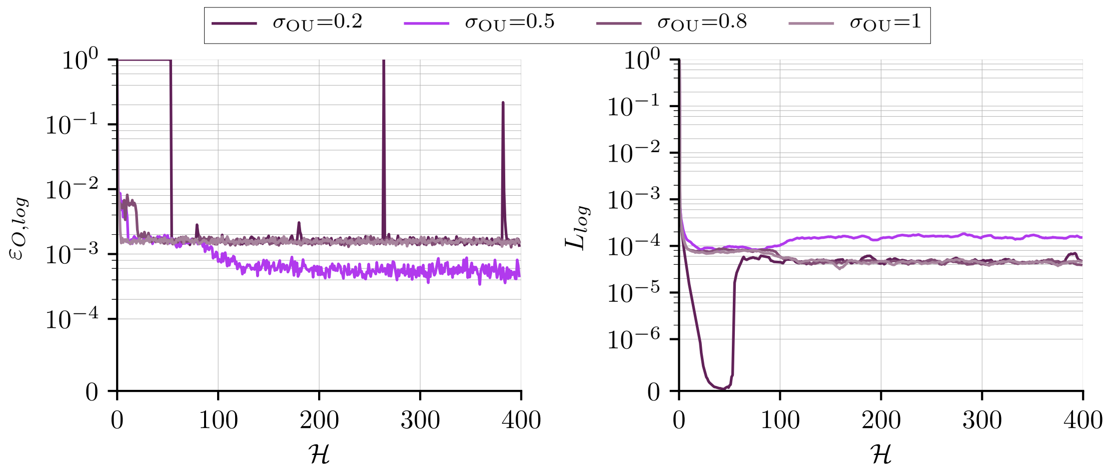


Figure 5.6: Error and action gradient for training with $K = 3$ in a fading channel based on outdated CSI for different $\sigma_{\text{max,OU}}$

With σ_{OU}, the influence of the random variable is determined. With an increasing number of steps σ_{OU} decreases from $\sigma_{\text{max,OU}}$ to $\sigma_{\text{min,OU}}$, which leads to less noise for later episodes. Fig. 5.6 shows that for lower $\sigma_{\text{max,OU}} = 0.2$ and $\sigma_{\text{max,OU}} = 0.5$, the loss increases after few episodes of training. With a low $\sigma_{\text{max,OU}}$ the agent needs more time to explore the action space.

Number of Hidden Layers

We consider three scenarios with different number of hidden layers for both the actor and the critic network. For $N_{\text{hl}} = 1$ hidden layer we consider the actor network with one layer with 64 neurons. For $N_{\text{hl}} = 4$ hidden layer, three layers with 32 neurons and one with 64 are considered. Using $N_{\text{hl}} = 8$ eight hidden layers, we design an actor network with seven 32 neuron layer and one with 64 neurons. The critic also increases its numbers of hidden layers accordingly from one to two to four. While each state layer has 16 neurons and each action layer has 32 neurons. Similar to the observations of the DQN agent, the network with $N_{\text{hl}} = 1$ hidden layer decreases its error probability $\varepsilon_{O,log}$ quicker than for the agents with more layers. While the agent with $N_{\text{hl}} = 4$ hidden layer still converges around the same value as with $N_{\text{hl}} = 1$ hidden layer, the agent with $N_{\text{hl}} = 8$ hidden layers does not improve its performance at all. It is also noted that for the critic network the loss decreases rapidly for eight hidden layers, while for $N_{\text{hl}} = 1$ and $N_{\text{hl}} = 4$ hidden layer it converges.

