TU WIEN Informatics

Cost-Aware Neural Network Splitting and Dynamic Rescheduling for Edge Intelligence

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

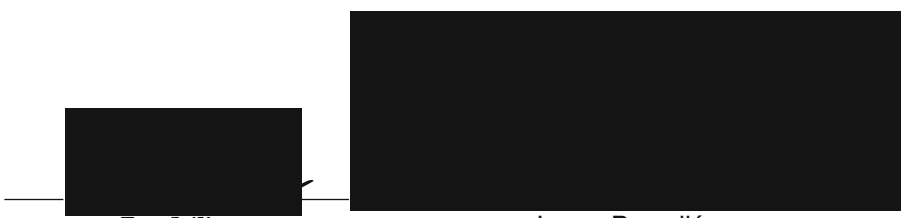
Daniel Luger
Registration Number 11778903

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Mag.a rer.soc.oec. Dr.in rer.soc.oec. Ivona Brandić
Assistance: Dott. mag. Dr. Atakan Aral

Vienna, 26th June, 2023



Daniel Luger Ivona Brandić

Erklärung zur Verfassung der Arbeit

Daniel Luger

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. June 2023

Daniel Luger

Abstract

With the rise of IoT devices and the necessity of intelligent applications, inference tasks are often offloaded to the cloud due to the computation limitation of the end devices. Yet, requests to the cloud are costly in terms of latency. Therefore, a shift of the computation from the cloud to the network's edge is unavoidable for time-sensitive applications. This shift is called edge intelligence and promises lower latency, among other advantages. However, some algorithms, like deep neural networks, are computationally intensive, even for local edge servers (ES). Such DNNs can be split into two parts to keep latency low and distributed between the ES and the cloud. We present a dynamic scheduling algorithm that considers real-time parameters like the clock speed of the ES, bandwidth, and latency and predicts the optimal splitting point regarding latency. Furthermore, we estimate the overall costs for the ES and cloud during run-time and integrate them into our prediction and decision models. We present a cost-aware prediction of the splitting point, which can be tuned with a parameter toward faster response or lower costs. We tested our rescheduling algorithm on a test bed with a Raspberry Pi as edge and an AWS instance as a cloud server. The results demonstrate that we achieved a 60.84% decrease in cost compared to the optimal splitting point regarding latency with an increase in latency of only 25.92% for the AlexNet CNN when the edge server is rented.

Contents

CHAPTER 1

Introduction

In the past several years, the number of IoT devices has increased enormously. According to Cisco, 50% of the worldwide network devices will be IoT devices by 2023, reaching 14.7 billion devices [12]. At the same time, the complexity of the applications running on IoT devices increases as well. Therefore, more and more IoT devices require Artificial Intelligence (AI) to solve their tasks. AI is a broad term with a giant field of use cases and implementations [18]. One prevalent form is Machine Learing (ML) which creates a model trained on a data set and then used to predict or decide without explicitly programmed to do so. Deep Learning (DL) is a ML subcategory consisting of multiple connected layers [26]. DL has gained popularity in the last several years and is already well integrated into our daily lives, from virtual assistants to autonomous driving.

Deep learning uses Deep Neural Network (DNN)s to solve problems similar to the neurons in our brains [7]. The significant advantage of DNN is that some problems are of such complexity that it would be impossible to describe them entirely in mathematical notation. DNNs use a strategy to iterate to a solution without knowing a complete function. Such networks run primarily in cloud data centers with nearly unlimited computational resources. As applications on IoT devices get increasingly complex, the usage of DNNs is required. Therefore those devices offload their tasks to data centers for inference, and the outcome is returned to the IoT device. This makes sense for non-time-sensitive applications because sending a task to the cloud and back costs substantial time. However, many applications have stringent time requirements, such as autonomous driving, distance surgery, and game streaming. Therefore, the computation of the NNs can be shifted to a server in the user's proximity, called an Edge Server (ES). This shift of AI into an ES is called Edge Intelligence (EI) [40, 16], which can significantly reduce the processing latency [32] and improve privacy [3].

However, the edge might not provide sufficient computation resources, so large DNNs could not be computed at an acceptable time at the edge, rendering the shift ineffective. Different methods, such as model pruning or quantization, exist that target the deployment of

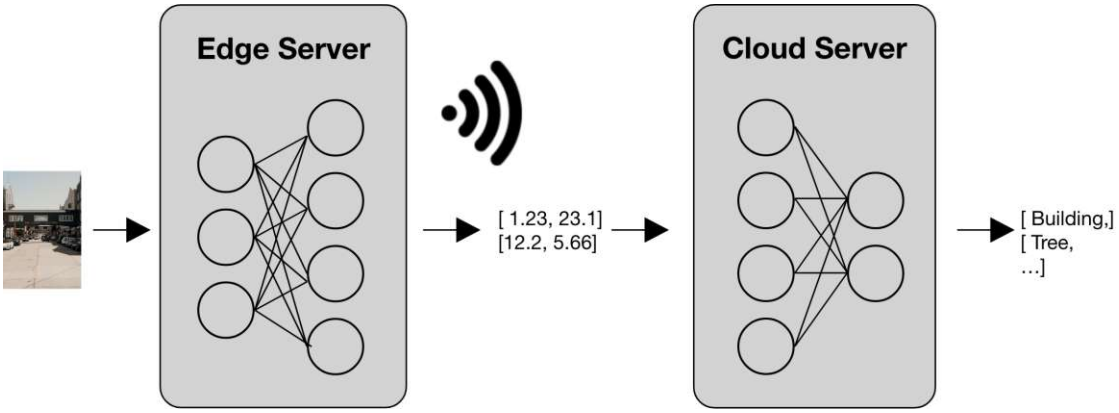


Figure 1.1: Splitting of a DNN over Edge and Cloud Servers [30].

DNNs on edge resources. Pruning is a compression technique that removes less important weights or filters, whereas quantization deals with mapping the model parameters and activations into low-precision quantized levels to avoid costly FLOPs. However, those two approaches can suffer from accuracy loss [35]. Another possible solution, which preserves accuracy, is to split the DNN into two parts, the first running at the edge and the second in the cloud shown in Figure 1.1. By doing this, the edge has to compute less, and only a small amount of inter-layer data is sent to the cloud compared to the raw input data. However, environments change over time, and therefore, the best splitting point of the DNN is not always the same [24]. Therefore, the splitting point might have to change during runtime since the DNNs could continue executing for days or weeks.

This volatility also happens at the physical hardware of edge and cloud. Cloud servers are usually rented; therefore, a virtual instance is provided. This instance has virtual Central Processing Unit (CPU)s (vCPU) with a fixed clock speed. On the other hand, ES can be bought and installed at the edge. Their hardware is not virtual but physical. Due to this, the clock speed of the ES can change depending on various parameters such as energy supply and load. Due to the changing clock speed, the computation time of the edge server varies. As mentioned before, ES can be bought and installed by users themselves. Meaning, they do not have to rent an ES and pay per usage. Therefore users can save costs by running as much as possible on their own edge device instead offloading to the cloud.

Previous proposals, such as Neurosurgeon [24], focus on finding the optimal splitting point. By building prediction models to reduce energy consumption and latency with layer-specific parameters, it is possible to infer the optimal splitting point. However, this approach does not take costs into account. Lin et al. [28] consider the cost between an end device, ES, and cloud. Yet, this algorithm neither co-optimizes cost and latency nor is dynamic. On the other hand, Gao et al. [17] apply multi-optimization for cost, energy, and latency and is dynamic but focuses on a mobile device rather than an edge server; hence, no cost is calculated at the edge. Table 2.1 summarizes and compares the previous

work to ours. Furthermore, it does not consider clock speed or cover the strategy that the ES can be owned. None of the previous papers included weight to optimize cost, latency, or a trade-off.

Three research questions arise from the necessity of a cost-aware rescheduling algorithm between edge and cloud, including the possibility of owning the ES. Furthermore, a change in clock speed, bandwidth, and network latency is considered.

- ***i)* What is the percentage optimization in cost and latency achieved through a trade-off algorithm compared to strategies exclusively focused on each parameter individually?** This first research question examines the potential efficiency of a trade-off algorithm aimed at striking a balance between cost and latency. The focus of the investigation is to quantify the proportionate reduction in latency and cost achieved when implementing this algorithm versus the strategies singularly dedicated to either cost or latency optimization. The relevance of this research is underscored by its potential implications for optimizing resource utilization and bolstering operational performance through algorithmically balanced decision-making processes.

- ***ii)* How effective is live clock speed as a predictive variable in estimating the execution time required for a Deep Neural Network (DNN) on an Edge Server?** This subsequent research question aims to explore the feasibility and accuracy of integrating live clock speed into a predictive model for DNN execution time on an ES. The ultimate goal of this research is to employ this predictive model for identifying the most efficient splitting point of the DNN, thus improving processing efficiency on the edge.

- ***iii)* What might be the design and functional characteristics of a multi-optimization algorithm incorporating cost, bandwidth, clock speed, and network latency for the optimization?** The third research question centers on the conceptualization and potential features of a dynamically adjusted rescheduling algorithm, which considers live operational parameters and cost factors. The objective of this research is to uncover strategies that could potentially enhance overall system efficiency through comprehensive, multi-factor optimization.

This research explores predictive models that integrate variables such as current clock speed, bandwidth, latency, Edge Server (ES) cost, and cloud cost. We introduce a dynamic rescheduler, designed to optimize system cost and latency parameters. The proposed methodology incorporates a weighting mechanism that allows for a use-case-specific algorithm adaptation. This means that users can modulate the relative importance of cost or latency according to their specific operational requirements. For instance, a user running a time-sensitive application with less emphasis on cost can increase the weight attributed to latency, thereby minimizing system latency. The efficacy of this proposed approach is assessed using an authentic testbed composed of a Raspberry Pi serving

as the ES and an Amazon Web Services (AWS) instance as the Cloud Server (CS). A limited number of results from this work were presented at the EdgeSys23 conference in Rome, as documented in the referenced paper [30].

- Chapter 2 gives an overview of the state-of-the-art technologies used in this work. Furthermore, it introduces DNN splitting and similar methods for fast inference and summarizes the related works.
- Chapter 3 provides a detailed depiction of the system model integral to this thesis, elaborating on each constituent component and their respective functionalities. Consequently, the third research question regarding the design and functionalities of the algorithm is solved in this chapter.
- Chapter 4 explains the prediction of different latencies (edge, cloud, communication, and overall latency). Furthermore, it explains the cost model of edge and cloud and describes the trade-off algorithm between edge and cost. Therefore, this chapter further explains the third research question regarding the design implementation of the prediction models and proposes a solution for using the clock speed as an edge latency predictor, which explains the feasibility of research question three.
- Chapter 5 describes how a DNN model is split and briefly introduces used DNN models.
- Chapter 6 shows the experimental setup of our test bed. Furthermore, it describes the use case of this work and how everything is measured.
- Chapter 7 conducts an analysis of the collected measurements and presents the resultant findings. Results of the achieved trade-off optimization are presented, which solves the first research question.
- Chapter 8 offers a comprehensive conclusion to the thesis, encapsulating the key findings and insights. The second research question is also solved, regarding the effectiveness, in this chapter by providing results of the prediction models precision.

CHAPTER 2

Overview

2.1 Edge Computing

There is a significant degree of content overlap between the material presented in this chapter and my bachelor's thesis [29]. With the increasing number of IoT devices located at the network's edge, the data generated edge is enormous. Those IoT devices can be very simplistic such as a light switch, although the edge devices get more intelligent, and with this, the applications running on those devices get more complex. Due to the often limited IoT devices regarding energy, memory, and computational power, the algorithms often run in big cloud data centers. As a result of this, lots of data is sent to the cloud for processing, which causes some shortcomings [8]:

- **Real-time:** Due to the growing amount of devices at the edge, combined with cloud computing, where everything is sent to the cloud, lots of data is sent for processing. This causes enormous traffic with intermediate data, which is not required in the cloud. This limits the network bandwidth, and therefore applications suffer from increased latency times. Those limitations are no problem for non-time-critical applications like smart homes. Still, high latency times could be fatal for time-critical applications such as autonomous driving or remote surgeries.

- **Security and privacy:** Due to the proximity of the IoT devices to the users, security, and privacy are essential. Device security breaches are limited as long as the data stays at the edge. Nevertheless, with the cloud computing approach, sensitive and private user data is uploaded to the cloud and vulnerable to security attacks. Energy consumption: The enormous cloud usage results in extreme energy consumption at the data centers. Even strategies to run them more efficiently cannot meet the increasing demand.

- **Location:** Due to the costs and sheer size of today's data centers, they cannot be located everywhere. Therefore, only a limited set of cloud servers are available. This means that they are geographically often far away from the user, and a request must travel hundreds of kilometers to reach the intended server. This results in an increased latency time and, therefore, the cloud is hardly feasible for real-time systems.

Those problems are just a few that result from the extensive usage of the cloud and show the necessity of another solution. Edge computing (EC) is a promising enabler for solving some of the problems in our current internet. EC follows the strategy to bring the cloud data centers in closer proximity to the user where the data is produced. Figure 2.1 shows that the user equipment and the cloud are connected through the edge servers. This brings a lot of advantages and opportunities for new technologies [8, 29]: Fast data processing and analysis, real-time: Due to the location of the edge servers, which are close to the user, the latency time to this server is lower than to the cloud. For example, a hospital could have its edge server installed somewhere in the hospital. Health-specific applications like computer vision for X-rays could run directly in the hospital without requiring a request to the cloud. This is also very important for highly time-sensitive applications like autonomous driving or augmented reality, where latencies of 10 ms or less are required [37]. Furthermore, if the cloud is needed, nevertheless, the edge can preprocess the data to minimize the data size. This helps to limit the traffic to the cloud and enables, therefore, a lower latency. Security: When using CC, all the data is sent to the cloud. This enables possible lack of security during the transmission or even when stored in the cloud. By using EC, the data stays local, which makes it harder for an

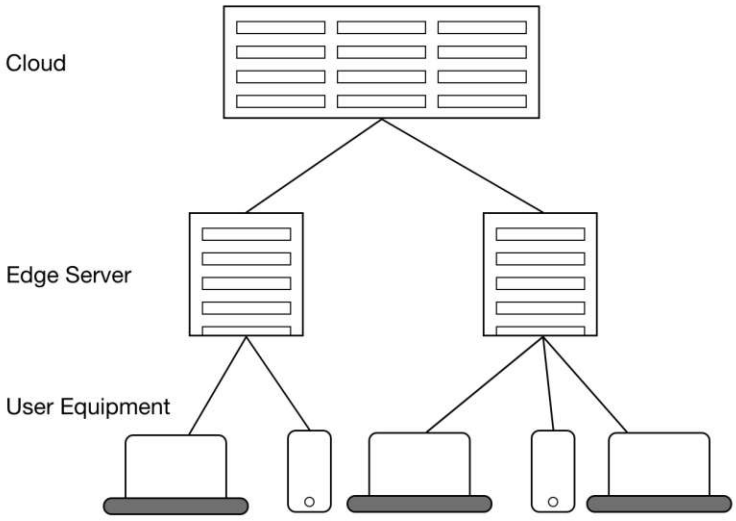


Figure 2.1: The different layers in the cloud continuum [29].

attacker. Furthermore, personal data like health data could stay at the hospital, and only anonymized information is forwarded to the cloud. Low cost, low energy consumption, low bandwidth cost: Because the data does not need to be uploaded to the cloud, it occupies not too much network bandwidth. Therefore, the network bandwidth, as well as the energy consumption of intelligence devices at the edge, is reduced. Companies, furthermore, can reduce the costs of processing data with local equipment. To sum up, edge computing improves computing efficiency, reduces the energy consumption of local equipment and, therefore, prices, and reduces bandwidth pressure.

2.2 Mobile Edge Computing

Mobile Edge Computing (MEC) is a standard used for mobile devices. MEC primarily operates in the Radio Access Network (RAN). RAN is the communication between a user equipment UE and the core network located at the Base Station (BS). The idea is to put an ES in between the UE and the core network to reduce costly requests to the cloud. Therefore, when combining 5G with MEC, very low latencies can be reached, which enables real-time applications. MEC was standardized in the European Telecommunications Standards Institute (ETSI) Industry Specification Group (ISG) and is characterized by [23, 29]:

- On-Premises. The MEC platforms run isolated from the rest of the network and have access to local resources. For machine-to-machine communication dealing with security systems, they need a high level of safety.
- Proximity. The MEC platform is near the UE and can gather essential information for analytical purposes. This information can be used for business-specific applications if the server can access the device and the applications.
- Lower Latency. Due to the good location, latency can be reduced because, in the best case, a request needs only one hop to the ES, where the required information or algorithm is stored. This enables new real-time applications where the traditional network fails.
- Location Awareness. The MEC platform is aware of the location of the user. This brings business-specific advantages like automatic language settings or frequent search requests in a particular area. Network Context Information. Because the MEC platform is aware of the bandwidth and the network conditions, companies can use those to optimize their application. For example, depending on the network conditions, a video streaming service could deliver the best video resolution.

