TU WIEN Informatics

System Support & Orchestration Mechanisms for Distributed DNN Inference

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Sofware Engineering & Internet Computing

eingereicht von

Matthias Reisinger, BSc
Matrikelnummer 01025631

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Pantelis Frangoudis, PhD

Wien, 25. Jänner 2022

Matthias Reisinger	Schahram Dustdar

TU WIEN Informatics

System Support & Orchestration Mechanisms for Distributed DNN Inference

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Sofware Engineering & Internet Computing

by

Matthias Reisinger, BSc
Registration Number 01025631

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Pantelis Frangoudis, PhD

Vienna, 25th January, 2022

Matthias Reisinger	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Reisinger, BSc

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, 25. Jänner 2022

Matthias Reisinger

Danksagung

Zuallererst möchte ich Schahram Dustdar und Pantelis Frangoudis für die Betreuung dieser Arbeit danken. Mein Dank gilt auch Thomas Rausch, der mir freundlicherweise empfohlen hat, mich an Pantelis zu wenden. Danke, Pantelis, für die vielen Stunden fruchtvoller Diskussion sowie die unbezahlbaren und lehrreichen Ratschläge.

Abschließend möchte ich meiner Familie und meinen Freunden für ihre endlose und bedingungslose Unterstützung danken.

Acknowledgements

First and foremost, I want to thank Schahram Dustdar and Pantelis Frangoudis for supervising this thesis. I also want to thank Thomas Rausch who referred me to Pantelis. Thank you, Pantelis, for the many hours of fruitful discussion as well as the invaluable and instructive advice on how to improve this work.

Finally, I want to thank my family and friends for their endless and unconditional support.

Kurzfassung

Der Einsatz von Edge Computing als Plattform für die verteilte Ausführung von tiefen neuronalen Netzen ist ein aktiver Forschungsbereich. Aktuelle Arbeiten schlagen neue Architekturen künstlicher neuronaler Netze vor, die die Verteilung von Berechnungen in solchen Umgebungen begünstigen. Zusätzlich zum Classifier im obersten Layer führen diese Architekturen sogenannte "Side-Exits" in Zwischen-Layern ein. Dieser Ansatz ermöglicht es, Klassifizierungs-Resultate bereits an früheren Stellen im neuronalen Netz zu erhalten und dadurch Berechnungs-Kosten zu senken, was entscheidend für den Einsatz auf Geräten mit limitierten Ressourcen ist.

Die vorliegende Arbeit folgt diesem Forschungszweig, der diese neuartigen Architekturen verwendet, um die Ausführung neuronaler Netze zu weniger leistungsfähigen Geräten am Netzwerkrand zu verlagern. Im Unterschied zu bestehenden Arbeiten, deren Fokus algorithmischen Aspekten gilt, stellt die vorliegende Arbeit Design-Aspekte in den Vordergrund, die die Implementierung eines erweiterbaren Orchestrierung-Frameworks ermöglichen sollen. Jedes Gerät betreibt eine Laufzeitumgebung, die APIs zur Orchestrierung und Ausführung von neuronalen Netzen bietet, sowie eine Komponente zur Überwachung von Ressourcen und Netzwerkbedingungen. Teilnehmende Geräte registrieren sich bei einem zentralen Controller, der eine globale Sicht auf die Geräte-Hierarchie verwaltet. Schließlich entscheidet ein Scheduler über die Zuweisung und Orchestrierung eines gegebenen neuronalen Netzes auf die zur Verfügung stehenden Rechner. Dieser Scheduler bietet ein Plugin Framework, das es Benutzern erlaubt, eigene Algorithmen für individuelle Strategien zu implementieren und anzuwenden.

Das System bietet bereits eine Reihe solcher Algorithmen zur Minimierung der Ende-zu-Ende-Latenz der Ausführung neuronaler Netze. Der optimale Betrieb in der beschriebenen Systemlandschaft im Hinblick auf minimale Latenz, ist ein NP-hartes, kombinatorisches Optimierungsproblem. Daher bietet das System einen exakten Algorithmus in Form eines ganzzahligen linearen Programms, das dieses Problem optimal löst, sowie heuristische Ansätze für größere Problem-Instanzen.

Abschließend evaluieren wir eine prototypische Implementierung in simulations-basierten Szenarien sowie in einer physischen Testumgebung. In simulierten Umgebungen übertrifft der exakte Algorithmus klar die traditionelle Cloud-basierte Ausführung. Eine Machbarkeitsstudie in einer physischen Testumgebung bestätigt, dass das System basierend auf beobachteten Umgebungsbedingungen effiziente Orchestrierungs-Entscheidungen trifft.

Abstract

The use of edge computing as a platform for distributed DNN inference is an active area of research. Recent work proposes new neural network architectures that facilitate the distribution of DNN workloads in such environments. In addition to the classifier on a DNN's final layer, these architectures introduce side-exit classifiers at intermediate layers. With this approach it is possible to obtain inference results at earlier points in the network and thereby reduce the compute overhead, which is critical for the operation on more constrained devices.