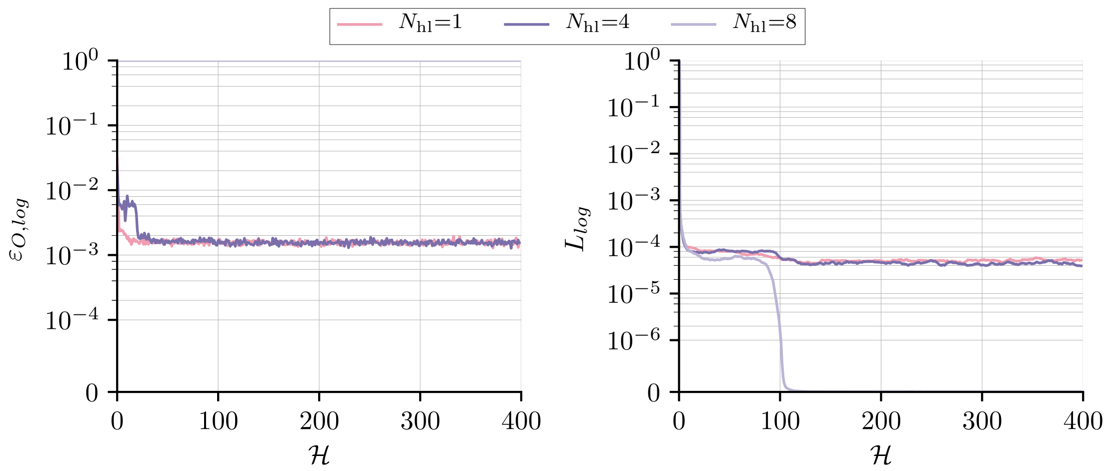


Figure 5.7: Error and Action Gradient for training with $K = 3$ in a fading channel based on outdated CSI for different N_{hl}

5.3.2 DDPG based Offloading Scenarios

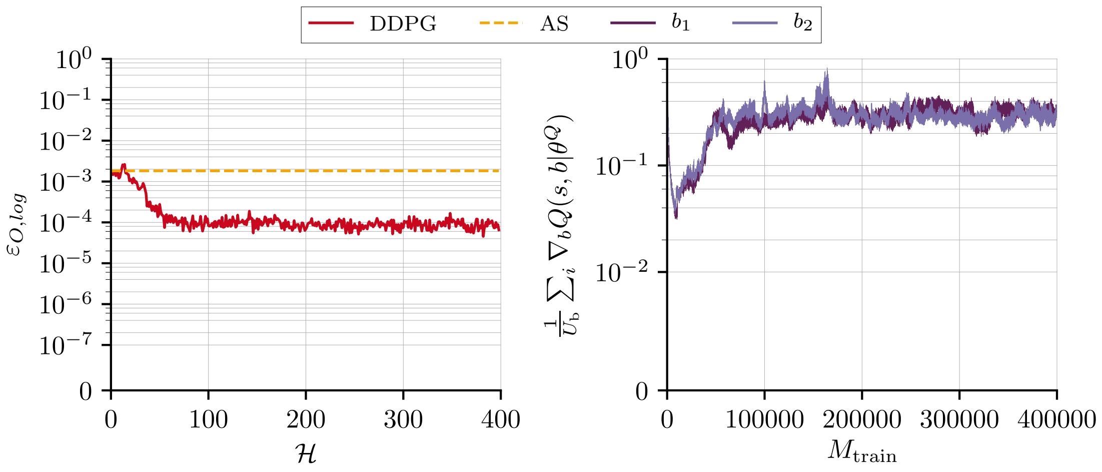


Figure 5.8: Error and action gradient for training with $K = 3$ in a fading channel based on outdated CSI

First we consider the same network architecture for the actor and critic as in Section 5.2.2. We train the network $M_{\text{train}} = 1000$ times and test the network $M_{\text{test}} = 1000$ times. The simulation runtime is $\mathcal{H} = 400$. Note that for different number of server K the random sequence regarding channel gain and noise creation is different. Also, between each test run there is a different random sequence regarding channel gain.

We observe a learning scenario for $K = 3$ servers with an average SNR $\bar{\gamma} = 10$dB. Over the episodes its shown in Fig. 5.8 that the agent learns a policy, which saturates at just below 10^{-3}. After around 330 episodes the agent learns how to decrease the error probability $\varepsilon_{O,log}$ to just under 10^{-4} while having a small oscillation between each testing. The convergence is also shown by the action gradient, where b_1 is the gradient for determining the blocklength and b_2 for assigning the workload. The DDPG outperforms the benchmark of the of the analytical solution after roughly 10 episodes. For the following scenarios we consider a network which is dependent on the state size. We design an actor network with $N_{\text{hl}} = 4$ hidden layers with three layers containing three times the state size. The last layer has neurons four times the state size. The critic has two layers with neurons two times the state size for the state input and three times the state size for the action input. These two layer sequence are concatenated with a layer of the size of four times the state size. Both network use batch normalization for the input layer and the critic uses batch normalization before and after the concatenation.