2.3 Deep Learning

In the past years, the term AI got more prevalent in our society and daily use cases like speech recognition through Apples Siri, image recognition like Apple's search feature in

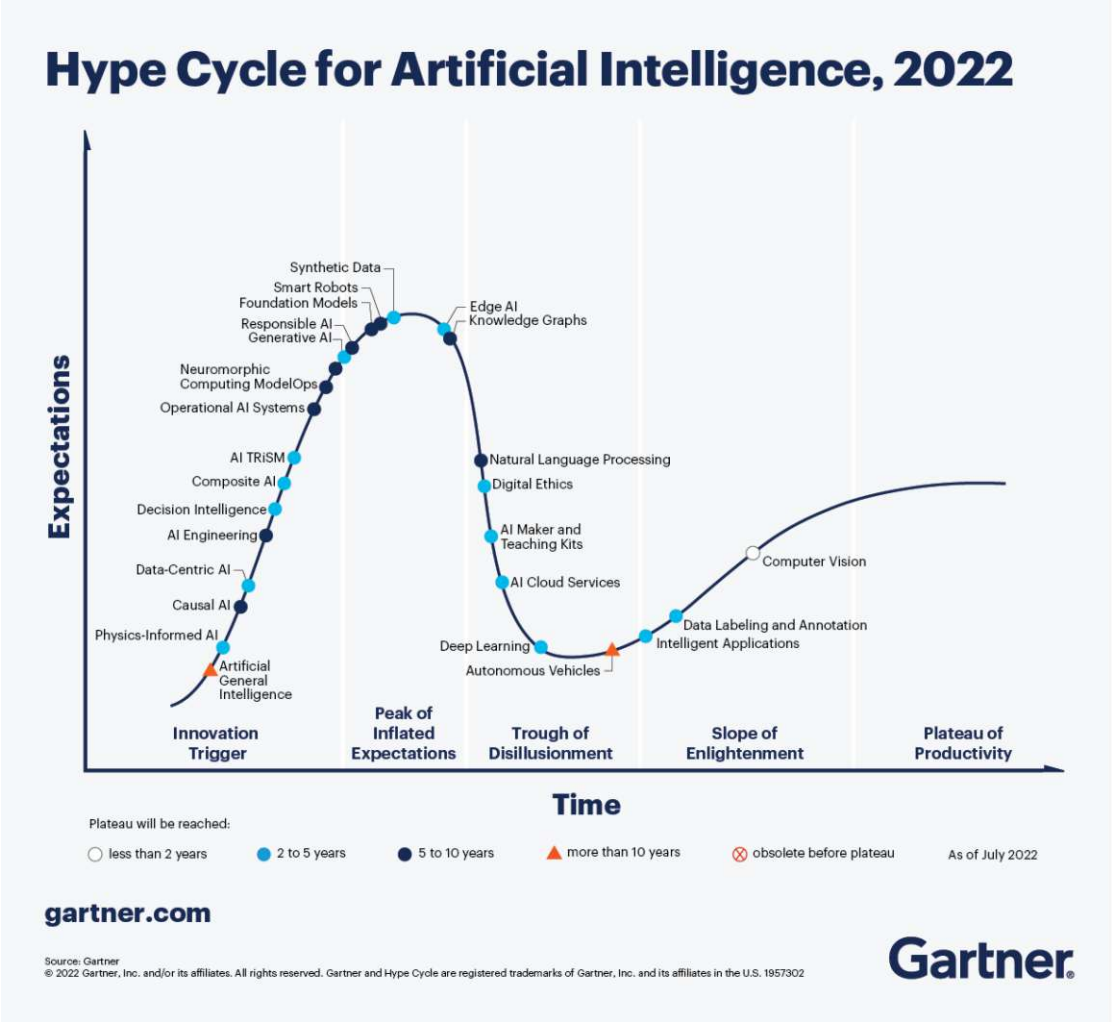


Figure 2.2: Gartner Hype Cycle for AI, 2022 [2]

Photos, or even temperature regulation like NEST learning thermostat. AI is ubiquitous in our lives and helps us do our jobs better, faster, and more accurately. As an example, speech recognition has already reached the plateau of productivity according to the Gartner Hype Cycle from 2019 [1]. Deep learning is a subcategory of ML and allows computational models to learn data representations with multiple levels of abstraction [26]. Deep learning models are composed of multiple processing layers which are connected.

Edge AI and Deep Learning will reach the plateau of productivity in two to five years, according to the Gartner Hype Cycle from 2022 [2]. Figure 2.2 shows the Gartner Hype Cycle from 2022 and describes which state a technology currently occupies. A typical cycle starts with innovation, gains then much attention, and years after that, the technology reaches the plateau of productivity where it is beneficial for our society. This hype cycle is only for technologies in connection to AI. It can be seen that Edge AI,

which will be discussed later, currently gets much attention and is predicted to reach in 2 to 5 years the plateau of productivity. On the other hand, artificial general intelligence currently gets less attention and has more than ten years of innovation ahead.

One type of artificial intelligence (AI) is deep learning (DL). It is composed of Artificial Neural Networks (ANN) and finds applications in various domains [40]. The primary structure of a DL network resembles the organization of the human brain, comprising interconnected layers. Each layer consists of neural nodes connected to nodes in the preceding and succeeding layers, as depicted in Figure 2.3. A simple form of such a network consists of three layers: 1) an input layer for data injection, 2) one or more hidden layers capable of hosting functions, and 3) an output layer for presenting the results. When data is received, the input layer forwards it to the hidden layers, where each node aggregates the inputs from preceding nodes using defined functions. The connections between nodes possess weights, and each node has a bias. Neural networks operate through two main steps: the training step, where the model is trained, and the inference step, where predictions are made using new data. Initially, the weights and biases in the deep neural network (DNN) are randomly set [40]. Consequently, if data is passed through the model, the output generated is random. To adjust the weights and biases to yield accurate outputs, the network requires extensive training data. The training data comprises input data paired with the corresponding correct results. This data is fed into the model, and the output is compared against the correct result to improve the model using the backpropagation algorithm [33].

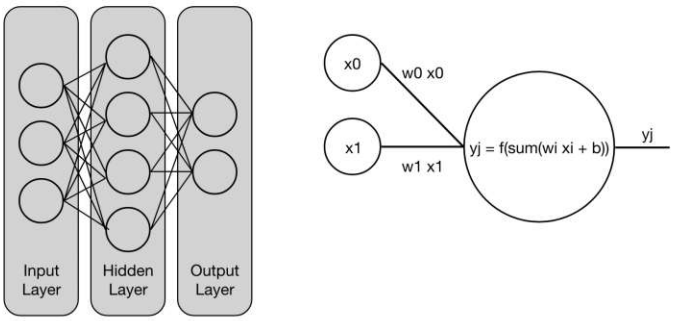


Figure 2.3: Principal composition of a NN model. Left: Layers in a model. Right: Neuron in more detail. [40, 29]

For instance, consider the task of handwriting recognition, specifically recognizing handwritten digits. In this case, a training set would consist of images of handwritten numbers along with their corresponding correct digits. The handwritten image is fed into the input layer, and the generated output is compared to the expected result from the training set. Through backpropagation, the weights and biases are adjusted iteratively, gradually improving the model's performance. Handwritten digit recognition serves as

a representative example and has received significant attention through datasets such as MNIST, which comprises 60,000 training images and 10,000 testing images [27]. By employing multilayer perceptrons (Multilayer Perceptron (MLP)), impressive error rates as low as 0.35% can be achieved [11]. However, while MLP is suitable for handwritten digit recognition, there exist numerous applications where other types of neural networks may be more effective. Over time, different types of neural networks have been developed, each specifically configured for its intended task. The following enumeration provides a basic introduction to the functionalities and use cases of a few distinct neural network types [21, 29].

1. Fully Connected Neural Network (FCNN). In an FCNN, each neuron, excluding the input and output layers, is connected to every neuron in the preceding and succeeding layers. Multilayer perceptrons (MLPs) represent a specific type of FCNN and serve as the fundamental form of DNN, comprising a minimum of three layers: input, hidden, and output. In Figure 2.3, the data flow is feed-forward, moving in a unidirectional manner from left to right, i.e., from input to output. MLPs find applications in tasks such as feature extraction and function approximation, yet they are characterized by complexity, suboptimal performance, and slow convergence rates [21].

2. Convolutional Neural Network (CNN). CNNs demonstrate high effectiveness in image processing tasks, making them particularly valuable in computer vision applications such as object detection. A CNN comprises three main layers: convolutional layers, pooling layers, and fully connected layers. Each layer performs a specific function within the network. The convolutional layers are responsible for convolving the input image using diverse kernels, producing multiple feature maps. This process is illustrated in Figure 2.4. Pooling layers play a role in reducing the number of network parameters by aggregating neighboring pixels. This is crucial since images typically consist of a large number of pixels, and without pooling, the subsequent fully connected layers would become excessively large. In the context of this work, a CNN is employed due to the objective of number detection in images [19].

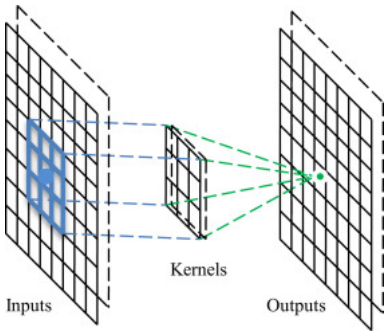


Figure 2.4: The operation of the convolutional layer. [19, 29]

3. Recurrent Neural Network (RNN). RNN find significant applications in human speech recognition tasks, such as with the home assistant Alexa developed by Amazon. RNNs are specifically designed for processing sequential data, making them well-suited for scenarios where the input length is variable, as is often the case with language processing. Within an RNN, each neuron possesses an internal memory that stores and utilizes previous samples during the training process. Backpropagation is employed for training the network, allowing for the adjustment of weights and biases based on error feedback [40].

A plethora of DNN architectures exist, catering to various use cases. Frameworks supporting multiple programming languages have been developed, simplifying the creation of new DNN models. The DNN community continues to expand, with ongoing advancements and innovations in this field. Furthermore, the implementation of these DNN models on ES paves the way for a new field known as edge intelligence. The integration of DNNs into ES opens up exciting possibilities for further exploration and development in this emerging domain.

2.4 Edge Intelligence

Edge Intelligence is a term used to describe performing intelligent tasks, such as analyzing data and making predictions, on devices located at the edge of a network. This contrasts traditional methods, which rely on sending data to the cloud or a centralized data center for processing. Edge intelligence allows for low latency and high-speed processing, as well as improved security and privacy since sensitive data does not need to be transmitted over the network. Applications of edge intelligence include autonomous vehicles, industrial automation, and smart cities. The field of edge intelligence is rapidly evolving, and new technologies such as edge computing, 5G networks, and machine learning enable a wide range of new and exciting applications. With the fast-growing data from the Internet of things, edge intelligence will be a key technology to analyze the data, make predictions and perform real-time decision-making. Deep learning has gained enormous attention in the last several years and has succeeded in various application domains, including natural language processing, computer vision, and extensive data analysis. Although to meet the computational requirements of deep learning, the cloud infrastructure is used. However, data must be moved from the edge to the cloud when using the cloud infrastructure, for example, from IoT sensors to a centralized location in the cloud. This solution, however, brings several challenges [9, 29].

- Latency. For some applications, real-time inference is critical. For example, remote surgery cannot suffer from long latencies. However, sending data to the cloud for inference or training cannot satisfy the stringent latency requirements due to network delays. For example, experiments with the Amazon Web Services server have shown that offloading a computer vision task takes more than 200 ms end-to-end [34].

- Scalability. Uploading all data from the edge to the cloud for inference introduces scalability issues, as the cloud can become a bottleneck as the number of connected devices increases.
- Privacy. The sensitive and private data generated at the edge is sent to the cloud with this approach which risks privacy concerns from the user who owns the data.

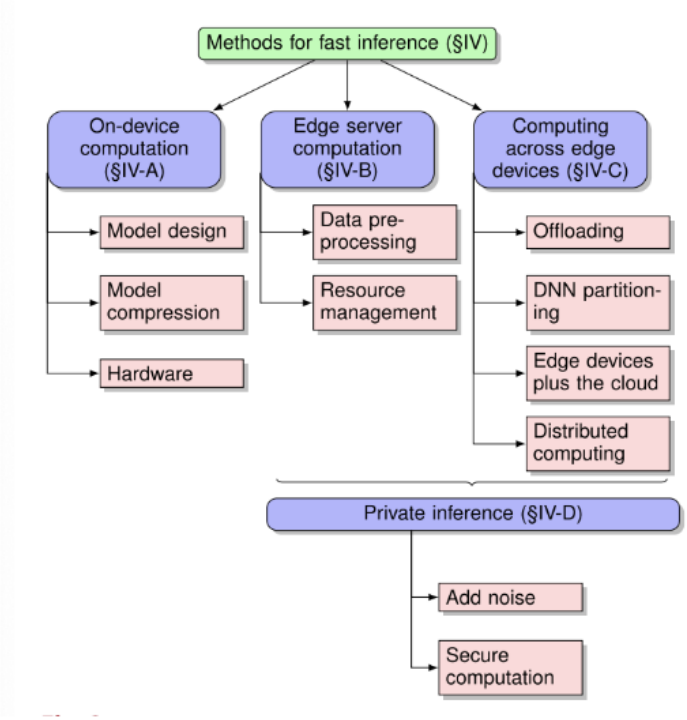


Figure 2.5: Different inference acceleration methods [9].

EC, particularly EI, is a promising solution to those challenges. Different architectures for performing quick inference are proposed, shown in Figure 2.5, to enable low-latency applications that use deep learning. There are three general approaches to on-device computation, edge server computation, and joint computation. They can be further partitioned into different approaches shown in Figure 2.5 [9].

- On-device computation. The first approach focuses on reducing the computation time on resource-constrained edge devices such as IoT devices or ES. Three major efforts in efficient hardware and DNN model design *i)* Model Design tries to reduce the computations in a NN to a minimum by designing the model lightweight; *ii)* Model Compression focuses on compressing the DNN models with minimal accuracy loss. Therefore DNNs could run on-device without the necessity of offloading. Parameter quantization, parameter pruning, and knowledge distillation

are a few popular model compression methods; *iii)* Hardware improvements are the third possible approach for running DNNs on the device. It improves the hardware at the edge by adding specialized hardware to the devices, such as DNN accelerators like Google's Coral for the TensorFlow framework.

- Edge server-based architectures. Sometimes DNN algorithms are too computationally intensive to run on the local device even with the before mentioned approaches. Therefore the algorithm has to be offloaded. Since the cloud is far away and unsuitable for real-time applications, the computation is offloaded to an ES in close proximity with enough resources. There are two approaches for fast inference with ES computation. *i)* Data preprocessing. Data preprocessing tries to reduce as much data as possible, yet enough to fulfill the tasks. Through this, the data transmitted to the ES is reduced, leading to faster decision-making. As an example, Glimpse [10] uses change detection filters on the camera frames. Therefore, only new images are uploaded for the processing which saves a lot of communication. *ii)* Edge Resource Management deals with the problem of efficiently managing multiple DNN tasks of different devices on shared compute resources. Some papers focus on the tradeoffs between accuracy and latency and other performance measures.

- Joint computation. Although ES accelerates the prediction process and sometimes even enables it, it is not always required to offload everything to the ES. Joint computation deals with fast inference through intelligent offloading tasks to the ES. *i)* binary offloading decides if a DNN is offloaded or not; *ii)* Partial offloading deals with partitioning the DNN and offloading only parts of it to the cloud. For example, the first part is computed locally, and the second part at the ES; *iii)* hierarchical architectures deals with offloading across a combination of edge servers, cloud, and edge devices; *vi)* Distributed computing describes the approach where the DNN computation is distributed across multiple peer devices.

In this thesis, we focus on the third approach, joint computation, and there on the DNN partitioning as a method for fast inference. However, the presented approaches can be merged and used together. As an example, a model could be compressed and then partitioned to gain an even faster inference.

2.5 Related Works

Table 2.1 summarizes the related works of this thesis. In this section, we will summarize related works and how they differ from ours. [9] provides an introduction with to EI and summarizes methods for fast inference at the edge. One of those methods is DNN partitioning, used in this work. The Neurosurgeon framework from Kang et al. [24] was one of the first to propose a dynamic prediction of the splitting point. Therefore, it uses prediction models for the different layer types such as Fully-connected Layer, Convolution & Local Layer, Pooling Layer, Activation Layer, and some others. Based on

	Kang et al. [24]	Lin et al. [28]	Gao et al. [17]	This work
Dynamic	✓		✓	✓
Energy Opt.	PREDICTED		PREDICTED	
Cost Opt.		✓	COULD-ONLY	✓
Latency Opt.	LAYER-BASED	TIME-CONSTRAINT	LAYER-BASED	LINEAR, CLOCK-SPEED
Mobile Devices		✓	✓	
Edge Nodes		✓	✓	✓
Cloud Nodes		✓		✓

Table 2.1: Comparison of Our Approach to Related Literature on DNN Splitting [30].

those layers Neurosurgeon predicts the latency in each layer and, together with the data size in each layer, predicts the best splitting point. However, it furthermore not only optimizes latency but also gives the possibility to optimize energy consumption. However, in comparison to our work, Neurosurgeon does not include any cost optimization. Lin et al. [28] propose a self-adaptive discrete particle swarm optimization (PSO) algorithm using genetic algorithm (GA) operators is proposed to minimize system costs in hybrid computing environments. The approach considers DNNs partitioning and layer off-loading across cloud, edge, and end devices. By avoiding premature convergence of PSO through the mutation and crossover operators of GA, the system cost is reduced by enhancing population diversity. Comparative analysis against benchmark solutions demonstrates the effectiveness of the proposed off-loading strategy in reducing system costs for DNN-based applications across cloud, edge, and end devices. This work includes a detailed cost model but does dynamically change the splitting point during run time nor include clock speed. Gao et al. [17] present a joint task partitioning and offloading design for a DNN-task-enabled mobile edge computing (MEC) network. The proposed approach considers the characteristics of DNNs and involves a single server and multiple mobile devices. Contributions include a layer-level computation partitioning strategy, a delay prediction model, a slot model with dynamic pricing, and a joint optimization framework. Two distributed algorithms based on aggregative game theory are proposed to solve the optimization problem. Numerical results demonstrate the scalability and superiority of the proposed scheme in terms of processing delay and energy consumption over baseline schemes. However, this work distinguishes itself from ours in the following points: i) It does not include a cloud node. ii) It uses layer-based prediction, and we use latency prediction based on clock speed. iii) It is a simulation; we present a test bed. iv) The cost optimization is cloud only.