This thesis follows a recent line of research, that uses this novel architecture to shift DNN computations towards less powerful devices at the edge of the network, to improve user experience. In contrast to related work, which is more focused on algorithmic aspects to optimize the distributed execution of DNNs, this thesis puts a focus on the design aspects that enable the implementation of an extensible orchestration framework for distributing inference of feed-forward DNN models. Each host in the compute hierarchy operates a runtime environment that offers APIs for orchestration and execution of DNNs, as well as a component for monitoring the node's resource levels and network conditions. Compute nodes are required to register with a central controller, which maintains a global view on the compute hierarchy. Finally, a scheduler decides about the deployment and orchestration of a given DNN model over the available compute resources. From a software architecture perspective, the scheduler offers a plugin framework, that allows system users to implement and apply their own algorithms for custom placement policies.

The system also readily comes with a number of strategies, that aim to minimize end-to-end latency of the DNN inference. We show the optimal placement of layers in the described system landscapes to be an NP-hard combinatorial optimization problem, with respect to minimizing latency. Therefore, we provide an exact algorithm, in the form of an integer linear program, that solves the placement problem to optimality, as well as heuristic approaches for bigger problem instances.

Finally, experimental studies evaluate a prototypical system implementation in simulation-based scenarios and on a physical test-bed. On simulated compute hierarchies, the exact placement clearly outperforms the traditional cloud-centric placement. A feasibility study on a physical test-bed confirms that the system is able to identify efficient placements based on monitored environmental conditions.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In recent years, Deep Neural Networks (DNNs) have proven to be a potent tool for large-scale data analytics. Advances in hardware have spurred the design of deeper and more powerful DNN architectures which enabled their application for learning tasks in traditionally complex domains such as image classification, video analytics, and speech recognition.

Due to the limited computational capacity of end-user or IoT devices, DNNs are typically operated as centralized services in remote cloud data centers, that provide the needed compute resources. This design conflicts with application scenarios that depend on a high degree of responsiveness or lack the required network bandwidth for streaming large amounts of data to the cloud.

In contrast to such a monolithic service design, edge computing follows a more distributed approach that moves the processing of data nearer to the end-devices from where the data originate. The layered architecture of DNNs inherently lends itself to be mapped over such a distributed compute hierarchy in order to reduce inference time and end-to-end latency. Recent research also proposes new network architectures that facilitate the distributed execution of DNN inference. Instead of using only one classifier at the final network layer, these architectures introduce additional side-exit classifiers at intermediate layers that allow to obtain inference results at earlier points in the network. Each of these classifiers is associated with a configurable confidence threshold that determines whether the classification result for a given input sample is sufficiently accurate to stop inference at this side-exit. In the context of a distributed DNN, this architecture allows to retrieve prediction results at earlier compute nodes that are closer to the end devices.

1.2 Problem Statement

The proposed thesis sets out to address the following research questions in order to determine how DNN inference can be enabled in a distributed environment:

- ***RQ1:*** *What is the necessary system support that is required to enable the deployment, management, and orchestration of a DNN on top of the edge-to-cloud continuum?* In particular, this question addresses the required system components as well as the design of APIs and application-layer protocols for communication between components.
- ***RQ2:*** *How can a DNN be deployed and executed optimally or near-optimally in such an environment?* This could for example regard the placement of DNN layers in the device, edge, and cloud domain and their distribution to concrete hosts.
- ***RQ3:*** *Is it possible to dynamically adapt the deployed DNN in response to changing operational conditions?* Potential dynamic adaptions might involve the migration of DNN layers between compute nodes or the dynamic tuning of confidence thresholds based on current operational conditions.

1.3 Organization

The thesis is organized as follows. Chapter 2 discusses concepts and methods that build the basis for the work of this thesis. In Chapter 3, we review related work. The proposed system design is outlined in chapter 4. A reference implementation of the proposed system design is presented in chapter 5. Different placement strategies, that operate on top of the designed system APIs, are discussed in Chapter 6. In Chapter 7 we present an in-depth evaluation of the proposed system and placement algorithms based on different exemplary scenarios. Finally, Chapter 8 concludes this work and discusses possible directions for future work.

CHAPTER 2

Background

This chapter provides an overview of fundamental concepts and notions that build the basis for this thesis. Section 2.1 provides an overview of neural networks and introduces basic deep learning taxonomy. In section 2.2, we discuss neural network architectures that include multiple exit branches. The concept of DNN partitioning is described in section 2.3. As mentioned earlier, this thesis also aims at leveraging end-devices and the network edge for neural network inference. Therefore section 2.4 gives a brief overview over edge and fog computing in general.

2.1 Neural Networks and Deep Learning

From product recommendation in popular e-commerce websites to the diagnosis of illness in health care, machine learning powers many of today's applications and consumer products. Especially, in the era of big data, deep neural networks (DNNs) have proven to be a potent tool for large-scale data analytics. Traditionally, machine learning methods were not able to process data in its raw form. Extensive domain knowledge was necessary to engineer application-specific feature extractors to obtain suitable feature vectors from the raw input data. Due to the significant efforts for handcrafting good feature representations, research has aimed for automating these feature engineering tasks.