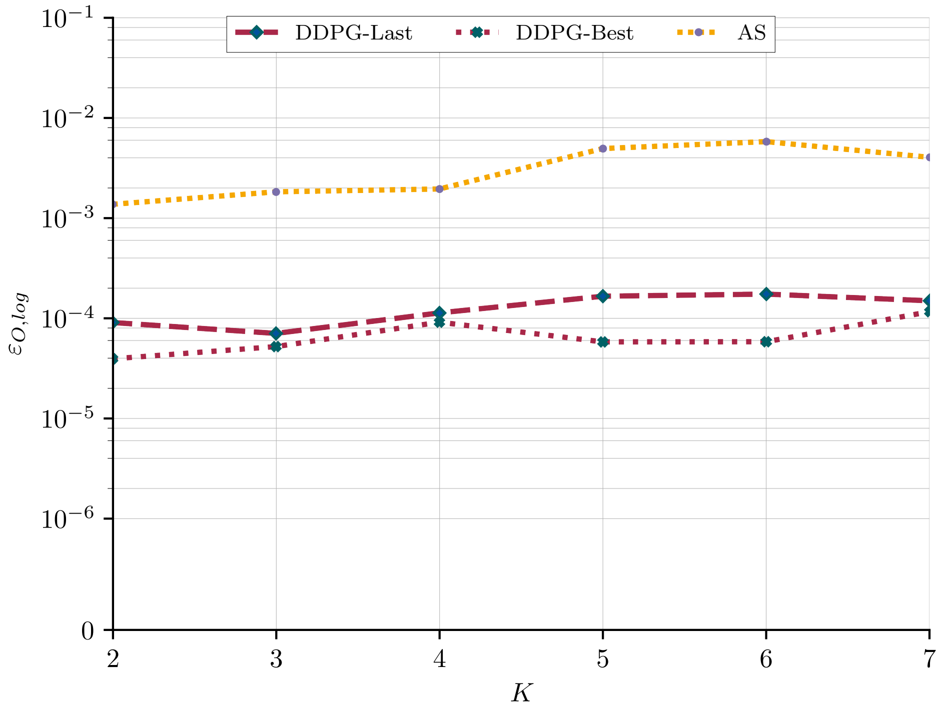


Figure 5.9: Error in a fading channel based on outdated CSI for different number of servers K