CHAPTER 3

System Model

3.1 Edge Server

The ES is the central part of the proposed system. It hosts the scheduling algorithm, the prediction models, and the DNN, as shown in Figure 3.1. The ES is located at the network's edge, where the data is generated. It can be the User Equipment (UE) itself, as in voice assistance, or a server close to the UE, like a private hospital server that processes local data. This server, however, should not be more than about one hop away from the UE. The ESs often have limitations, such as energy, bandwidth, and computational power. This leads to the necessity of offloading some of the decision-making to the cloud. However, this depends on the resources of the ES itself. Depending on the location of the server, requests to the cloud can be fast or slow. If installed directly at a base station requests to the cloud could be very fast since the connection to the cloud is strong. Although, if installed in unaffiliated regions like in the environmental monitoring, requests to the cloud can be expensive. Furthermore, depending on the hardware resources of the ES, more or fewer tasks can be computed by the server itself. With respect to environmental monitoring, the ESs need to be small due to stringent energy limitations. Therefore hardware resources are limited and computation-intensive tasks need to be offloaded to the cloud. To operate ESs an organization or company has the option to buy and maintain themselves. This has the advantage of a one-time purchase and allows the organization running it, to be completely independent and flexible. Although, the maintenance of the server can be quite difficult. A great advantage is that this server can be placed directly where it is required, and provide therefore high-speed requests to it. Although the request from the ES to the cloud can, again, be costly due to the location. Another possibility is to rent ESs with very low latencies from a certain area. Therefore the maintenance is done by the provider and it can be used directly. Based on the requirements of the owner, it can be more expensive to rent than to own. In our work, we have two objectives that have to be optimized. *i)* Cost. ESs can be

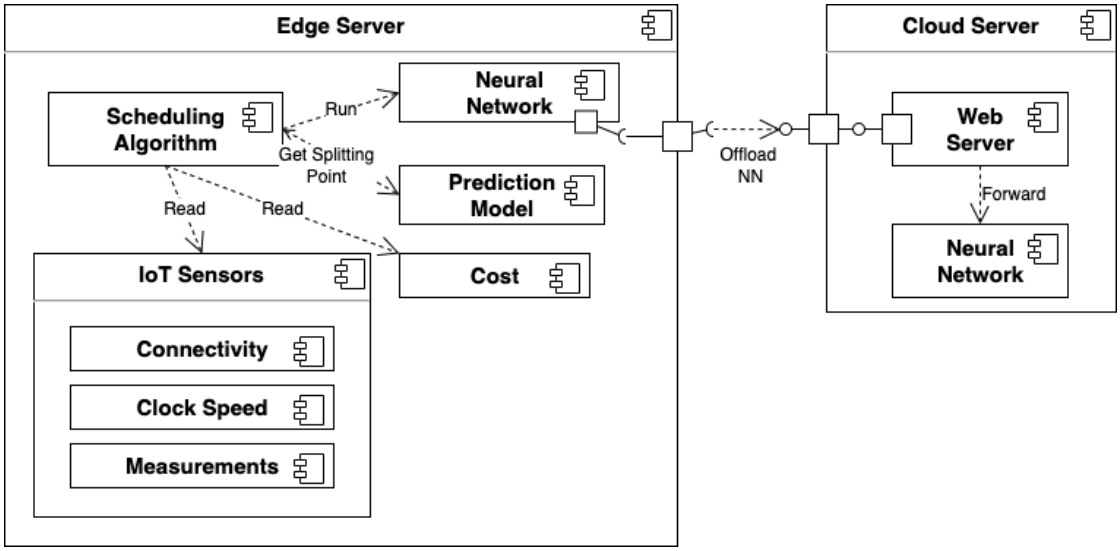


Figure 3.1: UML Component Diagram of the Proposed System [30].

rented or bought. When renting, ESs have a higher fee per hour than the cloud due to the optimized location of the ESs [5]. However, ESs can be owned by the application provided and used extensively at no further cost. In this case, computing locally on the ES can be cheaper than offloading to the cloud. *ii)* Latency. Low bandwidths and long latencies are the results of the utilized communication mediums due to the location of the servers. This directly affects communication latency. Those limitations also vary over time through environmental influences like the weather and flash crowds. Furthermore, the computational capability of an ES impacts the latency, represented as clock speed in our models.

Figure 3.1 holds five components, the *Scheduling Algorithm*, the *Neural Network*, the *Prediction Model*, the *IoT Sensors* and the *Cost Component*. The cost component is in our case a constant, delivering the prices for the edge server and cloud servers. In future work, this could be replaced with a component gathering live prices from the cloud providers. The *IoT Sensors* deliver the current Bandwidth, latency to the cloud, clock speed, and the actual measurements. The measurements are the real raw data to be investigated. The *Prediction Models* deliver the current optimal splitting point based on the live conditions of the IoT sensors and the cost to the scheduling algorithm. The component *Neural Network* holds the actual NN, prepared for splitting. In this work, we investigate three different networks. The *AlexNet* model, a CNN with 19 layers, the *VGG16* model, also CNN with 37 layers, and a model a CNN model for the MNIST data set with nine layers. When one of those NNs is called along with the splitting point, the network runs until the splitting point and offloads the rest to the cloud. The last component is the *Scheduling Algorithm* responsible for orchestrating all the components.

3.2 Stateless Cloud

The cloud data center consists of servers with virtually unlimited resources regarding energy, storage, and computational power. However, they are often geographically far from the UE, which can result in high response times. The time required for data to reach the cloud depends on the network bandwidth, latency, and data size. In most DNNs, data size varies among the layers and therefore the latencies vary. In this work, the cloud server is a web server with the above proposed NN models stored on it, shown in Figure 3.1. The web server has a REST interface with only one endpoint listening to an incoming request from an ES shown in Listing 2. To finish the NN at the CS, the splitting point, the tensor, and the model name are required. Furthermore, the model itself must already be stored on the server since only the tensor is offloaded and not the model itself. The web server then calls the corresponding model with the tensor and the splitting point. This can be seen in Figure 3.1, 3.2. All possible models have to be stored along with all layers since the splitting point is unknown. If the splitting point would be fixed, a possible efficiency increase could be to only store the required layers. After calling the NN on the CS the output holds the prediction. This could then be used to further cause actions, however, in our work, we stop at this point.

Because speed is crucial for a fast prediction, the cloud is implemented stateless. A stateless cloud is an approach for service-oriented architecture (SOA) for cloud computing environments. In traditional SOA, service providers maintain state information about their clients, which can lead to performance and scalability issues. With the implementation of a stateless cloud, providers do not maintain the state information of the users. This leads to better scalability and performance in cloud environments. This fits perfectly with our approach since multiple servers send a huge number of requests, and there is no need for storing client state information.

3.3 Communication

In general, ESs are in close proximity to where data is generated, enabling high-speed communication between UE and ES. On the other hand, the cloud is far away, so communication is slower. Nevertheless, ESs are often installed in environments where the connection to the cloud is also fast. However, ESs have to be installed in rough and rural areas, like in environmental use cases or in autonomous vehicles. Especially under rough conditions, efficiency is crucial for fast decision-making. Since the data size significantly impacts communication latency, scheduling the splitting point to a layer where the data size is small might have a huge impact. Therefore, our goal is to reduce bandwidth use and latency. Figure 3.2 illustrates the complete communication sequence from the data generation to the predicted result in the cloud. The scheduling algorithm orchestrates everything and holds a timer for triggering model calls. Based on the requirements of the application, the predictions can be made in a certain time interval or based on a threshold value of the raw data. This does not matter to our proposed rescheduling algorithm and would just be an additional component. The same

holds for the splitting point prediction. It can be eighter time triggered, e.g. every two minutes the best splitting point is predicted, or based on a threshold. This could be triggered e.g. when the bandwidth or clock speed changed drastically. Once triggered, the scheduling algorithm calls the prediction models with the current bandwidth, latency, clock speed, and costs. Furthermore, a variable deciding either to optimize more cost or latency has to be provided. The prediction models then return a predicted splitting point on those life parameters. When a prediction is triggered, the DNN model is called with the raw data and the splitting point. After finishing until the splitting point the tensor is base64 encoded and sent to the web server via HTTP. For secure transmission, HTTPS would be recommended, however, this is not implemented in this thesis. When the DNN has reached the splitting point, the current tensor, intended as input for the next layer is base64 encoded and sent via HTTP as JSON to the web server. An exemplary request can be seen in Figure 3.2 between *Neural Network* and *Webserver Cloud*. A response is not explicitly required, and therefore not drawn, however, it might be useful and makes sense since in the case of a failure the data can be stored at the edge.

3.4 Latency Measurement

In this work, we consider three latency types as highlighted in Figure 3.2 with the red boxes: *i)* edge latency is the time required by the edge to run the first part of the DNN. *ii)* communication latency is the time required to send the data from the edge to the cloud *iii)* Cloud latency is the time to finish the execution of the second part. The ES receives the input data, runs it at the edge for a given amount of layers, and then offloads the data to the cloud, where the DNN inference is finalized. The time spent until offloading on the ES is called edge latency and depends on the computation power of the ES. The overall end-to-end latency is the time between calling the DNN at the edge and the predicted output. The time taken to finish the NN by the cloud is called

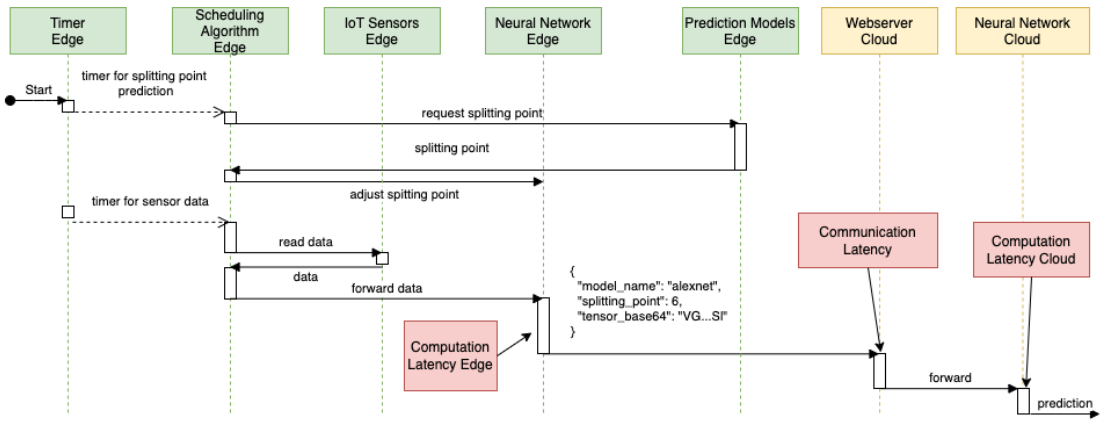


Figure 3.2: UML Sequence Diagram of the Proposed Cost-Aware Dynamic Scheduler [30].

cloud latency. Finally, the time consumed between offloading and receiving is called communication latency. The workflow of the latency (Lat.) measurement in more detail

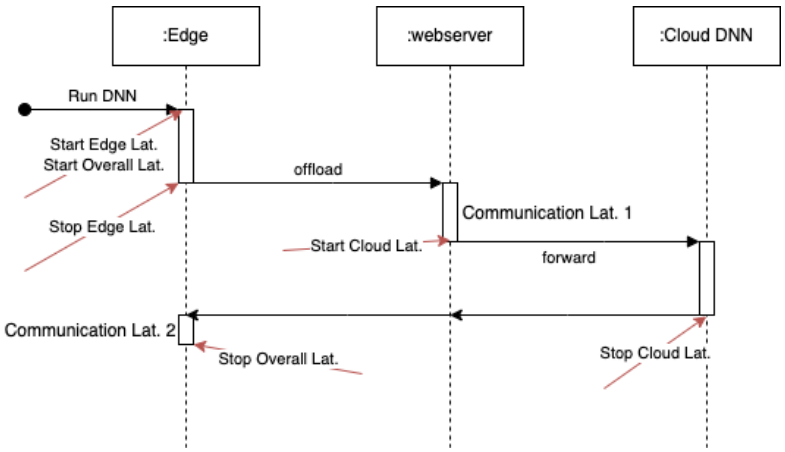


Figure 3.3: UML Sequence Diagram of the latency measurement.

is shown in Figure 3.3. When the DNN at the edge is called, we store that time as the overall starting point and at the same time, the starting point of the edge latency is called *overall_lat_start*. During the run of the DNN, we store the latency of every layer for later evaluation. When the network at the edge reached its splitting point, and the tensor is offloaded, right before sending the data, the edge latency stop time is stored. We call it *edge_lat_stop*. Through this, we can calculate the time the DNN took to run at the edge. When the offloaded tensor reaches the web server, this time is stored as *cloud_lat_start*, since from here the cloud is running the DNN. We also store the time in each layer of the cloud model. When the model is finished, we store this time as *cloud_lat_stop* and send the two cloud latencies back to the edge. This is for reasons of simplicity to measure. At the edge of the last latency, the overall stop latency is stored as *overall_lat_stop*. Therefore the three different latencies shown in Figure 3.2 can be calculated as followed: The latency at the edge is the subtraction of the time before offloading and the overall starting point shown in Equation 3.1.

$$edge\_lat = edge\_lat\_stop - overall\_lat\_start \tag{3.1}$$

The computation latency of the cloud is simply the subtraction of the start time from the stop time, shown in Equation 3.2.

$$cloud\_lat = cloud\_lat\_stop - cloud\_lat\_start \tag{3.2}$$

The last latency, the communication latency, is slightly more complicated to measure. Due to the time differences between different devices, the measurements cannot be compared directly. Therefore we calculate we subtract the edge latency and the cloud latency from the overall latency. However, there is a failure because the response communication latency should be excluded. This can be achieved by subtracting the latency taken from

the cloud to the edge, which can be measured, along with the duration of the transmission which can be calculated with the transmission size and the bandwidth. However since the response data is really small, the response communication latency is it as well. The resulting formula is presented in Equation 3.3.

$$comm\_lat = overall\_lat - cloud\_lat - edge\_lat - \frac{ping\_lat}{2} - \frac{response\_size}{bandwidth} \tag{3.3}$$

$$overall\_lat = overall\_lat\_stop - overall\_lat\_start \tag{3.4}$$

CHAPTER 4

Prediction Models

We focus on two main objectives: cost and latency. Our goal is to predict the optimal splitting point for minimizing the cost of the servers and the latency based on real-time parameters. We investigate two different strategies i) the application provider owns the ES; therefore, does not have to pay rent for the ES, and ii) they rent the ES as well as the cloud. In the first scenario, it is cheaper to compute at the edge, whereas, in the second, it is cheaper to offload since the ESs are more costly [15]. To find the optimal splitting point, three prediction models are created. The first predicts the influence of a change in the clock speed on the computation time, the second predicts the influence of the bandwidth and latency change on the communication latency, and the third predicts the cost of a task. Combining these models, a dynamic splitting point scheduler that simultaneously minimizes latency and reduces cost is proposed. Substantial overlap exists between the content presented in this chapter and our previously published workshop paper [30].

4.1 Edge Computation Latency

Deploying ESs in unaffiliated environments with limited energy supply and relying on batteries and solar power results in fluctuations and scarcity in computational resources. Therefore a variety of edge nodes with different clock speed settings coexist. For this reason, we included clock speed in our prediction models to predict the edge computation latency.