In contrast to conventional machine learning techniques, neural networks are able to learn data representations in an automated manner. In the 1970s and 1980s multiple research groups discovered, independently from each other, that multi-layer networks can be trained automatically via *stochastic gradient descent (SGD)* based on a *backpropagation* procedure [LBH15]. Backpropagation is a mechanism to compute the gradient of a multi-layer network, by repeatedly applying the chain rule of derivatives. This procedure is a form of *supervised learning* since it relies on the existence of *labeled* training data.

SGD trains a neural network by minimizing a *loss function*, which represents the deviation of a network's predictions from the actual labels of the employed training data. This mechanism can be tuned by a number of so-called *hyper-parameters*. The *learning rate*, for example, determines the amount by which the gradients are updated during training. On the one hand, lower learning rates can generally lead to better results but suffer from long training duration. High learning rates, on the other hand, decrease training time but might result in poor accuracy of the network.

Training is typically concluded by evaluating the performance of the trained model on a separate dataset, referred to as *test set* or *validation set*. This dataset does not overlap with the training data and only contains samples that have not been presented to the model during training. This helps to estimate its accuracy when applied to unseen input data. Deep learning can also be parallelized in order to speed up the training process [PSY+18]. Different techniques have been proposed to decrease training time by distributing deep learning workloads. With data parallelism, for example, the model to train is replicated on multiple distributed hosts, together with a subset of the training data.

2.1.1 Types of Neural Networks

From the earliest works on pattern recognition, such as Rosenblatt's perceptrons [Ros57], to state-of-the-art deep learning methods, a vast number of different neural network architectures have evolved, which differ in their complexity as well as their application scenarios. On a high level, these different architectures may be classified as either *feed-forward* or *recurrent* networks [Sch15].

Feed-forward Neural Network (FNN)

FNNs are characterized by their unidirectional flow of information between their layers. Figure 2.1 illustrates the structure of a simple FNN consisting of an input layer, followed by a hidden layer, and a final output layer. A typical class of FNN architectures, that have become popular especially in the area of image classification, are *Convolutional Neural Networks (CNNs)*, also referred to as *ConvNets*, which process array-shaped input data [LBH15]. For example, they may be used for processing audio signals that come in the form of 1D arrays, or color images that could come in the form of multiple arrays, e.g., as one array per color channel and pixel.

CNNs have their roots in the so called *Hubel-Wiesel Architecture* [HW62] which is based on the structural principles of the cat's primary visual cortex. In their work, Hubel and Wiesel identified two types of cells — *simple cells* and *complex cells* which correspond to convolutional and pooling layers respectively. One of the first simulations of this model on a computer was the Neocognitron by Fukushima et al. in 1983 [FMI83].

Vast improvements in this domain have become possible due to the availability of a large corpus of training datasets such as Cifar [KH+09, Kri12] or ImageNet [FFDL10]. Well

known CNN architectures are AlexNet [KSH12], GoogLeNet [SLJ+15], VGG [SZ15], and ResNet [HZRS16].

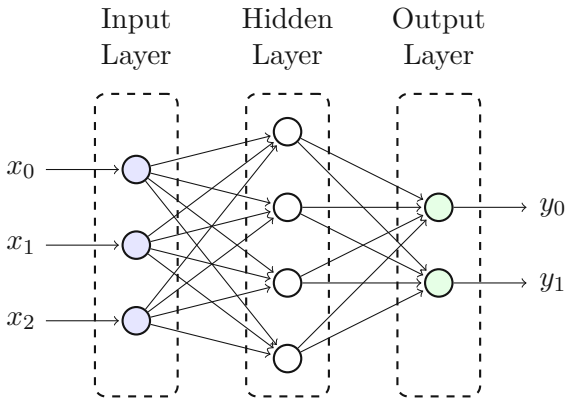


Figure 2.1: Fully connected FNN with a single hidden layer

Recurrent Neural Network (RNN)

Another class of neural networks, which are characterized by their recursive structure, are *recurrent neural networks (RNN)* [RHW86, CVMG+14, PGCB13]. They have become a popular choice in the domain of natural language processing as well as speech recognition.

Formally, a RNN simulates a discrete-time dynamic system that takes inputs $\mathbf{x}_i$, produces outputs $\mathbf{y}_i$, and has a hidden internal state $\mathbf{h}_i$, where t denotes a timestamp [PGCB13]:

$$\mathbf{h}_t = f_h(\mathbf{x}_t, \mathbf{h}_{t-1}) \tag{2.1}$$

$$\mathbf{y}_t = f_o(\mathbf{h}_t) \tag{2.2}$$

f_h denotes a state transition function and f_o denotes an output function. These functions are parameterized by the parameters θ_h and θ_o respectively, whose values are learned during training. Intuitively, a RNN predicts an output value at time t, based on an input value and its remembered state at time $t-1$.

2.1.2 Types of Layers

The rise of different neural network architectures in deep learning induced an increase in diversity of their internal structure as well. In particular, different types of layers have emerged, in order to improve accuracy or to reduce training time. Since this work is focused on feed-forward CNNs, in the following, we give a short overview over the most important kinds of layers that are used in CNN architectures in particular [KHG+17].