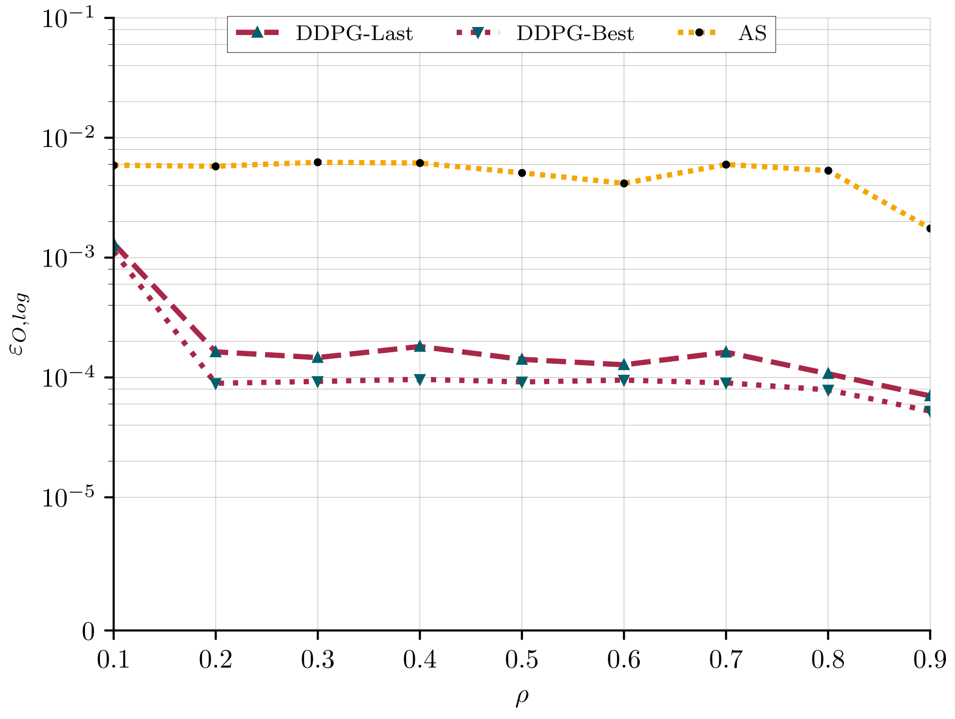


Figure 5.10: Error in a fading channel based on outdated CSI for different channel correlation factors ρ

When observing the error probability over K, it is worth noting that the network size as well since the number of neurons of each layer depend on the state size. In Fig. 5.9 its shown that the best found error probability over all K stays roughly the same. The performance after the last training set worsens a little bit with increasing K, because of the larger network and state size. Note that the number of training cycles stays the same for all scenarios. Opposite to Fig. 4.10 the analytical solution increases with the number of K, which could also be due to the lack of channel initialization. For improving the performance for larger K, some changes in the networks design and parameterization could help. For example different values for the parameters of the noise process for better exploration over larger action space $\mathcal{B}$. Further, longer training would benefit larger networks..

When observing in Fig. 5.10 the performance for different channel correlation factors between time-slots, its shown that $\varepsilon_{O,log}$ decreases with a higher correlation ρ. For lower ρ, the agent has less correct information and the decisions are randomly regarding the channel. For higher ρ the agent receives more knowledge, hence the agent makes better decisions. Since the performance is also dependent on the random batch selection, some agents do not converge to the same value in the same training time. It is also shown that the analytical solution slightly decreases over ρ.

5.3.3 Trained Agent in Different Environment

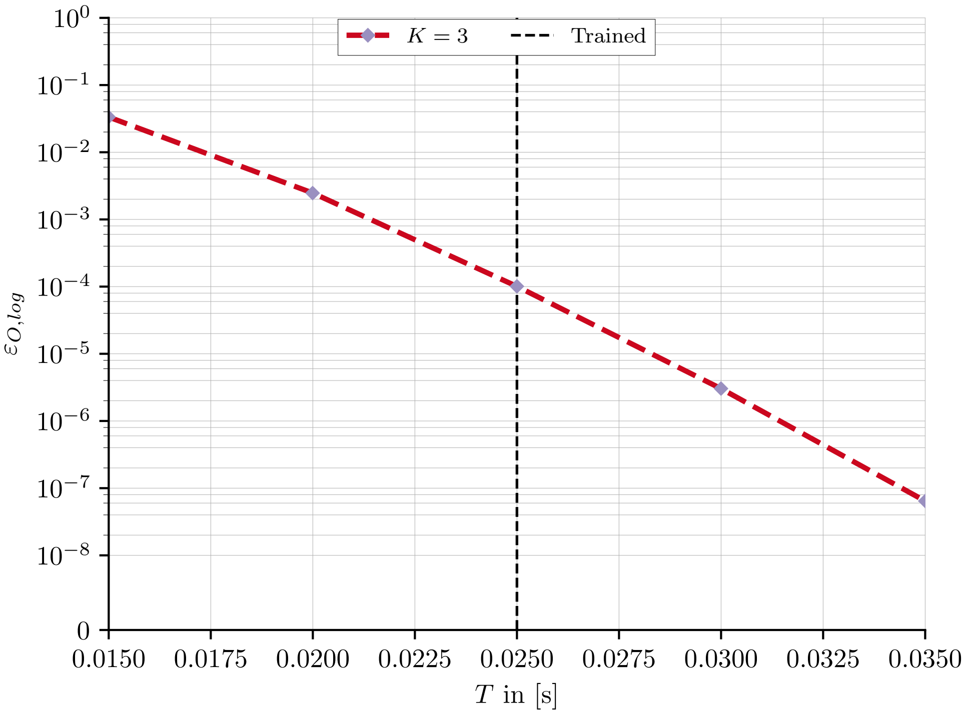


Figure 5.11: Error based on learned DDPG agent over different time delay limits T