The first step is the prediction of the latency in a specific layer independent of the clock speed. Therefore we measure the final layer latency and assume that the edge latency has a linear behavior. Since the edge latency is zero before the first layer, we can create a linear equation. With this equation, the predicted latency in a specific layer can be calculated, albeit only for this specific server, under the specific setting and the specific DNN. The next step is to adapt the prediction model so that the calculation of the edge

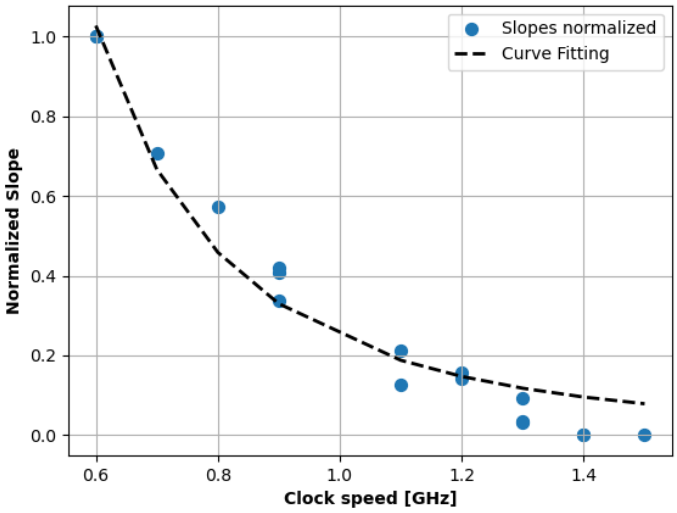


Figure 4.1: Normalized Slopes with Curve Fitting [30].

	Instance Name	Hourly Rate	vCPU
Edge Server	t3.xlarge	$0.224	4
Cloud Server	t3.xlarge	$0.1664	4

	Memory	Storage	Type	Location
Edge Server	16 GiB	EBS Only	Wavelength Zone	US East (Verizon) - NY
Cloud Server	16 GiB	EBS Only	Region	US East (Ohio)

Table 4.1: Reference Configurations and On-Demand Prices for AWS EC2 [5, 30]

latency takes the clock speed into account. To that end, we investigate the relationship between the latency and the clock speed on the specific server by running the DNN on the server. We benchmark the execution time under various clock speed settings and compute the latency slope, which estimates the latency in this layer when multiplied by the layer number. Although, only the measurement for all-in-edge, meaning nothing is offloaded, is required since only this value is required to calculate the slope of the edge latency. Because each DNN has completely different layer latencies due to the different numbers and types of layers, the data points cannot be combined directly and must be normalized to the range $[0, 1]$. Figure 4.1 illustrates the plot of all data points of the different DNNs. After normalizing the data points, Power Law can be used to generate a fitting curve through all measurements. The resulting model to predict the edge latency has to be de-normalized before being used for a specific DNN. This can only be done with the upper and lower bounds. This is the reason why two measurements are required for a new DNN. The all-in-edge latency of the highest and the all-in-edge latency of the lowest clock speed. With those two bounds, the latencies for the other clock speed

settings can be predicted. The resulting prediction function for the latency based on the clock speed is given in Equations 4.1 and 4.2.

$$f_{edge}(CS, SP, M) = f_{denorm}(a * CS^b, up\_b(M), l\_b(M)) * SP \tag{4.1}$$

$$f_{denorm} = slope_{norm} * (up\_b(M) - l\_b(M)) + l\_b(M) \tag{4.2}$$

$a,\ b$	=	Parameters from the curve fitting
CS	=	Clock speed
up_b	=	NN specific upper bound
l_b	=	NN specific lower bound
M	=	NN model
SP	=	Splitting Point

The prediction of the cloud computation latency works the same way as the prediction for the edge computation latency but without the dependency on the clock speed. This is because the cloud runs on virtual CPUs with a fixed clock speed. Therefore the computation latency of the cloud can be measured once with the all-in-cloud strategy. With this measurement and the knowledge that with the all-in-edge strategy the computation latency in the cloud is zero, a linear equation can be created, with which the latency in each layer can be determined.

4.2 Cloud Computation Latency

Since the cloud is running on an AWS instance and only virtual hardware is provided, the CPUs are also virtual (vCPU). Therefore we do not have a change in clock speed in the cloud and therefore no necessity to include the clock speed in our model. Figure 4.2 shows some measurements of the cloud latency of the Mnist model. It can be seen that, except for two outliers, all the measurements lie very close. Due to this fact, and that we try to keep the expenditure to add new models as low as possible, we measure the all-in-cloud strategy. With the all-in-cloud measurement, meaning the raw data is offloaded we get the first measure point. To then get the slope we assume the all-in-edge measurement is zero. We are then able to predict the cloud latency, shown in gray in Figure 4.2.

4.3 Communication Latency

ESs are usually installed in close proximity to the data generation to achieve low latencies. However, when the ES sends data to the cloud, the communication latency can be high, depending on where the ES and cloud data center are located. In this work, we assume the ES is in a region with volatile internet connectivity. This can result from an external condition such as weather or increased internet traffic. Furthermore, intermittent connectivity can also appear when the ES is mobile, such as a car driving through regions

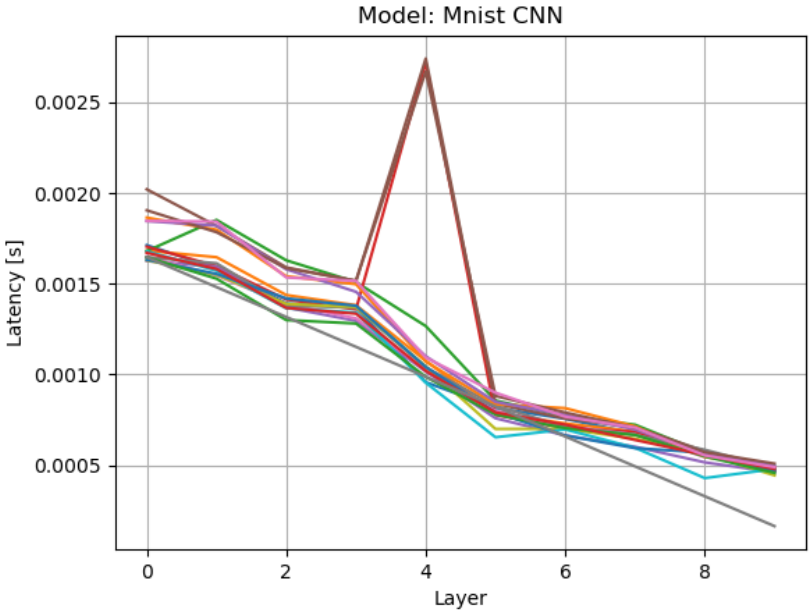


Figure 4.2: Cloud Latency measurements for Mnist model.

with good or bad connectivity. Therefore, the connectivity must be considered when offloading a DNN since the data size can be massive. Therefore, the splitting point needs to be dynamically recalculated. To achieve this, we create a prediction model for the splitting point based on the current bandwidth and latency. Since we know the current bandwidth and latency and the data size in each layer, the communication latency can be estimated. Equation 4.3 takes the bandwidth, latency, and splitting point as input and calculates the communication latency.

$$L_{comm}(BW, PL, SP, OS) = \frac{OS(SP) * 8}{1000 * BW} + \frac{PL}{2 * 1000} \tag{4.3}$$

OS = Array of layers with offload sizes [Byte]
BW = Bandwidth [kbit/s]
PL = Ping Latency [ms]

Equation 4.3 shows that the communication latency depends on the bandwidth, ping latency and the splitting point. The offload size of the offloaded data is fixed for a DNN in each layer and can be gained through the splitting point. If a new NN is added to the scheduling algorithm, the sizes in each layer must be stored.

With all the predicted latencies, edge, cloud, and communication, we can calculate the overall latency through the sum of all. This predicted overall latency is then used to find the optimal splitting point in terms of latency. Figure 4.3 shows the different predicted

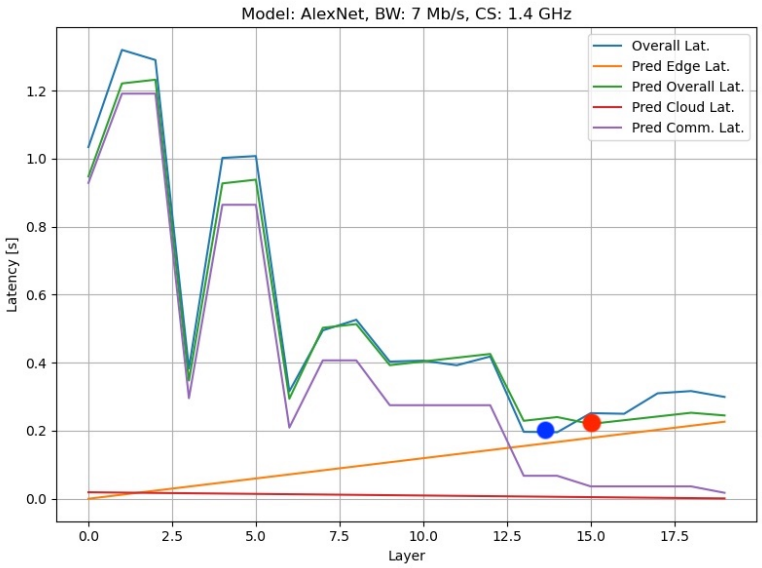


Figure 4.3: Predicted Latencies for the AlexNet model [30].

latencies, cloud in red, edge in orange, communication in purple and the overall in green. It can be seen that the edge latency and the cloud latency are now linear. This results from our prediction model for edge and cloud. The blue line represents the actual overall latency and it should correspond with the predicted overall latency in green. If we would just investigate the best prediction model in terms of latency, we would be finished here. We could take the minimum layer (the red dot) and would split it there. However, we also add costs to our rescheduling algorithm, which is discussed in the next section. It has to be remarked that the blue dot shows the actual best splitting point, which is slightly different from our prediction. This is the error therm in our model.

4.4 Resource Cost

To predict the costs arising during the computation of a task, we use the two strategies presented at the beginning of this section, namely owned and rented ES. The two strategies are given in Equation 4.4. In the first scenario, the ES is owned; therefore, we do not calculate any costs for running tasks at the edge. In this case, it, therefore, makes more sense to compute as much as possible at the edge to save costs. In the second scenario, we rent the ES as well as the cloud and calculate the cost based on the time running at the edge and the time running in the cloud. We use real-world AWS prices for our calculation. AWS offers ESs with ultra-low latencies for 5G services at a higher price than the cloud resources, as shown in Table 4.1 [5]. We choose the corresponding price for the partitions that run on edge and cloud.

$$f_{cost}(RT) = \begin{cases} 0 + C_{cloud}(RT_{cloud}), & \text{if ES is owned.} \\ C_{edge}(RT_{edge}) + C_{cloud}(RT), & \text{if ES is rent.} \end{cases} \tag{4.4}$$

4.5 Combined Prediction Model

To combine the prediction models to optimize latency as well as cost based on clock speed, bandwidth, latency, and cost, an optimization function is presented in Algorithm 1. The optimization function can be weighted to optimize lower latency or cost. The weight parameter w can take values from 0 to 1, with 0 to optimize only the latency and 1 to optimize only the cost. First, the cost and the latency are calculated for each layer, assuming it is the split point. The overall latency, including edge computation, communication, and cloud computation, and overall cost, including edge and cloud, are considered. The results are stored in two arrays and are sorted in decreasing order of preference. Then based on the weight variable w, one array entry is compared with the corresponding other table entry. Each item in one array is compared to $|entries|*((w-0.5)*2)$ items in the other array. For example, if w is 0.5, precisely one new entry of cost and latency are compared, along with the ones already compared. Although, if a weight is added, not just one entry is taken into account, but $percentage(w)*|entries|$. If w is 0.6 one entry of cost is compared with $|entries| * ((0.6 - 0.5) * 2) + 1$ of latency entries along with the cost and latency entries before.

RT = Runtime

Algorithm 1 Weighted cost latency optimization algorithm [30].

Require: w
Require: $L = [pred\_overall\_lat]$
Require: $C = [pred\_best\_cost]$
sort[C] ▷ Cheapest first
sort[L] ▷ Fastest first
$visited\_cost = []$
$visited\_lat = []$
for `i in range(0, layer_length)` **do**
$visited\_cost \leftarrow add\_visited(w, C, i)$
$visited\_lat \leftarrow add\_visited(w, L, i)$
if `visited_cost in visited_lat ||`
`visited_lat in visited_cost` **then** ▷ Check if a match
return `match`
end if
end for

To explain the algorithm more visually we created Figure 4.4. It shows how the algorithm works with a *weight* of 0.55 and 20 layers, meaning, we slightly favour cost. First we create two arrays which holds the best layers for eighter cost or latency. We sort them

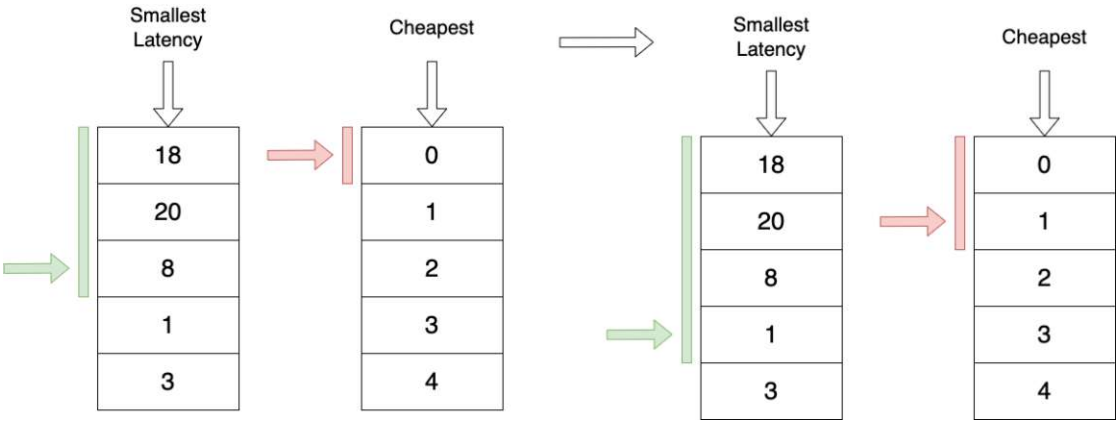


Figure 4.4: Cost, Latency trade-off Algorithm with $weight = 0.55$.

both ascending meaning the fastest layer first and the cheapest layer first. Then, since we have a *weight* of 0.55 we calculate the advantage of the cheapest array. The *weight* gives us 10% advantage ($(0.55 - 0.5) \times 2 \times 100\% = 10\%$) which results in a two layer advantage ($20 \times 10\% = 2$). We then iterate through the arrays and comparing them (with an advantage of cheapest of 2) if there is a match. We also take into account the already compared ones. In the second step it can be seen, that a match has been found with layer 1. Through this, the first match of the two weighted arrays can be found. Figure 4.5 shows the different latencies in our experiment (overall, cloud, and edge latency). As well as the predicted edge latency in red and the predicted overall latency in violet. It can be seen, by increasing the clock speed, the resulting edge latency as well as the predicted edge latency decrease. This results in a different optimal splitting point regarding latency. Furthermore, Figure 4.5 shows three different dots, representing the best splitting point regarding cost, the best splitting point, regarding latency, and the best splitting point if cost and latency are evenly weighted. In this scenario, we rent the ES which makes it cheaper to offload directly making it cheaper the sooner to offload. The red dot represents the best optimal splitting point regarding latency. It reschedules dynamically if clock speed, ping latency, or bandwidth changes. It can be seen that e.g. in the plot's right bottom corner, with clock speed 1.5GHz the best-predicted splitting point is layer 15 but it actually would be layer 13. This is because the optimal latency splitting point is predicted and therefore holds some failure. The blue dot represents the combination of latency and cost when both have the same weight.

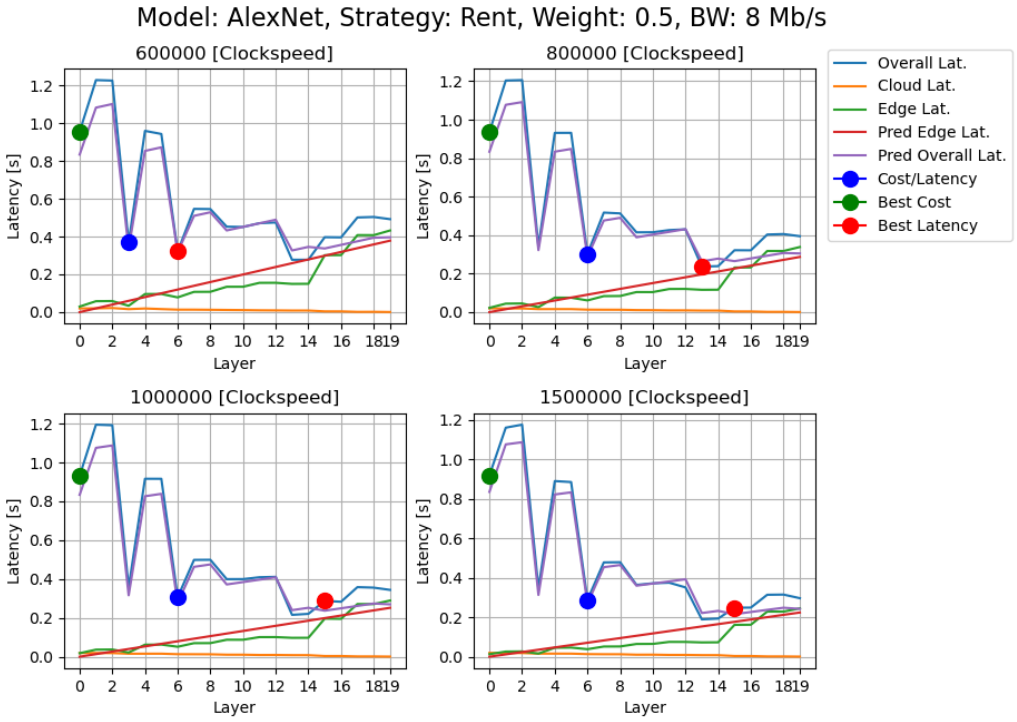


Figure 4.5: Dynamic Rescheduling of Splitting Points Under Different Clock Speeds [30].