Fully Connected Layer

In a fully connected layer, each neuron is connected to each neuron of the previous layer. The output of such a layer is a weighted sum of its inputs, which is parameterized by a weight matrix and a bias, that are learned during the training phase. Typically, these layers are used at the later stages of DNNs to produce the final classification result.

Convolutional Layer

As opposed to fully connected layers, each neuron in a convolutional layer is only connected to a sub-set of the neurons in the previous layer. In mathematical terms, the operations of these layers correspond to *discrete convolutions* [LBH15].

Pooling Layer

Pooling layers group the outputs of their preceding layer, which typically results in a size reduction of the intermediate results. There are different types of pooling, e.g., *max pooling* and *average pooling*, which are characterized by the function that is used for the grouping. The pooling operations are applied to subsets, or *pooling regions*, of the preceding layer's outputs. These regions are defined in terms of their size and shape, and might also overlap based on a chosen stride. [KHG+17]

Activation Layer

An activation layer proceeds by applying a *non-linear* function to each of its inputs. Hence, the number of outputs of these layers corresponds to the number of its inputs. Traditionally, non-linearities such as *tanh* or the *sigmoid* function $f(x) = (1 + e^{-x})^{-1}$ have been used for these layers. Recent research uses the *rectified-linear unit* ($ReLU$) function which is basically a half-wave rectifier $f(x) = max(0, x)$. The use of $ReLU$ as activation function can also lead to significantly faster training compared to *tanh* [KSH12].

BatchNorm Layer

Batch normalization has been introduced to dramatically decrease the training time of DNNs [IS15]. Training time in deep learning is influenced by the learning rate — a hyper-parameter which determines the amount by which a model's weights are adjusted during training. However, learning rates that are set too high could induce exploding or vanishing gradients and might result in gradient descent getting trapped at poor local minima. Including batch normalization layers can prevent these problems and thereby enable higher learning rates, which in turn leads to faster training.

2.2 Neural Networks with Multiple Exits

Recent research also proposes new network architectures that facilitate the distributed execution of DNN inference. Instead of using only one classifier at the final network layer, these architectures introduce additional side-exit classifiers at intermediate layers that allow to obtain inference results at earlier points in the network, which is illustrated in Figure 2.2. Each of these classifiers is associated with a configurable confidence threshold that determines whether the classification result for a given input sample is sufficiently accurate to stop inference at this side-exit. In the context of a distributed DNN, this architecture allows to retrieve prediction results at earlier compute nodes that are closer to the end devices.

Originally, first versions of this architecture were proposed to tackle a phenomenon called "overthinking" [KHD19, WLC+17, LZQ+19]. This concept is actually inspired by an analogy to *human overthinking* which occurs when a person pays attention to irrelevant details and puts more thought into a decision problem than actually necessary. This slows down the thought process and oftentimes might also lead to bad decisions. In terms of neural networks, a model is overthinking on an input sample, if the simpler representations at earlier layers would already be sufficient to do an accurate classification. Further computations at later layers would therefore be considered as wasteful and in the worst case might actually lead to misclassifications and a decreased accuracy at the final classifier.

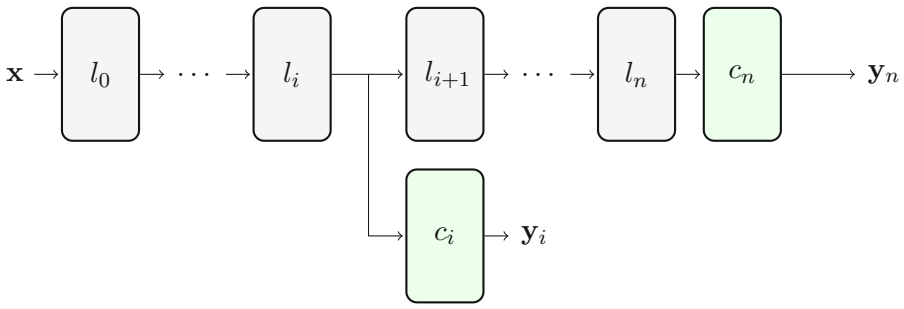


Figure 2.2: Neural network with multiple exit classifiers

With BranchyNet [TMK16], Teerapittayanon et al. propose a concrete neural network architecture that follows this concept. The classifiers in BranchyNet are trained jointly by minimizing the weighted sum of their loss functions. At inference time, each early exit is equipped with a threshold value that defines the minimal level of confidence for a test sample in order to leave the network at this point. An evaluation of BranchyNet based on state-of-the-art architectures such as LeNet, AlexNet, or ResNet demonstrates that its design can reduce the overall inference time, increase energy efficiency, and improve classification accuracy of the network.