In reality, the requirements of the user change and we have to assume that communication and computation parameters differ from the original learned parameters when interacting with the environment. These parameters could be the available bandwidth due to change of traffic, frequency, packet size and time delay limit. Regarding the computation, the need of the user could change in workload, computation power due to other allocated resources and also the influence of waiting time due high demand of offloading could change. We consider now cases where the time delay limit changes in the communication phase as well as the total workload in the computation phase. The time delay limit could change due to different priorities of data transmissions, where some information is important at a certain time than the other. The workload could change due to different applications. We now consider a trained DDPG agent for the parameters presented in Table 3.1. The network is trained with a time delay limit of $T = 25 \times 10^{-3}$ and a workload of $c_o = 24 \times 10^{6}$. We consider the case with $K = 3$ servers.

It is shown in Fig. 5.11 that $\varepsilon_{O,log}$ decreases with T. $\varepsilon_{O,log}$ decreases equally before and after the trained $T = 25$ms. It is shown that even when the agent is not trained for a time delay, it still can perform for lower T as well as for slightly higher T. Because the agent still makes the same decision as for the trained parameter environment the error probability is influenced almost linearly by T.

As seen in Fig. 5.12, the error probability increases over the total workload c_o. For workloads smaller than $c_o = 60 \times 10^6$ cycles all scenarios perform the same. For workloads greater than $c_o = 40 \times 10^6$ cycles, the agent converges to a better $\varepsilon_{O,log}$. Obviously, DDPG performs better with a lower workload c_o. For higher c_o the error probability saturates.

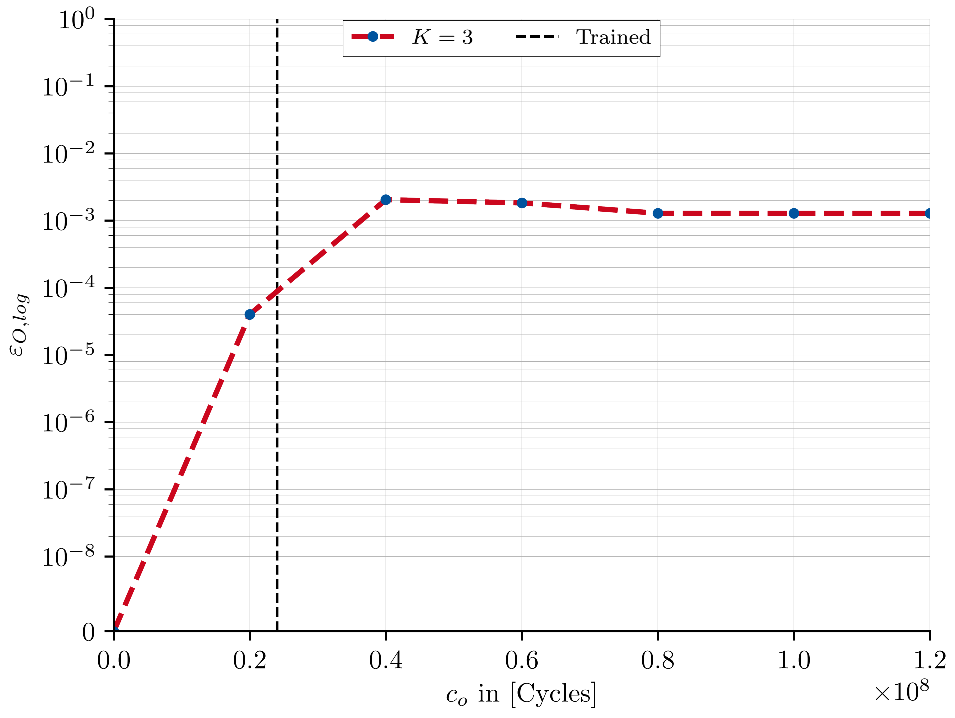


Figure 5.12: Error based on learned DDPG agent over different total workloads c_o