CHAPTER 5

Neural Network Splitting

The majority of the content presented in this chapter exhibits substantial similarities with our previously proposed workshop paper [30].

5.1 Splitting

In general, splitting a network requires a point at which the neural network is partitioned and split into two parts. The first part runs at the first server, or the edge, and the second part in the cloud. The point where the network is partitioned is called the splitting point and is dynamically calculated in this work. Therefore the network must be able to offload the second part of the network after each layer and before the first layer (this would be the offloading of the raw data). However, the existing networks are usually not built for splitting; therefore, the network must be prepared. This is a manual process in which particular functions are implemented into an existing NN. Although, it is essential to preserve all the existing layers and functions and ensure the prediction outcome stays precisely the same. The following parts must be considered to make a network separable. However, some can be reused for all networks, and some must be done specifically for the network.

- **Inspect network:** The first step is to find out how many and types of layers are used in a specific network. This can be done by manually searching through the code or, if implemented, printing out a summary of the network. Depending on the configuration, networks can consist of hundreds of layers, often created during runtime. Furthermore, the layers themselves can contain again layers that must be considered for splitting. If a network uses a layer that consists of multiple layers, this work splits the layers of the layers until the elementary layers are reached. Such elementary layers are the basic modules of the PyTorch framework found in

the "/nn/modules" directory, such as Linear, Conv2d, MaxPool2d, ReLu and many more.

- **Gather layers:** In the next step, all layers are combined into an array that can be iterated. Furthermore, the layers are made properties of a class "Layer" to add additional properties to a specific layer, such as id (the id is equal to the layer's position in the network) or start and stop time for measurement purposes.

- **Split the network at the edge:** When a network is called, the forward method is entered, and the network calls its layers. The content of this forward method is completely deleted and replaced with a loop that iterates over all layers shown in Algorithm 2. As long as the $splitting_point$ is greater than the layer id, the layer should normally run at the edge. However, when the $splitting_point$ is equal to the layer id, meaning that the network shall be partitioned after this layer, the network runs this layer and offloads the hole tensor to the cloud via a REST request.

Algorithm 2 Iterate and Execute Layers at the Edge

```
Require: splitting_point, x (raw data as tensor), layers (dictionary of layers)
  for k, v in layers do
    if splitting_point >= k then
      x = v.layer(x)
      if splitting_point == k then
        data = offload(splitting_point, x)
      end if
    end if
  end for
  return x                                   ▷ Tensor at splitting point
```

- **Offloading:** The tensor, which has to be offloaded, is converted into a base 64 encoded string. This string is added to the body of a POST request to the cloud. Furthermore, the splitting point has to be sent along to continue at the right point in the cloud. If multiple models are used with the same cloud server, the model must also be specified and sent along. The base 64 encoded tensor is when received by the cloud decoded and transformed back into a tensor, which can be directly used to call the model.

- **Continue running in the cloud:** The network model running in the cloud is the same as at the edge, however, with a slightly different forward method. In the cloud, the network is already partitioned and it just needs to finish it from a given splitting point. The algorithm for finishing the network in the cloud is shown in Algorithm 3. The line 3 shows that nothing has been done as long as the *splitting_point* is smaller than the layer id. Although, after the *splitting_point*, the network runs normally, and all layers are called.

Algorithm 3 Iterate and Execute Layers in the Cloud

Require: splitting_point, x (tensor from edge server), layers (dict of layers)

```
for k, v in layers do
    if splitting_point >= k then
        continue
    end if
    x = v.layer(x)
end for
return x                                              ▷ Final tensor
```

5.2 Neural Networks

5.2.1 AlexNet

The AlexNet model is a convolutional neural network designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It was created to participate in the ImageNet Large Scale Visual Recognition Challenge and outperformed the previously leading models in 2012. The ImageNet dataset, used for training, is a comprehensive image database developed by Standford University researchers in the early 2000s. It includes over 14 million images, organized into 20,000 categories, representing a wide range of objects, scenes, and animals. The original AlexNet network contained eight layers. First, five convolutional layers, followed by three fully connected layers. Although, in the following years, the model was improved. For this thesis, we use an implementation of the PyTorch community [13], which again uses an improved version of AlexNet created by Alex Krizhevsky in 2014 [25].
Input. The input for the AlexNet is a random image with dimension 4625×6938 and

Figure 5.1: Input image for our Alexnet model.

resolution 72×72. The image used in this work is shown in Figure 5.1. The image stays

#	Layer Type
1	`Conv2d(3, 64, kernel_size=11, stride=4, padding=2)`
2	`ReLU()`
3	`MaxPool2d(kernel_size=3, stride=2)`
4	`Conv2d(64, 192, kernel_size=5, padding=2)`
5	`ReLU()`
...	
16	`ReLU()`
17	`Linear(4096, 4096)`
18	`ReLU()`
19	`Linear(4096, num_classes)`

Table 5.1: Layers of the used AlexNet Model.

for all measurements the same to ensure the data and, therefore, the size always stays the same. The figure is resized to 224 x 224 pixels and normalized before being transformed to a tensor. **Layers.** The network implemented in Pytorch from the PyTorch community consists of 21 Layers; however, two are Dropout layers which are only relevant for training. Table 5.1 shows a list of the used layers along with their ids and features.

5.2.2 Mnist

The modified National Institute of Standards and Technology database (MNIST) is a widely used dataset of handwritten digits that serves as a benchmark for image classification tasks in computer vision. The database contains 60,000 training images and 10,000 testing images, each with a grayscale 28×28 pixel image of a handwritten digit ranging from 0 to 9. The MNIST dataset has been used to evaluate and train a range of machine learning models, including linear classifiers, support vector machines, and convolutional neural networks. Its small size and ease of use make it an ideal starting point for exploring new developments in computer vision. Overall, MNIST remains a standard benchmark dataset for image classification tasks in computer vision and has been instrumental in the development of new algorithms and techniques in the field. It was developed by Lecun et. al. out of the NIST dataset and was specially designed to train CNNs [27]. We use images out of this dataset for inference and use a CNN implementation from a paper that achieved up to 99.91% accuracy on this dataset [4]. We use the version with 5×5 kernel size in convolution layers. Table 5.2 shows the nine layers used for this model.

5.2.3 VGG

The Visual Geometry Group (VGG) model is a deep convolutional neural network that was introduced in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" by Karen Simonyan and Andrew Zisserman, which was published

#	Layer Type
1	`Conv2d(1, 10, kernel_size=5)`
2	`MaxPool2d(kernel_size=2)`
3	`ReLU()`
4	`Conv2d(10, 20, kernel_size=5)`
5	`MaxPool2d(kernel_size=2)`
6	`Linear(320, 50)`
7	`ReLU()`
8	`Linear(50, 10)`
9	`Softmax(dim=1)`

Table 5.2: Layers of the CNN for the MNIST Dataset.

in the Proceedings of the International Conference on Learning Representations (ICLR) in 2015. This paper investigates the impact of increasing the depth of convolutional neural networks on their accuracy in large-scale image recognition. The authors evaluate networks with small convolutional filters and show that significant improvements over prior state-of-the-art models can be achieved by increasing the depth to 16-19 weight layers. The resulting models achieved first and second place in the 2014 ImageNet Challenge for localization and classification respectively. The authors also demonstrate that their models generalize well to other datasets and have made their best-performing models publicly available for further research [36].

In this thesis, we use a PyTorch implementation of VGG16 by the Pytorch community [14]. This model is built out of 37 layers and is therefore the biggest model in our tests regarding the layer count. The input for our model is the same image as for the AlexNet model shown in Figure 5.1. However before the model is called with this image, it is resized to dimension 256×256 and transformed into a tensor to fit the model.

5.3 Limitations

The first DNN we prepared for splitting was a Graph Neural Network (GNN) since a GNN fitted best for the use cases of the SWAIN project. This results from the fact, that a river with measurement stations can be mapped into a directed graph with the measurement stations as nodes and the river as edges. The idea was, to gather the data at the edges, run a part of the GNN at the edge, and offload the rest to the cloud. However, this has proven not feasible since the measurement stations themselves are not connected which stops the flow of the GNN. Therefore it only works if all the raw data is offloaded to the cloud and there the GNN is called. Although, this means no acceleration through partitioning. Future work could investigate if a connection between edge servers could enable the partitioning of GNNs. If however, all data is already available at the edge in case a local problem can be mapped to a graph, e.g. a smart home, our splitting approach can be used. Further limitations of our approach are DNNs with non-chain

#	Layer Type
1	`Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))`
2	`ReLU()`
3	`Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))`
4	`ReLU()`
5	`MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1)`
6	`Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))`
7	`ReLU()`
8	`Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))`
...	
30	`ReLU()`
31	`MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1)`
32	`AdaptiveAvgPool2d((7, 7))`
33	`Linear(512 * 7 * 7, 4096)`
34	`ReLU(True)`
35	`Linear(4096, 4096)`
36	`ReLU(True)`
37	`Linear(4096, num_classes)`

Table 5.3: VGG16 Model with 37 layers.

structures. In such DNNs layers can run in parallel which makes it hard to offload and predict splitting points. However, this might be possible to overcome in future work and would be very interesting to investigate.

CHAPTER 6

Experimental Setup

6.1 SWAIN Project

This thesis arises from a project called Sustainable Watershed Management Through IoT-Driven Artificial Intelligence (SWAIN). The project aims to address the significant threat of water resource contamination to the environment. The team proposes a solution that focuses on the rapid identification of chemicals and their emission sources in watersheds to ensure sustainable water resources management. Despite several studies on micropollutant measurements in water resources across Europe, the efficient utilization of collected data for informed decision-making to protect water resources from harmful chemical pollution is currently lacking. To address this challenge, the project team is exploring the potential of advanced Artificial Intelligence (AI) strategies and Internet of Things (IoT) technologies, which offer faster and more efficient responses in both real-time reactions and long-term planning.

The proposed solution is designed to improve understanding and provide near real-time response to pollution incidents, predict pollution spread, and mitigate its long-term effects. The project relies on data-driven AI lifelong learning and the evolution of the algorithm to achieve its primary objective, which is to develop an integrated decision support system. This system will utilize micropollutant measurements, along with real-time hydrodynamic and meteorological data of a watershed, to manage water quality sustainably. Since micropollutants are resistant to degradation and are linked to emission sources, they provide an accurate indication of pollution and its sources.

The project team's approach involves introducing and combining novel technologies to improve water pollution management in various critical phases. They rely on an advanced, scalable IoT technology that adapts to specific problem requirements through a novel mechanism called dynamic rescheduling. This approach allows the team to obtain the desired data from optimal locations and times for further data analysis. The team also

introduces a novel methodology to create a more accurate hybrid model that integrates expert-based physical environment models and data-driven evidence-based techniques. To achieve this, they introduce a novel graph-based functional representation of data that captures affinities and dependencies among data streams more efficiently. This thesis focuses on the dynamic rescheduling part of the proposed project to ensure fast prediction-making in rough environments. This can be due to limited energy resources, unstable internet connection, long latencies, or limited computational resources.

6.2 Use Case

The use case of this proposed cost-aware scheduling algorithm is for ESs installed in unaffiliated areas with low internet connection. Especially under such conditions, efficiency is very important. To accelerate inference we measure current network conditions and clock speed and provide a latency-optimized splitting point. Furthermore, if multiple ESs are required, similar to the SWAIN project, the price can also be important. Therefore our approach considers if the user rents or owns the ES and integrates it into the proposed splitting point. Some use cases might require fast predictions, e.g. if an autonomous vehicle is driving in a region with low connection, prediction is still required fast. In this case, the cost might not be that important, and more weight can be added to the latency. On the other hand, for the SWAIN project, the price is highly relevant, and therefore the weight can be adjusted more in the direction of cost. To sum up, our proposed rescheduling algorithm accelerates the inference of a DNN on edge devices that are installed in regions where resources are limited or unstable. In the case of autonomous driving, if a vehicle drives back into a well-connected region, our algorithm adapts automatically to the new circumstances.

6.3 Design Science as Methodological Approach [22]

Design Science (DS) has been chosen as the methodological approach for this thesis. Figure 6.1 shows the structure of DS in information systems (IS) and how it is used. On the left side, we see the *Environment* in which the artifact is built. We have people, organizations, and technologies that create an environment the artifact must fit. From there, the business need for the artifact arises and should be continuous input during construction. On the opposite side, we can see the *Knowledge Base*, which consists of foundations (theories, frameworks, instruments..) and methodologies (data analysis techniques, formalisms, Measures..) to create and validate the artifact or the theories. The *IS Research* is occupied between the environment and the knowledge base. This describes developing/building and justifying/evaluating the artifact. This loop of building and validation requires the knowledgebase's tools and ensures the artifact fits the business needs. Furthermore, [22] describes seven guidelines to be followed when using the DS approach for ISs. The DS approach is applied to this work in the following three sections.

6.3.1 Environment

The object of interest and the problem space is occupied in the environment. Everything the artifact faces on the business side, like management, business decisions, used/planned technologies, or people, are stored in the environment. This would be the area or the use case in which the proposed rescheduling algorithm is installed. The original intention of our work is environmental monitoring, where DNNs run for days, months, or even years. Furthermore, infrastructure is suffering from long latencies and low energy supply. Due to that, the algorithm is specifically designed for such unaffiliated regions. The design science environment faces the used technology like infrastructure, applications, and communication but also in which people and organizations are involved.

6.3.2 Knowledge Base

The knowledge base provides raw materials from and through which IS research is accomplished. The knowledge base consists of foundations and methodologies. Theories, frameworks, instruments, constructs, models, methods, and instantiations are gathered and used in the develop/ build phase, methodologies are used in the justify/evaluate phase. Table 3 in [22] shows possible methods to test and justify the artifact. To select an appropriate evaluation method, artifacts can be subdivided into Ex Ante, Ex Post, Naturalistic, and Artificial [39]. This work is Ex Post and Artificial, which means that the preferred methodologies are: Mathematical or Logical Proof, Lab Experiment, Role Playing Simulation, Computer Simulation, and Field Experiment. To ensure our algorithm works as intentioned also in a real edge cloud environment, we conducted a lab experiment.

6.3.3 IS Research

Literature Review [38]

A literature review is the ground on which modern research is built. Therefore it is essential to do the literature review detailed and precise. Due to the increasing speed at which research is done, a literature review is getting more complex. Therefore Literature Review as a research method is increasingly important. There are different approaches to doing a literature review, to mention a few: Typical purpose, Research question, and Sample characteristics. Choosing the appropriate one for the thesis ensures a well-designed literature review. We have research questions and use a systematic literature review (Quantitative articles). Several steps must be taken to create a literature review regardless of the chosen methods. The review itself can be split into four phases: (1) designing the review, (2) conducting the review, (3) analyzing, and (4) writing up the review. Those steps should be executed as described in the paper. If finished, the quality of the literature review should be checked. Therefore, Table 4 in [38] presents guidelines to assess the quality of the literature review.

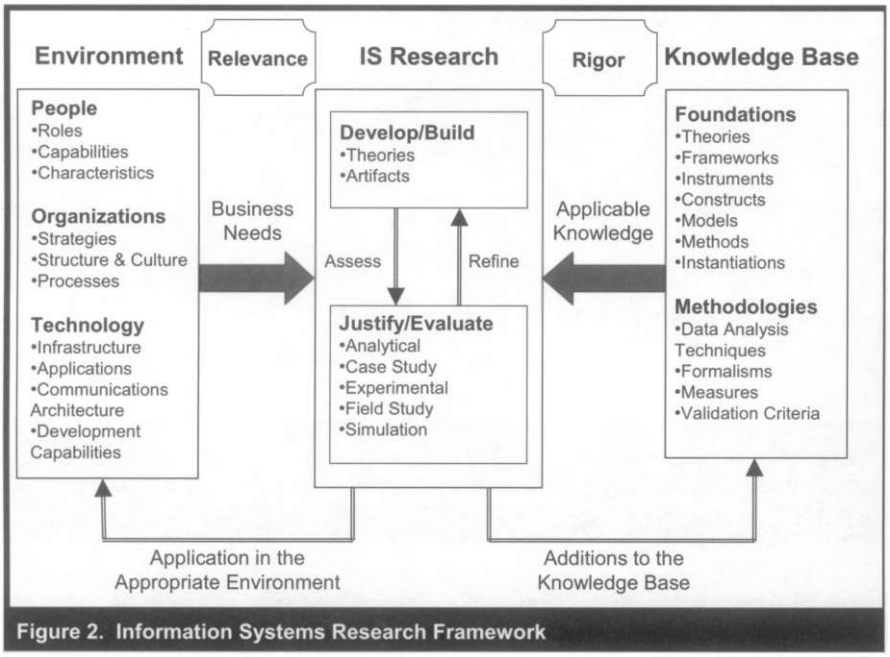


Figure 6.1: Information Systems Research Framework [22].