In subsequent work, Teerapittayanon et al. seize the concepts that are introduced with BranchyNet for efficient and accurate inference in *distributed deep neural networks*

(DDNNs) [TMK17]. In a DDNN, the layers of a neural network are mapped over a distributed computing hierarchy formed by cloud resources, edge (fog), and end devices. Instead of performing inference in a centrally deployed cloud model, this allows for the execution of a subset of DNN layers directly on the end devices or at the edge of the network. The placement of early exit classifiers allows to obtain prediction results at earlier points in this hierarchy without having to send all input data to the cloud. This reduces the cost of communication between the nodes in this distributed computing hierarchy and may also improve data privacy and fault tolerance.

2.2.1 Training Approaches

The idea of adding early exits to a neural network is also addressed by Baccarelli et al. who contribute a formal characterization of this type of network architecture [SSBU20] and explore approaches to combine this type of DNN with edge (fog) computing in the context of IoT applications [BSS+20]. In addition to the work of Teerapittayanon et al. [TMK16, TMK17] they identify three training algorithms.

Joint Training

With this approach, all the layers of a network, including all side-exit classifiers and the final classifier, are trained at the same time. In particular, the back-propagation now has to consider the classification results of multiple exits simultaneously. This means, the losses of all exits are combined during back-propagation and weights are updated in a joint-optimization approach. This training approach is used by GoogLeNet [SLJ+15] and BranchyNet [TMK16].

Layer-wise Training

With *layer-wise* training, individual layers and their corresponding local exits are trained separately, starting from the input layer. The learned weight matrix of already trained layers are frozen while the training procedure continues to the next layers.

Classifier-wise Training

Classifier-wise training follows a more decoupled approach that might also be parallelized. After first learning the weight matrices of the baseline DNN, the corresponding weight matrices of the exit classifiers are obtained independently from each other. As described later in section 7.3, this is the training approach that we adopt for producing example models for the experimental studies of this work.

2.2.2 Inference

Algorithm 2.1 outlines an inference mechanism for sequential feed-forward networks, taking into account classification results that are obtained at earlier layers.

Algorithm 2.1: Inference in a multi-exit neural network

Input: Tensor $\mathbf{x}$
Result: Class label
1 **for** $i \leftarrow 0$ **to** $N-2$ **do**
2 $\mathbf{x} \leftarrow l_i(\mathbf{x})$;
3 **if** *layer i has a local exit* **then**
4 $\mathbf{y} \leftarrow c_i(\mathbf{x})$;
5 $\mathbf{s} \leftarrow softmax(\mathbf{y})$;
6 $e \leftarrow \eta(\mathbf{s})$;
7 **if** $e \leq \mathbf{T}_i$ **then**
8 **return** $argmax(\mathbf{y})$
9 **end**
10 **end**
11 **end**
12 $\mathbf{x} \leftarrow l_{L-1}(\mathbf{x})$;
13 $\mathbf{y} \leftarrow c_{L-1}(\mathbf{x})$;
14 **return** $argmax(\mathbf{y})$

After obtaining the output of a layer on line 2, the inference mechanism does not proceed to the next layer immediately. Instead, it checks whether a side-exit classifier c_i is assigned to the current layer l_i and obtains the corresponding classification result on line 4. Next, the *softmax* function is used to normalize the classification result. On line 6 the normalized entropy of the classification result is computed. The normalized entropy $\eta(\mathbf{x})$ of a vector $\mathbf{x} \in \mathbb{R}^{|C|}$, where C is the set of class labels, is defined as follows:

$$\eta(\mathbf{x}) = -\sum_{i=1}^{|C|} \frac{x_i \log x_i}{\log |C|}, \tag{2.3}$$

Informally, this entropy is a measure of confidence that characterizes how "certain" the network is about the result of a certain classifier. Each exit classifier c_i is equipped with a user-specified confidence threshold T_i. If the entropy of the corresponding classification result does not exceed this threshold, i.e. $\eta(\mathbf{x}) \leq T_i$, the network is confident about the classification. In that case, inference stops at that layer and returns the side-exit's classification result.

Finally, if no side-exit was taken at the hidden layers of the network, the algorithm leaves the loop to compute the final classification at the uppermost layer of the network and returns the corresponding class label.

2.3 DNN Model Partitioning

In traditional AI-backed applications, DNNs are hosted as centralized services in the cloud. With this model of execution, all computations are offloaded to a — potentially far-away — cloud-server. This can impose high network traffic, since all input samples have to be transferred to the cloud. With the emergence of more lightweight DNN architectures, such as MobileNet [HZC+17, SHZ+19, HSC+19], as well as lightweight DNN framework implementations, such as *TensorFlow Lite*[1] or *PyTorch Mobile*[2], DNN execution on end-devices has become a viable alternative.

However, instead of (i) completely offloading DNNs to the cloud or (ii) hosting a full network directly on the end- or edge-device, a third model of execution has evolved which involves partitioning the computations of a DNN model over multiple hosts. Concretely, if a DNN is composed of a sequence of N layers and its i^{th} layer is chosen as partitioning point (also referred to as *split point*), then the first i layers would be executed on the end device (or on an egde-server that is located in near proximity to the end-device), whereas the last $N - i$ layers would be executed on the cloud server. This means instead of sending all the input data to the cloud, only the intermediate result of the i^{th} layer would be sent to the cloud.