6 Conclusion and Outlook

Throughout this thesis, we study the ability of two deep reinforcement learning methods to make offloading decisions in an ECN environment. We introduce an offloading model and its reliability optimal design based on perfect knowledge. The design is jointly optimally selecting multiple servers, optimally assigning partitioned-workload and blocklength. We train a DQN agent in the proposed model environment and show its convergence. DQN is able to decide on server selection when using the optimally assigned blocklength. The agent is able to converge close to the optimal solution in a static channel. Due to the increased state space the agent performs close to the optimal solution in a fading channel based on perfect CSI. Based on imperfect information the DQN agent converges to a policy comparably to the analytical solution. Furthermore, we introduce the DDPG method for making offloading decisions. The agent is able to converge to a policy deciding on blocklength, server selection and workload allocation. Based on numerical simulations, the DDPG agent can learn a policy in a static and fading channel based on both perfect and outdated CSI and server states. The DDPG performs better for an increased channel correlation. The DDPG agent outperforms the analytical solution when only outdated information is available.

When designing a deep RL agent, multiple factors have to be taken into account. First, the networks used for function approximation have to be large enough to receive and process the incoming information from the environmental state. However, when the network is too large, the agent takes too much time to converge. In particular regarding DDPG, the learning process becomes unstable very quickly. Other than in DQN, DDPG is very sensitive to the chosen parameters. The combination of learning rate, number of hidden layers, discount factor, batch size and exploration noise is very important when designing the agent. When comparing different deep RL setups to each other or with existing methods, pseudo-random sequences are helpful for reproducibility regarding fading channel realization and random batch selection. When designing a network, the input data has to be taken into account. For better training performance each input should have approximately the same value range. Because of the different characteristics, the values may differ i.e: SNR is typically a few magnitudes of 10 lower than the workload. A high difference in values makes it harder for the network to train its weights accordingly, which results in longer or even bad training performance. The input values can be adjusted manually or by using a batch normalization layer at the beginning of each network.

Expanding the agent for serving a larger amount of servers needs more parameter adjustment than just increasing the number of neurons.

It is shown that deep RL is able to make offloading decision based on outdated information for a small number of servers. Since DDPG needs very detailed configuration, further work regarding an optimal parameterization should be done. Also, the limitations of the agents regarding the maximum number of served servers should be taken into consideration. For more trustful analysis a simulation environment with a fixed pseudo-random sequence for the testing phase, in particular for the channel realization, but also for the random batch selection would be helpful. The impact of training the same network with different environmental parameters and its performance compared to a network trained only in one environment may be interesting for future work. Another aspect in improvement of the agent would be to implement a network which decides on smaller workload partitions or even on continuous workload. When deciding on the workload individually, the problem of meeting the workload constraints could arise. Besides DDPG, further investigation should study other methods as Advantage Actor Critic (A2C)/Asynchronuos Advantage Actor Critic (A3C), Soft Actor Critic (SAC) or Proximal Policy Optimization (PPO), which are also able to implement continuous actions.

List of Figures

List of Tables

Bibliography

[1] S. A. A. Shah, E. Ahmed, M. Imran, and S. Zeadally, “5g for vehicular communications,” *IEEE communications magazine*, vol. 56, no. 1, pp. 111–117, 2018.

[2] R. Vannithamby and S. Talwar, *Towards 5G: Applications, requirements and candidate technologies.* John Wiley & Sons, 2017.

[3] M. Erol-Kantarci and S. Sukhmani, “Caching and computing at the edge for mobile augmented reality and virtual reality (ar/vr) in 5g,” in *Ad Hoc Networks.* Springer, 2018, pp. 169–177.