Laboratory Experiment [20]

Since the actual settings of an existing and running edge server and cloud server with all the parameters are too complex and volatile, a real test bed is built to simulate the actual behavior. Therefore, we have variable parameters, and everything from the bandwidth to clock speed is controlled. The client is a Raspberry Pi, and the server is an AWS EC2 instance. Four parameters, clock speed, network latency, bandwidth, and cost, are considered for the evaluation. Furthermore, three different DNNs are used to show the effect of the rescheduling algorithm regarding different DNNs. Ultimately, all latency measurements of the new proposed trade-off analysis are compared to the best splitting point regarding latency and cost.

6.4 Hardware Setup

In our test bed, the ES is a Raspberry Pi 4 with 4 GB of RAM, installed directly where the data is generated. This has the advantage that no time is lost from the data generation to the ES. The ES hosts three software components; the scheduling algorithm, the prediction models, and the neural networks themselves. Each part is implemented in Python, and the DNNs use PyTorch as the framework. We evaluated our approach with the AlexNet CNN, Mnist classification CNN, and VGG16 CNN. The test data are images with the dimension 28×28 for the Mnist model, dimension 1546×1213 and resolution 72×72 for the VGG16 model, and dimension 4625×6938 and resolution 72×72 for

the Alexnet model. The images for Alexnet and VGG16 are resized to the dimensions 224×224 and 256×256 to fit the model. The cloud server runs in an Amazon EC2 instance using a Linux Server, with four virtual CPUs at a clock speed of 1,5 GHz and with 32 GB of RAM.

6.5 Web Server

The cloud server is implemented as a web server using Python with a REST interface. The web server has only one endpoint responsible for receiving the offloaded data. The endpoint is presented in Listing 2. It is a POST request and requires the splitting point, the tensor base 64 encoded, and the name of the DNN as input parameters. This input data is exemplarily shown in Listing 1.

```
"input_data" : {
    "tensor": "ABD23BSD...X2S",
    "splitting_point": 15,
    "model": "alexnet"
}
```

Listing 1: Data transferred from ES to the web server.

```
@app.route('/offload', methods=['POST'])
def offload():
    input_data = request.get_json()
    model(input_data)

    return "successful"
```

Listing 2: Endpoint at webserver to receive the split NN.

6.6 Performance Metrics

- Bandwidth: To measure the latency with a certain bandwidth setting, it must be possible to set the bandwidth. Therefore, we use Wondershaper, which allows limiting the bandwidth of one or more network adapters during runtime [6].
- Latency: Since all parts of this work are implemented in Python, measuring the execution times is consistent through all components. To measure the execution times, Python function `time.time()` is used, which performs on Unix systems with a precision of 1 microsecond. Since the counter is only valid for one system, we calculate the communication latency by subtracting computation latencies from the total end-to-end latency.

- Ping Latency: To measure the exact response latency between the ES and the CS we ping the server with its IP address and store the latency. This latency is later required for calculating the actual communication latency.

- Storing Measurements: All measurements are stored at the edge in a CSV file. Those files are later read by an evaluation program written by us in Python. An exemplary entrance of the stored measurements is shown in Table 6.1. There we log the start and stop time of the edge, cloud and overall latency. Furthermore, we log the start and stop time of every layer called, the tensor size transferred to the cloud, the time and the splitting point. We must also store the current bandwidth and clock speed for later evaluation.

id	datetime	start	stop	size	SP	BW	CS
gen	4	5213	0.8748994	725	0.8115118	4000	1200000
1	datetime	0.5619493	0.5623462	0	4	0	0
2	datetime	0.562349	0.5624564	0	4	0	0
3	datetime	0.5624578	0.562519	0	4	0	0
4	datetime	0.5625205	0.5627294	0	4	0	0
5	datetime	0.5627317	0.56282	0	4	0	0
6	datetime	0.5628376	0.5629485	0	4	0	0
7	datetime	0.5629501	0.5630035	0	4	0	0
8	datetime	0.5630052	0.5630724	0	4	0	0
9	datetime	0.563074	0.5631313	0	4	0	0
finish	0.8125186	0.8746827	0.5615163	0.5631423	19.917	0	0

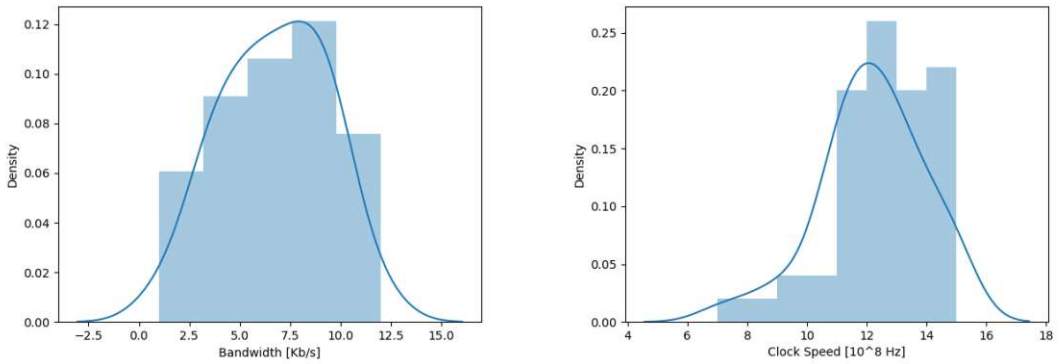
Table 6.1: Exemplarly CSV of stored Measurements.

6.7 Experimental Parameters

For the final measurements, where we compare the three approaches, best cost, best latency, and our trade-off, we used a pool of generated clock speed and bandwidth settings. From there, we randomly selected 15 settings and measured them on our test bed. For the bandwidth, we used a Poisson distribution, and for the clock speed, a normal distribution. We conducted each measurement for all the models three times and used their mean. This is to ensure that the measurements are correct.

6.7.1 Bandwidth

We chose a mean of $7.2Mb/s$ for the bandwidth and used a Poisson distribution to generate 30 bandwidth settings. From these 30 settings, we randomly selected 15 and used them for our measurements. Figure 6.2a shows the histogram of the Poisson distribution used for our measurements with a mean of $7.2Mb/s$.



(a) Poisson Distribution of Bandwidth $Mean = 7.2Mb/s$.

(b) Normal Distribution of Clock Speed $Mean = 1.5GHz$.

Figure 6.2: Distributions for Bandwidth and Clock Speed.

6.7.2 Clock Speed

For the clock speed, we used the maximal setting in our case, $1.5GHz$, as the mean. This is because we assume in a default mode that the CPU is on load and has enough power to run at full clock speed. We then use a normal distribution with a mean of $1.5GHz$. We also generate a pool of 30 clock speed settings and randomly choose for each bandwidth setting one clock speed setting out of this pool.

CHAPTER 7

Evaluation

7.1 Effectiveness of Predictions

In the quest to estimate edge latency, this study advocates the adoption of live clock speed as a predictive metric, given its dynamic nature during runtime. In the context of cloud latency, a rudimentary linear model is proposed, while the calculation of communication latency is direct. The efficacy of our predictive strategies is evaluated by juxtaposing the predicted DNN splitting point against the actual optimal solution. This is achieved by deducting the latency at the optimal splitting point from that at the worst splitting point, thereby ascertaining the relative performance deficit of our approach. The empirical results indicate that for the Mnist model, our approach performs, on average, 30.34% worse than the optimal splitting point. For Alexnet, this figure stands at 20.91%, while for the VGG16 model, the deviation is only 5.36%. Additionally, for the VGG16 and Alexnet models, all proposed splitting points reside within the top 5 potential splitting points. However, for the Mnist model, only 76.47% of proposed points fall within this top 5 range.

7.2 Strategies

Figure 4.5 illustrates the different offloading strategies when latency, cost, or both ($w = 0.5$) are optimized under various clock speeds and fixed bandwidth. It can be seen that latency optimization and the proposed cost/latency optimization are dynamic, whereas the cost is static. In this scenario, the ES is rented; therefore, the cheapest solution is always to offload immediately. It can be observed that the proposed approach identifies a good trade-off solution with only 13.73% higher latency or 35.28% more cost than the two single-objective solutions on average.

7.3 Weights

Figure 7.1 shows the different latencies along with the different offloading strategies when the weight changes. To generate this Figure, we started with a weight of 0.0 in the left top corner and increased the weight by 0.33. The strategy is rent, meaning it is cheaper to offload directly since the ES is more expensive. This is represented by the green dot. The red dot represents the predicted optimal splitting point for the best latency, and the blue one is the novel approach where cost and latency strategy are combined. It can be seen that when the weight is 0.0, the presented best splitting point merges with the best latency strategy since we tell our algorithm to optimize latency fully. The same holds when the weight is 1.0, but only here is the cost fully optimized. The weights between 0.33 and 0.67 optimize slightly more for latency or cost. This results in the provided splitting point in blue moving slightly more to the right and left.

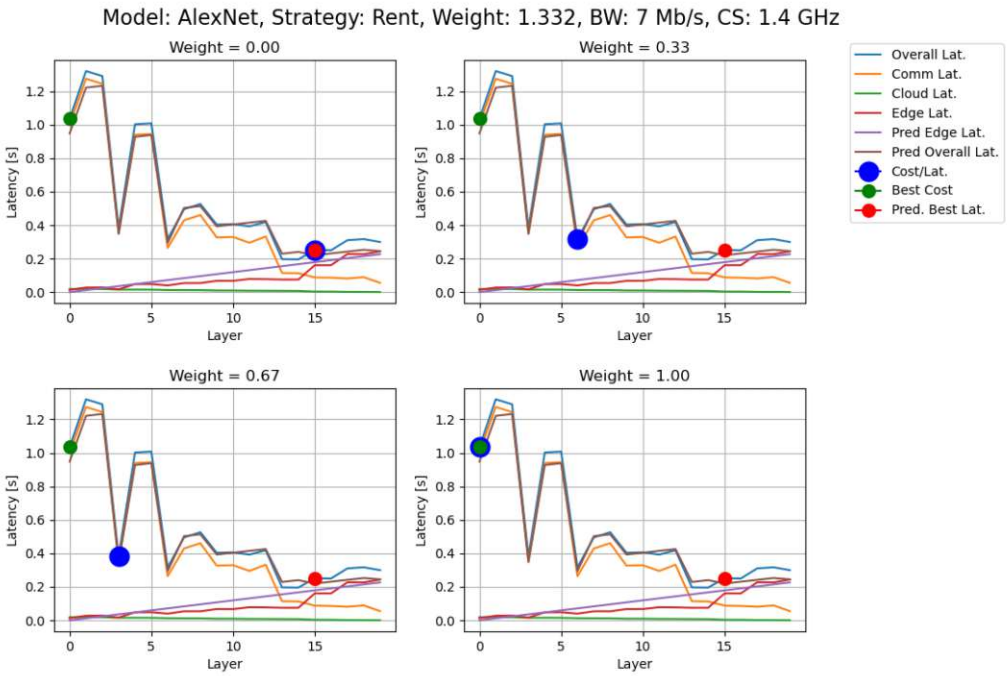


Figure 7.1: Different Strategies for Different Weights, Model: AlexNet [30].

7.4 Latency and Cost

The following two subsections present the numerical results for the three investigated DNNs. We implemented two different cost strategies for the ES, renting and owning. When the ES is rented, the optimal splitting point is to offload immediately since the ES is more expensive than the CS. Although, when the ES is owned, running more layers at

the edge is cheaper since we assume there are no costs. Therefore not only the cost of the ES is prevented, but also the costs when running at the CS. For all the following measurements, we use a weight of 0.5, meaning we try to find the best balance between price and latency without a preference for one. For the following measurements, we chose 15 samples from a pool with Poisson-distributed bandwidths and normally distributed clock speeds. Each measurement is run three times, and the median of the runs has been taken. The selection of the measurements is described in detail in Section 6.7. The following Figures 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8 compare the costs of the different offloading strategies, best cost, best latency, or the proposed combination. The green bar represents the splitting point for the optimal price, the green bar shows the best-predicted splitting point regarding latency, and the red bar is our proposed trade-off between both.

7.4.1 Rent

The first strategy is renting the ES. Therefore we take real-life prices from AWS wavelength edge servers and cloud servers shown in Table 4.1. To calculate the fee charged, we multiply the different latencies by the time per hour in seconds. As an example, the AlexNet model has 19 layers. If the splitting point is predicted at layer ten, e.g., then we charge the time it took to run the ten layers at the edge with the edge price. The lasting layers are computed in the cloud, and we multiply the cloud price by the time it takes to finish the model. The following results are exemplarily explained for the AlexNet model, although it reads the same for VGG16 and Mnist models. Figures 7.2 and 7.3 show the two measurements when renting the edge server. On the x-axis, we have the bandwidth in Mb/s and a clock speed setting in GHz. On the y-axis, we have costs in dollars and latency in seconds. The corresponding data is shown in Table 7.1a. The table is divided into two columns, the *Optimal Cost (OC)* and the *Optimal Latency (OL)*. The rows, then, show the increase or decrease in eighter cost or latency. For example, for

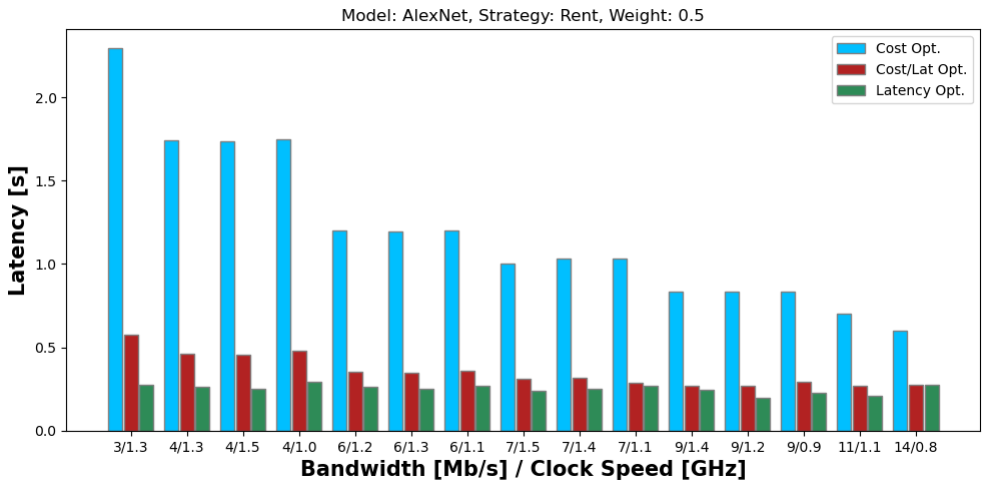


Figure 7.2: Latency Measurement for: Model: AlexNet, Strategy: Rent, Weight: 0.5 [30]

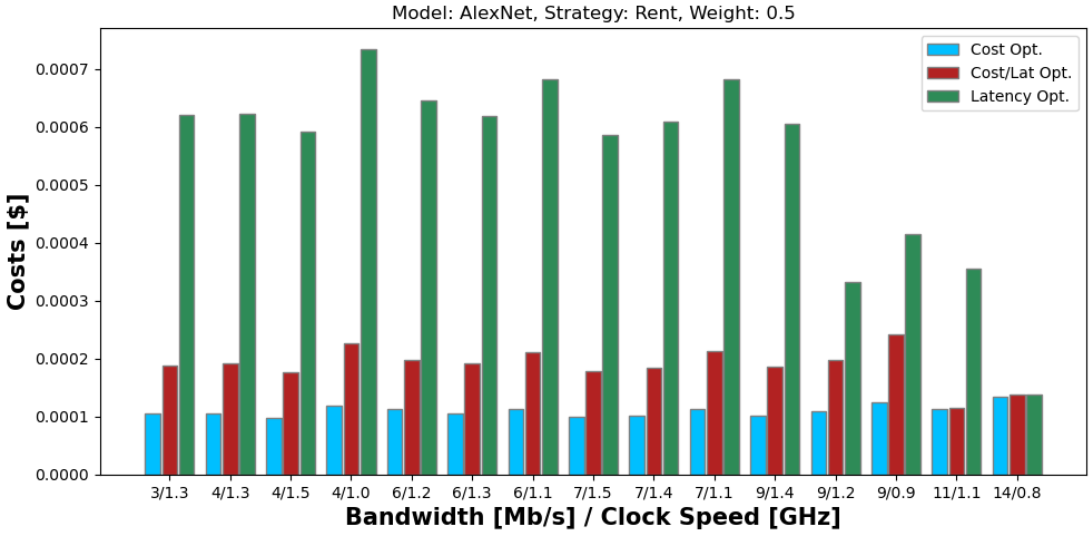


Figure 7.3: Cost Measurement for: Model: AlexNet, Strategy: Rent, Weight: 0.5 [30]

latency and renting, our new trade-off approach *Cost/Latency (CL)* decreases latency by 68.93% compared to the best cost approach while at the same time only increases the cost by 25.92% in comparison to the best cost approach. If the optimal cost approach is selected, this will result in a latency increase of 76.48% compared to the best latency approach. Figure 7.2 shows the latencies for the different bandwidth and clock speed settings when renting the ES. In blue, we have the optimal cost strategy; in green, the optimal latency strategy; and in red, our proposed trade-off. Figure 7.3 shows the cost for the measurements. The same table shows that our trade-off strategy is 60.84% cheaper than the optimal latency approach and only 39.77% more expensive than the optimal cost approach. In comparison, if the *OC* approach had been used, this would have resulted in a 74.93% decrease in latency.