The decision to partition a DNN at a certain layer might be based, for example, on the desire to reduce communication cost between the edge and the cloud server. This involves scanning the DNN in order to find a layer that performs an appropriate level of compression. For example, in practice, pooling layers often reduce the size of its inputs. A DNN might then be partitioned in a way, that places all layers before such a compression layer on the end-device and the remaining layers would be executed in the cloud. Hence, instead of sending the raw input data, only the compressed intermediate results after the partitioning point would have to be sent to the cloud.

Alternatively, for DNN architectures with side-exits, as described in the previous section, a model could be partitioned after an early layer that provides a side-exit classifier. This means, for some input samples it might be possible to obtain a classification result directly on the device. Hence, this approach would enable a further reduction of the communication overhead with a cloud service.

In addition to the reduced communication cost, partitioning could also improve the privacy of an AI-application [MDK+20]. For example, in an image classification scenario, if an end-device (e.g., a camera-device or a smartphone) executes parts of the partitioned DNN model locally, it would not have to send raw image data to a cloud-service, that is hosted by a third-party.

[1] https://www.tensorflow.org/lite/
[2] https://pytorch.org/mobile/

2.4 Edge and Fog Computing

In the last decades, cloud computing has become a dominant paradigm in the computing landscape which had a major impact on how companies organize their IT infrastructure. The NIST defines the notion of *cloud computing* as follows [MG+11]:

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

Instead of operating their own private data-centers, organizations shifted their IT operations to these centralized cloud data centers that are operated by a few large service-providers. This way, they benefit from economies of scale due to the decreased marginal costs of this shared infrastructure. Hence, this dominance of cloud computing resulted in a major consolidation of compute resources into large data centers that are spread across the globe [Sat17].

In opposition to these forces of centralization, the decentralized nature of the rising *edge computing* paradigm directs computation and data away from centralized servers and instead shifts them towards the end-user. Edge computing has its roots in *content delivery networks (CDNs)*, that were introduced by Akamai [DMP+02] at the end of the 90s, in order to accelerate web performance [Sat17]. In a CDN, hosts that are in near proximity to the user, prefetch and cache web content to achieve faster content delivery. Edge computing is a generalization of CDNs that extends the original concept by integrating cloud infrastructure. However, while CDN nodes are limited to cache web content, edge nodes can now be used to run arbitrary code, similar to cloud nodes. Various terms have been shaped to refer to these edge resources, such as *cloudlets* [SBCD09], micro data centers, or fog nodes [BMZA12]. The term *fog computing* was introduced by Bonomi et al. [BMZA12], however, oftentimes the notions of edge computing and fog computing are used interchangeably [SCZ+16].

2.4.1 Characteristics

Edge and fog computing share the basic characteristics of cloud computing. In addition, however, they are identified by the following key aspects [KAH+19].

Geographical Distribution

In contrast to cloud services, that are hosted in a centralized manner, edge and fog services are characterized by deployments that might span geographically distributed infrastructure.

Low Latency

With the availability of compute resources in near proximity of the end-devices, computation and services are shifted towards the user. This improves the end-to-end latency of applications which is required in scenarios such as augmented reality and self-driving cars. [KAH+19].

Heterogeneity

Fog and edge computing are characterized by heterogeneity in multiple aspects. Nodes might either be operated fully virtualized or on dedicated hardware platforms that range from powerful servers to lightweight sensor devices with highly diverse compute capabilities and resource constraints [HDNQ17]. Heterogeneity also regards networking capabilities, which might cover high-speed internet uplinks as well as wireless connections such as Wi-Fi, 3G, 4G, etc. [BMNZ14].

2.4.2 Challenges

The emergence of these new computing paradigms also introduces several challenges [SD16, SCZ+16].

Programmability

In cloud computing, the underlying infrastructure is usually transparent to the user [SCZ+16]. Typically, programmers write their applications for a single target platform and simply deploy them to the cloud, relying on the cloud provider to decide how and where the compute workload is run. In the area of heterogeneous fog environments, however, programmers often still are required to partition their applications manually between devices, the edge, and the cloud. The heterogeneity of hardware and software platforms has to be addressed explicitly during implementation. In order to improve scalability and extensibility in this regard, stream-lined frameworks and tool-chains are needed [SD16].

Privacy and Security

With the rise of IoT and smart-home devices, a lot of highly sensitive data can be learned by attackers [SCZ+16]. In order to improve security and privacy at the edge of the network, several aspects need to be considered. First, all stakeholders — such as service providers, developers, and end-users — need to be aware of the importance of appropriate privacy and security measures. For example, unsecured Wi-Fi networks or default passwords on routers account for significant attack vectors that can easily be closed by an improved awareness for those issues. Furthermore, data protection can be improved by granting end-users advanced control over their data. By ensuring that data resides on devices and only selected information is sent to service providers, leakage of sensitive information to third parties can be avoided. Additionally, edge nodes might

perform automatic anonymization to remove private data that should not be shared. Finally, edge computing is lacking efficient tools for data protection and privacy on edge nodes. Suffering from constrained resources on some edge platforms, certain security methods might not be applicable due to their high demand in computing power.