[4] W. Shi and S. Dustdar, “The promise of edge computing,” *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[5] N. Fernando, S. W. Loke, and W. Rahayu, “Mobile cloud computing: A survey,” *Future generation computer systems*, vol. 29, no. 1, pp. 84–106, 2013.

[6] “3gpp release 16.” [Online]. Available: https://www.3gpp.org/release-16

[7] T. de Cola, E. Paolini, G. Liva, and G. P. Calzolari, “Reliability options for data communications in the future deep-space missions,” *Proceedings of the IEEE*, vol. 99, no. 11, pp. 2056–2074, 2011.

[8] F. Boccardi, R. W. Heath, A. Lozano, T. L. Marzetta, and P. Popovski, “Five disruptive technology directions for 5g,” *IEEE communications magazine*, vol. 52, no. 2, pp. 74–80, 2014.

[9] E. Paolini, C. Stefanovic, G. Liva, and P. Popovski, “Coded random access: applying codes on graphs to design random access protocols,” *IEEE communications magazine*, vol. 53, no. 6, pp. 144–150, 2015.

[10] G. Durisi, T. Koch, and P. Popovski, “Toward massive, ultrareliable, and low-latency wireless communication with short packets,” *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1711–1726, 2016.

[11] Y. Zhu, Y. Hu, T. Yang, and A. Schmeink, “Reliability-optimal offloading in multi-server edge computing networks with transmissions carried by finite blocklength codes,” in *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2019, pp. 1–6.

[12] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “A brief survey of deep reinforcement learning,” *arXiv preprint arXiv:1708.05866*, 2017.

[14] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[15] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[16] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[18] M. L. Puterman, *Markov Decision Processes.: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 2014.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, p. 529, 2015.

[20] Hado V. Hasselt, “Double q-learning,” in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc, 2010, pp. 2613–2621. [Online]. Available: http://papers.nips.cc/paper/3964-double-q-learning.pdf

[21] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[22] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, Eds., *Deterministic policy gradient algorithms*, 2014.

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[24] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Physical review*, vol. 36, no. 5, p. 823, 1930.

[25] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[26] M. A. Nielsen, *Neural networks and deep learning.* Determination press San Francisco, CA, USA, 2015, vol. 25.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] S. Liang and R. Srikant, "Why deep neural networks for function approximation?" *arXiv preprint arXiv:1610.04161*, 2016.

[29] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[30] M. Telgarsky, "Benefits of depth in neural networks," *arXiv preprint arXiv:1602.04485*, 2016.

[31] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance optimization in mobile-edge computing via deep reinforcement learning," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall).* IEEE, pp. 1–6.

[32] 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, no. 1, pp. 44–55, 2017.

[33] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[34] T. Yang, Y. Hu, M. C. Gursoy, A. Schmeink, and R. Mathar, "Deep reinforcement learning based resource allocation in low latency edge computing networks," in *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, 2018, pp. 1–5.

[35] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet of Things Journal*, 2018.

[36] L. Huang, S. Bi, and Y. J. Zhang, “Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks,” *IEEE Transactions on Mobile Computing*, 2019.

[37] Z. Chen and X. Wang, “Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach,” *arXiv preprint arXiv:1812.07394*, 2018.

[38] Y. Dai, Du Xu, S. Maharjan, G. Qiao, and Y. Zhang, “Artificial intelligence empowered edge computing and caching for internet of vehicles,” *IEEE Wireless Communications*, vol. 26, no. 3, pp. 12–18, 2019.

[39] R. K. Mallik, “On multivariate rayleigh and exponential distributions,” *IEEE Transactions on Information Theory*, vol. 49, no. 6, pp. 1499–1515, 2003.

[40] J. D. C. Little, “A proof for the queuing formula: L= l w,” *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[41] S. Coles, J. Bawa, L. Trenner, and P. Dorazio, *An introduction to statistical modeling of extreme values.* Springer, 2001, vol. 208.

[42] M. Lubin, E. Yamangil, R. Bent, and J. P. Vielma, “Polyhedral approximation in mixed-integer convex optimization,” *Mathematical Programming*, vol. 172, no. 1-2, pp. 139–168, 2018.