7.4.2 Own

The second strategy is owning the ES. This is the case in the SWAIN project, where multiple servers are purchased and privately installed. Therefore we assume no further costs when running tasks on the ES. Thus running at the edge does not only reduce costs at the edge but also reduces the overall costs since the time running in the cloud is reduced. Exemplary, we explain the tables and figures for the Mnist model when the ES is owned. In Table 7.1c, we can see that, regarding the cost, we have with CL a decrease in the cost of 24.78% while at the same time no increase in comparison to the OC. This is because the CL merges with the OC in this scenario. Regarding latency, however, we have an increase of 1.11% and no improvement compared to OC since OC and CL propose the same splitting point. The results for the other measurements can be read the same way.

7.4.3 Results

		Optimal Cost (OC)	Optimal Latency (OL)
Own			
Cost	CL	-57.02%	50.32%
	OC		86.44%
Latency	CL	8.05%	-14.22%
	OL	21.43%	
Rent			
Cost	CL	-39.77%	60.84%
	OC		74.93%
Latency	CL	68.93%	-25.92%
	OL	76.48%	

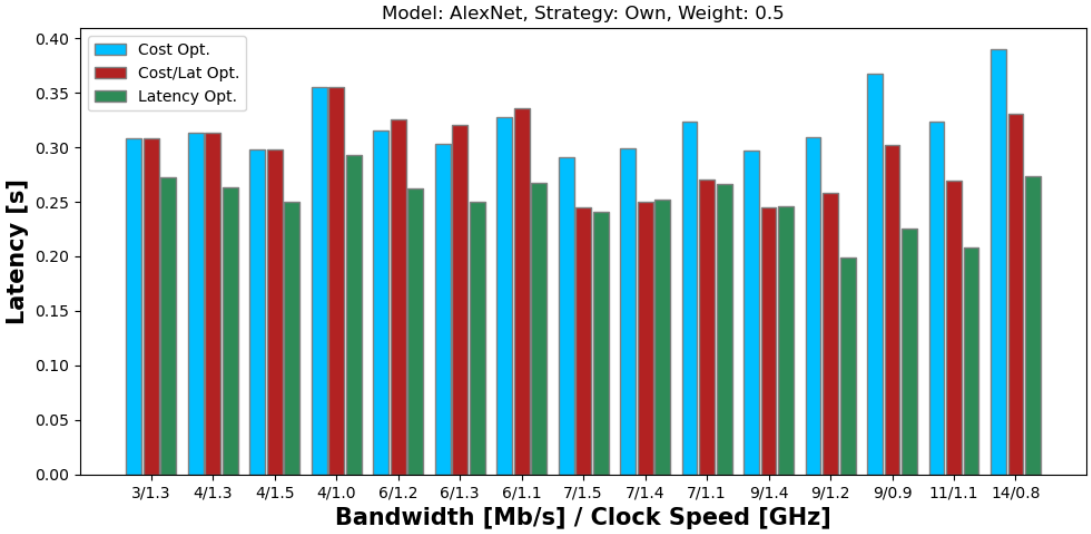
(a) AlexNet Results

		Optimal Cost (OC)	Optimal Latency (OL)
Own			
Cost	CL	-76.18%	94.85%
	OC		98.22%
Latency	CL	1.47%	-42.18%
	OL	42.85%	
Rent			
Cost	CL	0.00%	7.77%
	OC		7.77%
Latency	CL	0.00%	0.61%
	OL	-0.61%	

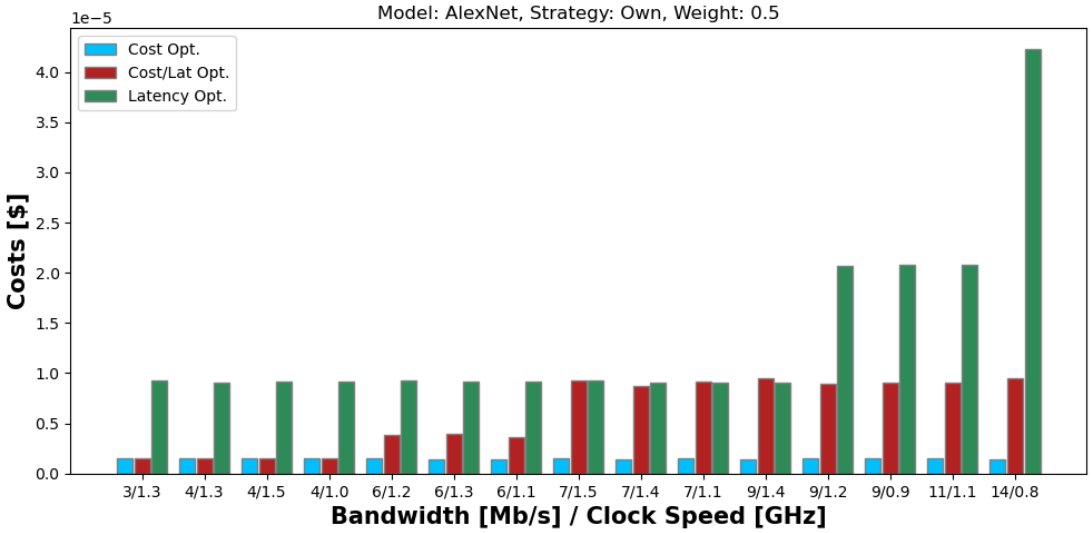
(b) VGG16 Results

		Optimal Cost (OC)	Optimal Latency (OL)
Own			
Cost	CL	0.00%	24.78%
	OC		24.78%
Latency	CL	0.00%	-1.11%
	OL	1.11%	
Rent			
Cost	CL	0.00%	26.78%
	OC		26.78%
Latency	CL	0.00%	-3.29%
	OL	3.29%	

(c) MNIST CNN Results

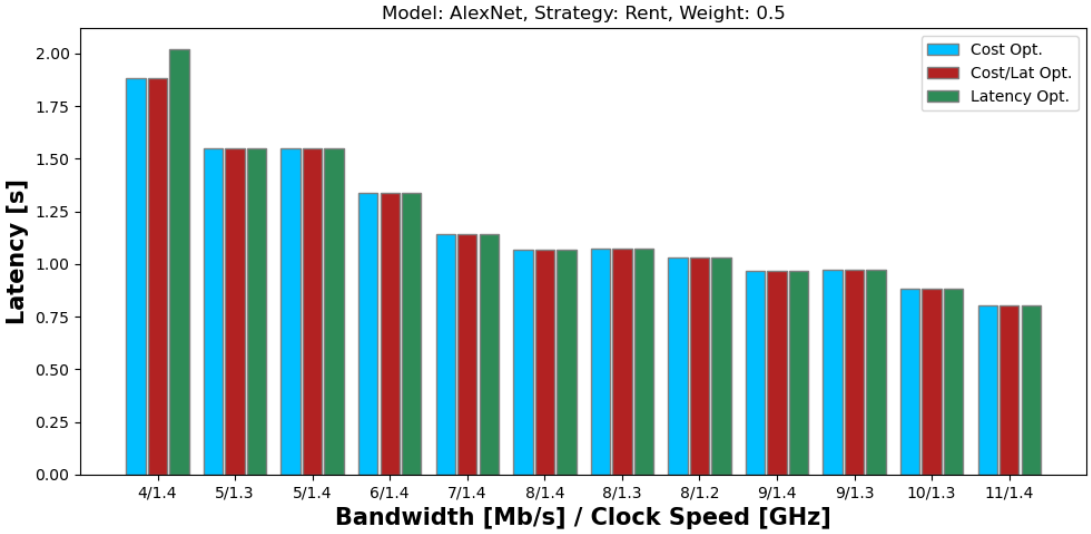


(a) Latency Measurement for: Model: AlexNet, Strategy: Own, Weight: 0.5

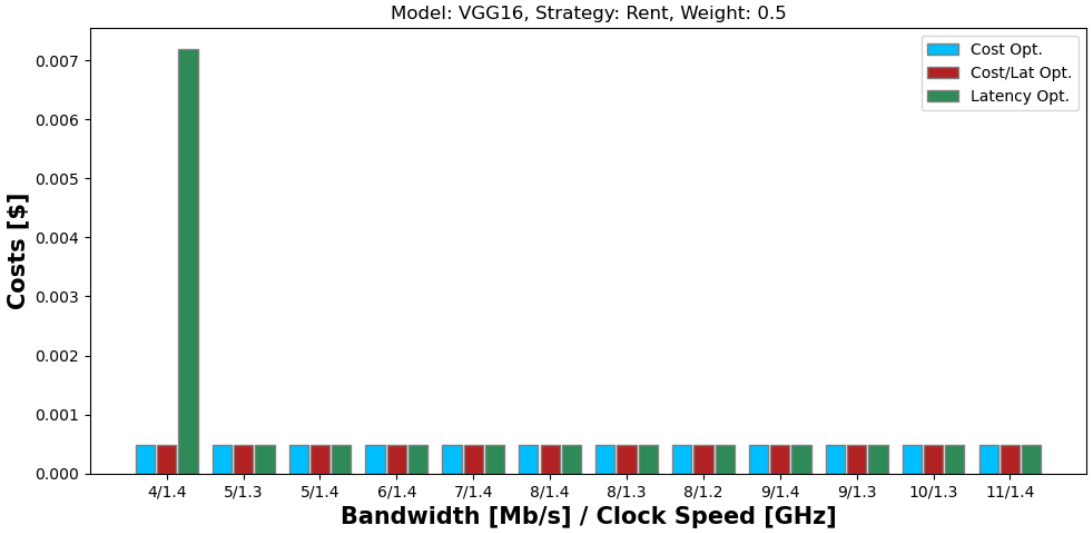


(b) Cost Measurement for: Model: AlexNet, Strategy: Own, Weight: 0.5

Figure 7.4: Alexnet Cost and Latency Optimization (w=0.5).

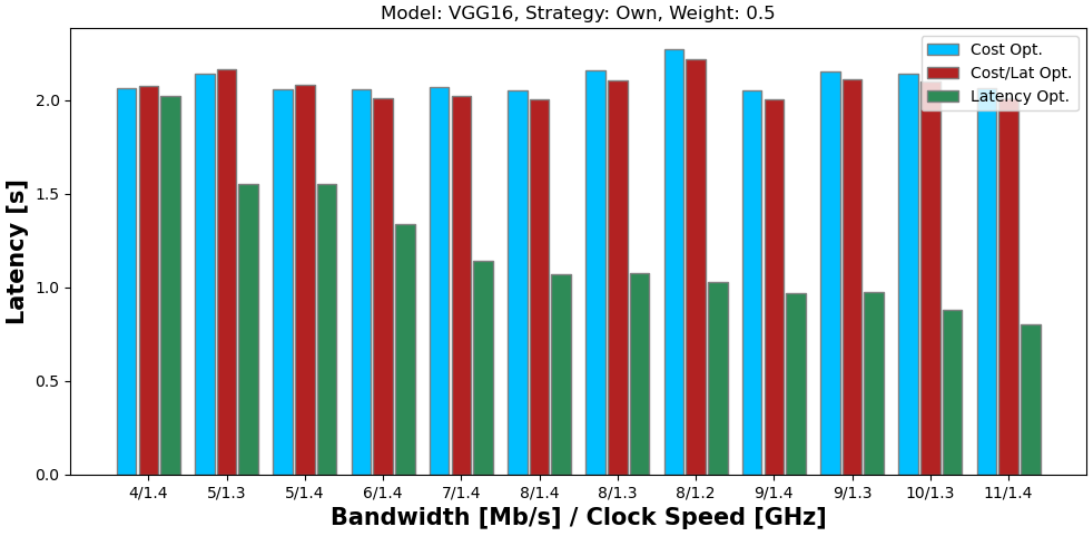


(a) Latency Measurement for: Model: VGG16, Strategy: Rent, Weight: 0.5

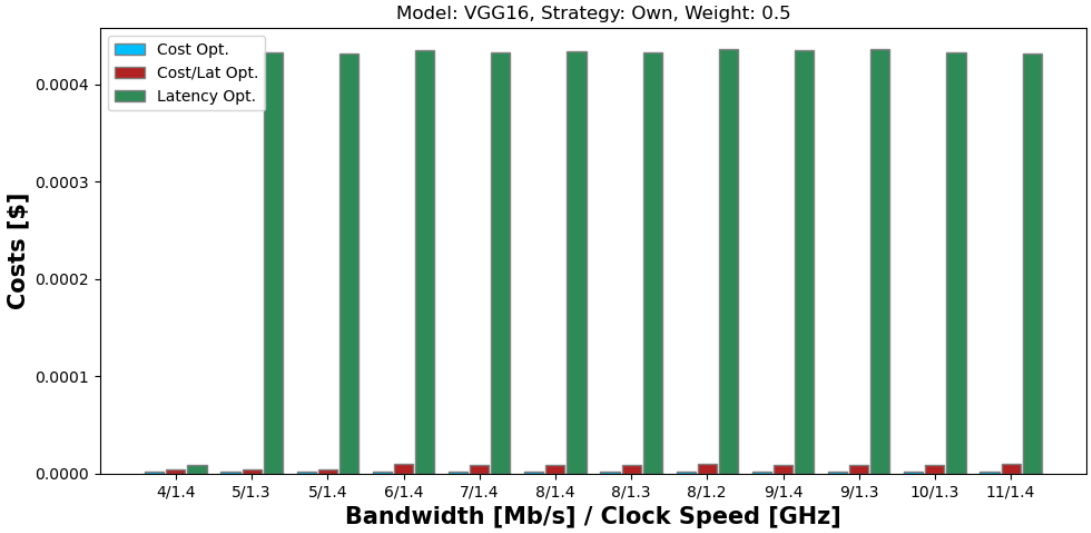


(b) Cost Measurement for: Model: VGG16, Strategy: Rent, Weight: 0.5

Figure 7.5: VGG16 Cost and Latency Optimization.

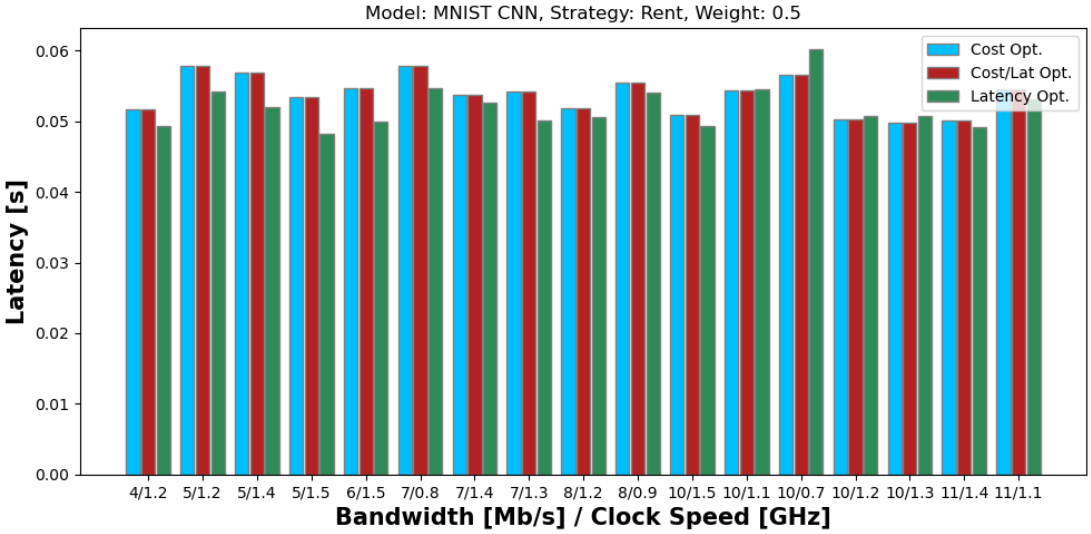


(a) Latency Measurement for: Model: VGG16, Strategy: Own, Weight: 0.5

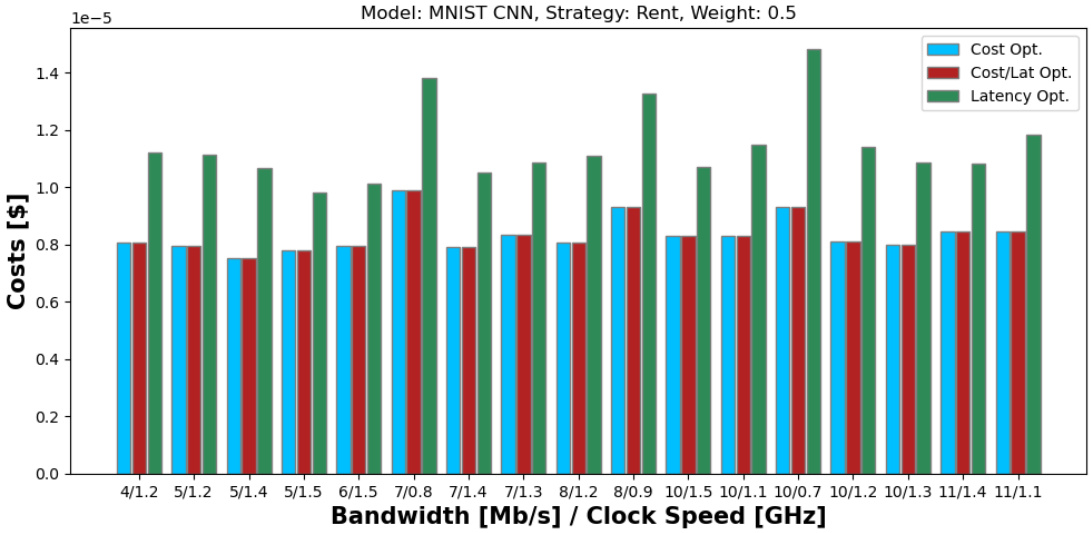


(b) Cost Measurement for: Model: VGG16, Strategy: Own, Weight: 0.5

Figure 7.6: VGG16 Cost and Latency Optimization.