Naming and Standardization

The terms and definitions for these new computing paradigms are not standardized. As mentioned before, fog computing and edge computing are oftentimes used interchangeably, while some research classifies them as different yet related paradigms. Currently, efforts for standardization are driven by organizations such as the *OpenFog Consortium*, *OpenEdge Computing*, and *ETSI*, which standardizes *multi-access edge computing (MEC)* [YFN+19, ETS20].

CHAPTER 3

Related Work

In the following sections, we give an overview of recent research that is closely related to the work of this thesis. In particular, we discuss frameworks and system designs that target co-inference over distributed device-to-cloud settings via the use of model partitioning and DNN architectures with early exit branches. The presented works are discussed in order of their year of publication.

3.1 Neurosurgeon

One of the earliest works proposing an end-to-end framework for distributed DNN inference is Neurosurgeon [KHG+17]. The framework architecture spans (i) a deployment stage and (ii) a runtime system. At the deployment stage, an application-independent profiling of the employed DNN model is done on an end-device and a server node. The profiling output is used to generate prediction models for the actual layer performance. It uses an estimation based technique to determine per-layer latency and energy consumption. Each layer's computation latency is estimated based on the FLOPs of its computations. The runtime system dynamically decides to repartition the DNN. Based on the obtained profiling information, it estimates the performance and energy consumption of each layer on the end-device and in the cloud. Together with the current network bandwidth and the load level of the server, the runtime system will dynamically decide the split point. This is done to either optimize for end-to-end latency or for best mobile energy consumption.

Neurosurgeon's system runtime consists of (i) a component that resides on the end-device (`NSmobile`), and (ii) a component that runs on the server-side (`NSserver`). The implementation of the runtime components is based on the DNN framework *Caffe* [JSD+14]. Communication between device and server uses *Thrift* [SAK07] for RPC. Both the device and the server runtime host the whole DNN model. Upon receiving an inference request, the device runtime `NSmobile` executes all layers preceding the partition point. It then sends the intermediate output at the partitioning point to the server

runtime `NSserver` which then executes the remaining layers and sends back the inference result to the device runtime.

An evaluation of Neurosurgeon on a set of different DNN applications shows clear improvement over the status-quo (i.e. cloud-only DNN inference). On average, end-to-end inference latency decreases by a factor of 3.1, mobile energy consumption is reduced by 59.5%, and datacenter throughput is increased by a factor of 1.5.

3.2 DNN Surgery

In the same line of work as Neurosurgeon, Hu et al. [HBWL19] propose dynamic adaptive *DNN Surgery* — a framework with different schemes for partitioning DNN inference on the edge and in the cloud. Whereas Neurosurgeon is only applicable to chain-topology DNNs, DNN Surgery also targets DNNs with a DAG topology, such as Resnet [HZRS16] or Googlenet [SLJ+15].

The proposed design covers two modes of execution, that target two different types of workload settings. The first mode of execution, *DNN Surgery Light (DSL)*, is designed for the operation under *light* workloads. It assumes that each classification request can be completed before the next classification request arrives. Its goal is to minimize the end-to-end inference latency of a single input sample (e.g. of a single frame in a video). The authors characterize this mode as equivalent to the graph theoretical *min-cut* problem. A second mode of execution, *DNN Surgery Heavy (DSH)*, is tailored for the operation under *high* workloads. With this mode, it is possible that a classification request arrives while other classification requests are still being processed. In contrast to the DSL mode, the goal of DSH is to maximize the overall inference throughput, i.e., it tries to maximize the number of input samples that can be classified in parallel. The system continually monitors the network conditions and adapts its partitioning choice dynamically, depending on the chosen mode of execution.

The authors evaluate the performance of DNN Surgery on a video dataset for self-driving cars on a test-bed with simulated network conditions based on recorded traces of a wireless network. When compared to cloud-only inference (i.e. every image is sent to the cloud server), DNN Surgery decreases inference latency by a factor of 6.45. Compared to device-only execution, inference latency is decreased by a factor of 8.08. Likewise, inference throughput is increased by a factor of 8.31 and 14.01 respectively.

The experimental study also includes a comparison between DNN Surgery and Neurosurgeon. The comparison shows that the two systems achieve similar latency and throughput levels for chain-topology DNNs on a light workload setting.

3.3 ADDA

While prior work focuses on model partitioning for distributing inference, Wang et al. [WCHD19] present a combined approach, that integrates DNN architectures with early

exits to improve DNN performance in edge computing environments. Their framework for *adaptive distributed DNN inference acceleration (ADDA)* aims at minimizing total inference latency while ensuring accuracy. The proposed framework architecture covers two stages: (i) an *offline* training and deployment stage and (ii) an *online* partitioning and offloading stage.

The aim of the first stage is to find the best set of side-exits for a given DNN model architecture. Some exit paths have the same capacity, in terms of inference accuracy, despite of their difference in length. Furthermore, adding too many exits introduces computational overhead. To tackle this, Wang et al. propose a *best-N exit selection* algorithm to nominate only the best N exits. This process consists of two steps. In a first step, the exit rate and inference time for each exit is collected, as well as the joint inference time for the whole network. In a second step, the algorithm aims to find the best N exits that minimize the total inference time in the given DNN model.