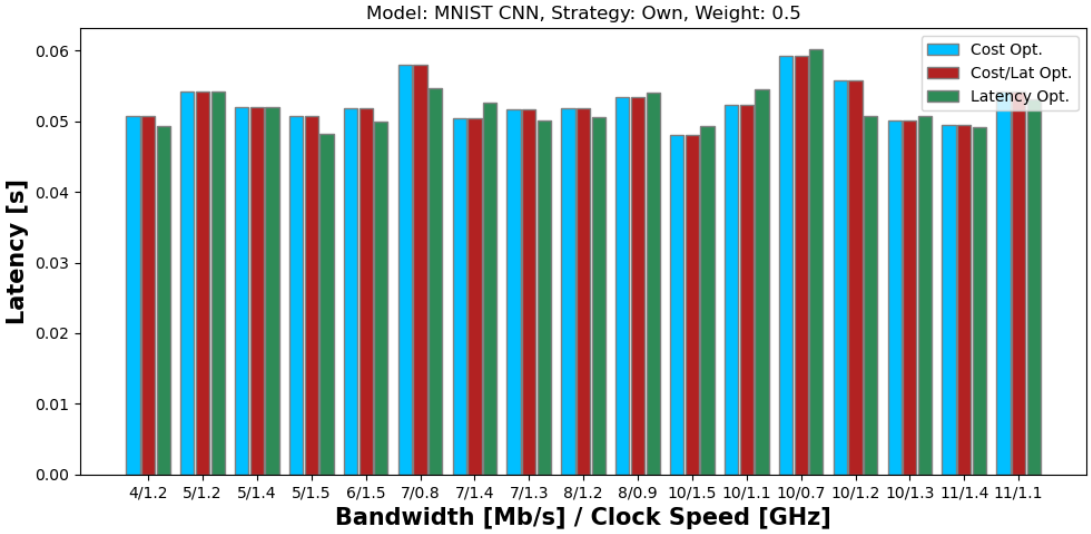


(a) Latency Measurement for: Model: MNIST CNN, Strategy: Rent, Weight: 0.5

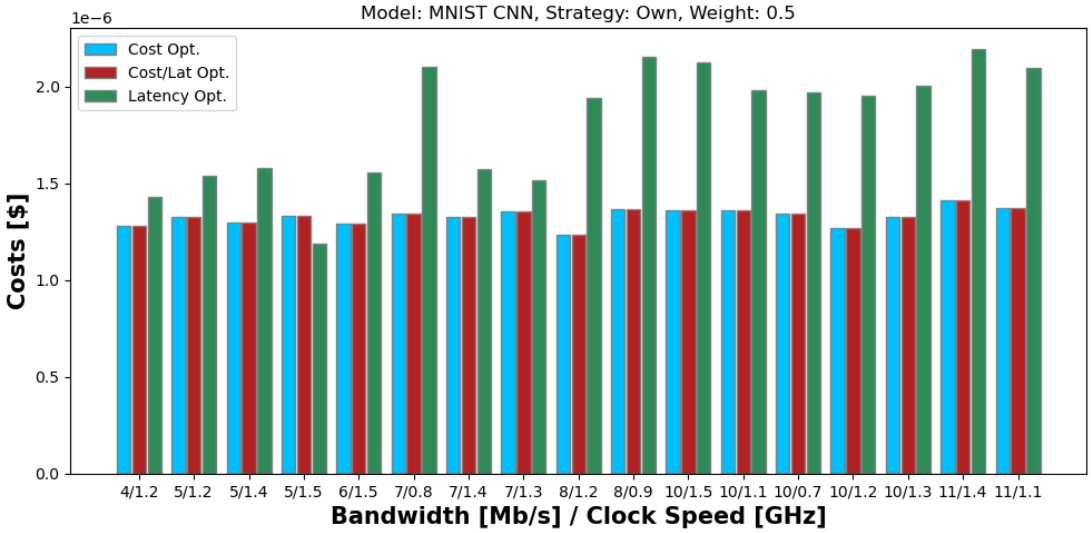


(b) Cost Measurement for: Model: MNIST CNN, Strategy: Rent, Weight: 0.5

Figure 7.7: MNIST CNN Cost and Latency Optimization.



(a) Latency Measurement for: Model: MNIST CNN, Strategy: Own, Weight: 0.5



(b) Cost Measurement for: Model: MNIST CNN, Strategy: Own, Weight: 0.5

Figure 7.8: MNIST CNN Cost and Latency Optimization.

7.5 Pareto Front

The Pareto front is a concept in multi-objective optimization that represents the optimal trade-off between two or more conflicting objectives. It consists of a set of solutions that are not dominated by any other solution, meaning that improving one objective would necessarily require a compromise on another. The Pareto front is used to help decision-makers identify the best solution based on their preferences and priorities. It is named after Vilfredo Pareto, an Italian economist and sociologist who introduced the concept in the context of income distribution in the early 20th century [31]. In the case of a trade-off between latency and cost, the Pareto front can help identify the set of solutions that are optimal in terms of both criteria. By considering different combinations of latency and cost, the Pareto front can help decision-makers visualize the trade-offs between these criteria and select the most appropriate solution based on their priorities. This is shown in Figure 7.9 with latency in seconds on the x and cost in dollars on the y-axis. The red dots represent all possible splitting points for all measurements with all the different weights from the AlexNet model. It can be seen that the red dots are approximately shaped like a Pareto front. In the following section, we present the numerical results of the different DNNs with different strategies.

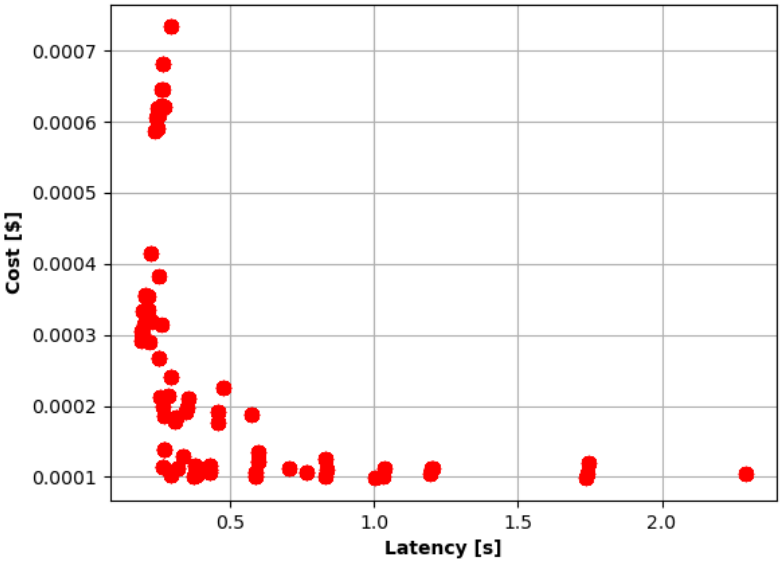


Figure 7.9: Pareto Front of Latency and Cost [30].

CHAPTER 8

Conclusion

Deep neural networks (DNNs) have revolutionized mobile applications, such as Apple Siri and Google Assistant, by delivering exceptional artificial intelligence capabilities in computer vision, natural language processing, and big data analysis. However, the computational demands of DNN tasks can overwhelm UEs with their limited energy and computing capacities. To address this challenge, edge computing has emerged as a promising solution. Edge servers, equipped with ample computation resources, deployed in close proximity to UE. This allows UEs to offload portions of their computationally intensive tasks to the edge servers, effectively alleviating the burden on the UEs. Task partitioning and offloading play pivotal roles in the practical implementation of MEC, enabling efficient utilization of computation and communication resources [17]. However, even ES can struggle to compute complex algorithms if their capabilities are limited. This holds especially for environmental monitoring where they are installed in unaffiliated regions with limited resources. For those use cases, DNN splitting brings an advantage as well. However, not only latency should be taken into account. For some use cases, where latency plays not a key role, the cost can be a driving factor. ESs are especially expensive when rented, therefor computing as little as possible at the edge reduces costs. However, organizations often invest in local ESs. Therefore computing at the edge has a cost advantage, and the cloud becomes the key cost factor. The tackle those challenges, we created a dynamic rescheduler that takes the real-time clock speed, bandwidth, and latency into account and predicts, through prediction models, the best splitting point for offloading regarding cost and latency. This is crucial for edge servers, which run DNNs in a volatile environment such as environmental monitoring or autonomous driving. The results show that our algorithm combines cost and latency optimization and can reduce cost as well as latency. A Pareto Front is presented, which allows a user of this rescheduler the focus on either cost or latency or a weighted combination. The answers to the research questions can be described as follows. *i)* The reduction of latency compared to the best cost strategy and vice versa is solved in Section 7.4.3. The results demonstrate that we

achieved a 60.84% decrease in cost compared to the optimal splitting point regarding latency with an increase in latency of only 25.92% for the AlexNet CNN when the edge server is rented. *ii)* In Section 4.1 we proposed a prediction model which uses the cloud speed to predict the edge latency for a specific model. It has to be mentioned that for the usage of this prediction model, two measurements for de-normalization are required. Section 7.1 evaluates the precision of the proposed prediction model. The empirical data suggest a differential in performance between our approach and the optimal splitting point for various models. Specifically, our method exhibits an average of 30.34% higher latency for the Mnist model. In the case of Alexnet, this relative performance deficit is 20.91%. Conversely, for the VGG16 model, the observed discrepancy is only 5.36%. *iii)* We proposed a multi-optimization algorithm that includes cost, bandwidth, and network latency and finds a trade-off between cost and latency. This algorithm is explained in Section 4.5. The next open challenge will be to improve this rescheduling algorithm to include other parameters like CPU Type, RAM, or kernels, which might enable this scheduling algorithm to work on different types of edge resources. Furthermore, to improve the prediction of the best splitting point in terms of latency, the Neurosurgeon approach could be used where the prediction happens based on the layer types. Another interesting topic could be to combine this cost model and dynamic rescheduling with the early exit approach to further reduce cost and latency.

Acknowledgments

This work was supported by the CHIST-ERA grant CHIST-ERA-19-CES-005, by the Austrian Science Fund (FWF): I 5201-N and Y904-N31 START-Programm 2015, and by the FFG Flagship Project High Performance Integrated Quantum Computing (HPQC): #45285029.

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 1, 7, 8

ANN Artificial Neural Networks. 9

CNN Convolutional Neural Network. 10

CPU Central Processing Unit. 2

DL Deep Learning. 1, 9

DNN Deep Neural Network. 1, 9, 10

EI Edge Intelligence. 1

ES Edge Server. 1

FCNN Fully Connected Neural Network. 10

ML Machine Learing. 1

MLP Multilayer Perceptron. 10

RNN Recurrent Neural Network. 11

UE User Equipment. 15

Bibliography

[1] Gartner hype cycle for ai 2019. https://www.gartner.com/smarterwithgartner/top-trends-on-the-gartner-hype-cycle-for-artificial-intel 2019. Accessed: 2023-04-03.

[2] Gartner hype cycle for ai 2022. https://www.gartner.com/en/articles/what-s-new-in-artificial-intelligence-from-the-2022-gartner-hype-cycle, 2022. Accessed: 2023-04-03.

[3] Ahmad, S., and Aral, A. FedCD: Personalized federated learning via collaborative distillation. In *Workshop on Distributed Machine Learning for the Intelligent Computing Continuum (DML-ICC)* (Vancouver, WA, 2022), IEEE.

[4] An, S., Lee, M., Park, S., Yang, H., and So, J. An ensemble of simple convolutional neural network models for mnist digit recognition, 2020.

[5] AWS. On-demand plans for amazon ec2. https://aws.amazon.com/ec2/pricing/on-demand/, 2023. Accessed: 2023-02-22.

[6] Bert Hubert, Jacco Geul, S. S. The wonder shaper. https://github.com/magnific0/wondershaper, 2021. Accessed: 2023-02-22.

[7] Bouwmans, T., Javed, S., Sultana, M., and Jung, S. K. Deep neural network concepts for background subtraction: A systematic review and comparative evaluation. *Neural Networks 117* (2019), 8–66.

[8] Cao, K., Liu, Y., Meng, G., and Sun, Q. An overview on edge computing research. *IEEE Access 8* (2020), 85714–85728.

[9] Chen, J., and Ran, X. Deep learning with edge computing: A review. *Proceedings of the IEEE 107*, 8 (2019), 1655–1674.

[10] Chen, T. Y.-H., Ravindranath, L., Deng, S., Bahl, P., and Balakrishnan, H. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2015), SenSys '15, Association for Computing Machinery, p. 155–168.

[11] Ciresan, D. C., Meier, U., Gambardella, L., and Schmidhuber, J. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation 22* (2010), 3207–3220.

[12] Cisco. Cisco annual internet report (2018–2023) white paper, 2020.

[13] Contributors, T. Alexnet. https://pytorch.org/vision/main/models/generated/torchvision.models.alexnet.html, 2017. Accessed: 2023-04-03.

[14] Contributors, T. Vgg16. https://pytorch.org/vision/main/models/generated/torchvision.models.vgg16.html, 2017. Accessed: 2023-04-03.

[15] De Maio, V., and Brandic, I. First hop mobile offloading of dag computations. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (Washington, DC, USA, 2018), IEEE, pp. 83–92.

[16] Ding, A. Y., Peltonen, E., Meuser, T., Aral, A., Becker, C., et al. Roadmap for edge ai: a dagstuhl perspective. *ACM SIGCOMM Computer Communication Review 52*, 1 (2022), 28–33.

[17] Gao, M., Shen, R., Shi, L., Qi, W., Li, J., and Li, Y. Task partitioning and offloading in dnn-task enabled mobile edge computing networks. *IEEE Transactions on Mobile Computing 22*, 4 (2021), 2435–2445.

[18] Gill, S. S., Xu, M., Ottaviani, C., Patros, P., Bahsoon, R., et al. Ai for next generation computing: Emerging trends and future directions. *Internet of Things 19* (2022), 100514.

[19] Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., and Lew, M. S. Deep learning for visual understanding: A review. *Neurocomputing 187* (2016), 27–48. Recent Developments on Deep Big Vision.

[20] Haki, K., Beese, J., Aier, S., and Winter, R. The evolution of information systems architecture: An agent-based simulation model. *MIS Quarterly 44*, 1 (2020).

[21] Han, Y., Wang, X., Leung, V. C. M., Niyato, D., Yan, X., and Chen, X. Convergence of edge computing and deep learning: A comprehensive survey. *CoRR abs/1907.08349* (2019).

[22] Hevner, A. R., March, S. T., Park, J., and Ram, S. Design science in information systems research. *MIS Quarterly 28*, 1 (2004), 75–105.

[23] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. Mobile edge computing—a key technology towards 5g. *ETSI white paper 11*, 11 (2015), 1–16.

[24] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News 45*, 1 (2017), 615–629.

[25] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[26] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436–444.

[27] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[28] Lin, B., Huang, Y., Zhang, J., Hu, J., Chen, X., and Li, J. Cost-driven off-loading for dnn-based applications over cloud, edge, and end devices. *IEEE Transactions on Industrial Informatics 16*, 8 (2020), 5456–5466.

[29] Luger, D. A latency evaluation for edge intelligence, 2021.

[30] Luger, D., Aral, A., and Brandic, I. Cost-aware neural network splitting and dynamic rescheduling for edge intelligence. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking* (New York, NY, USA, 2023), EdgeSys '23, Association for Computing Machinery, p. 42–47.

[31] Pareto, V. *Cours d'économie politique*, vol. 1. Librairie Droz, 1964.

[32] Peltonen, E., Ahmad, I., Aral, A., Capobianco, M., Ding, A. Y., et al. The many faces of edge intelligence. *IEEE Access 10* (2022), 104769–104782.

[33] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533–536.

[34] Satyanarayanan, M. The emergence of edge computing. *Computer 50*, 1 (2017), 30–39.

[35] Shuvo, M. M. H., Islam, S. K., Cheng, J., and Morshed, B. I. Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proceedings of the IEEE 111*, 1 (2023), 42–91.

[36] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2015.

[37] Simsek, M., Aijaz, A., Dohler, M., Sachs, J., and Fettweis, G. 5g-enabled tactile internet. *IEEE Journal on Selected Areas in Communications 34*, 3 (2016), 460–473.

[38] Snyder, H. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research 104* (2019), 333–339.

[39] Venable, J., Pries-Heje, J., and Baskerville, R. A comprehensive framework for evaluation in design science research. In *Design Science Research in Information Systems. Advances in Theory and Practice: 7th International Conference, DESRIST 2012, Las Vegas, NV, USA, May 14-15, 2012. Proceedings 7* (2012), Springer, pp. 423–438.

[40] Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K., and Zhang, J. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE 107*, 8 (2019), 1738–1762.