The second stage of the framework design covers an online computation of an optimal model partition. In order to estimate the total latency of an inference request, this stage provides an estimate of computation and transmission cost of the DNN layers. Computation cost is determined by the exit probability of each side-exit and load-level of end-devices and edge servers. Transmission cost is determined by network bandwidth and intermediate data between layers at split points. The result of this stage is a partition that distributes DNN inference between an end-device and an edge server.

A reference implementation of the proposed design is done in Python, based on the DNN Framework PyTorch. Communication between the components is done via message queuing, using ZeroMQ[1]. The performance of the system implementation is evaluated for the modified versions of the DNN architectures *AlexNet* [KSH12] and *VGG-16* [SZ15] on the CIFAR-10 dataset [KH+09] for image classification. The experimental setup comprises a physical testbed with a Raspberry 3B+ (as end-device) and a PC with a GPU platform (as edge server), connected via a Wi-Fi network. For different network conditions and different levels of server load, ADDA shows a speedup, in terms of end-to-end inference latency, of up to 6.6× compared to the status-quo (i.e. device-only and server-only DNN execution).

3.4 Couper

With Couper [HBG19] Hsu et al. also propose a framework for model partitioning over distributed computing hierarchies. However, in contrast to other works in that field, that focus on algorithmic aspects and optimality of partitioning mechanisms, the emphasis of Couper is the exploration of software-architectural aspects as well as system-level support for automatic model partitioning (also referred to as *slicing* in the context of this work) and deployment. In particular, this work focuses on the partitioning of DNNs for visual

[1] https://zeromq.org/

analytics applications and deployment of DNN partitions in container-based software stacks in edge computing environments.

At its core, Couper's architecture comprises four main components that are operated in a Kubernetes cluster: (i) a *model slicer* that finds all candidate split points for a given DNN model according to a user-defined algorithm, (ii) an *application wrapper* that compiles Docker container images based on the identified candidate split points, (iii) a *slice evaluation* that evaluates the candidate split points and selects the best split point according to a user-defined metric, and (iv) a *publisher* that prepares and deploys corresponding Docker containers for the chosen split point.

Finally, the proposed system design assumes a separate staging area, that runs all of Couper's components in a Kubernetes cluster, as well as a production environment (also running Kubernetes), that comprises the target infrastructure to host the final DNN partitions and for carrying out the DNN inference tasks. Furthermore, the reference implementation of Couper is based on TensorFlow and the SAF streaming framework[2].

3.5 Edgent

Similar to ADDA [WCHD19], Edgent [LZZC20] also aims at optimizing collaborative DNN inference between an end-device and an edge server, by means of combining DNN model partitioning with early exit architectures. In particular, the framework seeks to determine (i) a partition point and (ii) an early exit, so that the accuracy of the chosen DNN configuration is maximal, while respecting a user-specified end-to-end latency requirement. However, in contrast to ADDA, Edgent does not employ tuning of confidence thresholds to control the exit rates of the DNN's side exit. Instead, for a given DNN model with multiple exit branches, it will decide to activate only a *single* exit, that maximes accuracy under the given latency constraints, while the remaining side exits will be deactivated.

The design proposes two modes of execution to target different types of network conditions. For both modes of execution the proposed framework proceeds in three stages: (i) an offline configuration stage that trains the multi-exit DNN and collects profiling information, (ii) an online tuning stage that decides on the partition point and exit branch to take, and (iii) the co-inference stage that actually performs DNN inference over the device and the edge server.

The first mode of execution targets scenarios with relatively stable network conditions, where changes in available bandwidth only occur slowly over time. In this mode, the offline configuration stage trains regression models for predicting the per-layer inference latency of a given DNN model. These regression models are obtained by profiling the inference latencies for different types of layers both on the end-device as well as on the edge server. Furthermore, the offline configuration stage also covers the actual training of the DNN model with multiple exit points. During the online tuning stage, the partition

[2] https://github.com/viscloud/saf

and exit points will then be chosen based on the trained prediction models, using the current network conditions and the user-specified latency requirements as inputs.

The second mode of execution targets scenarios with dynamic network conditions. Here, bandwidth changes are expected to occur frequently and drastically. For this purpose, Edgent relies on recorded traces of bandwidth which are used during the offline configuration state to pre-generate execution policies for known network conditions. At run-time, the online tuning stage then applies the prepared policies based on the current network conditions.

A proof-of-concept implementation of Edgent is done on the basis of BranchyNet [TMK16] which in turn is based on the DNN Framework Chainer[3].

3.6 SPINN

SPINN [LVA+20] is a system for distributed DNN inference guided by multi-objective user-specified requirements. Simlar to ADDA [WCHD19] and Edgent [LZZC20] it relies on a combination of model partitioning and DNN architectures with early exits. The design of SPINN covers an offline as well as an online stage, spanning the following core components: (i) a *progressive inference model generator*, (ii) a *model splitter*, (iii) a *profiler*, (iv) a *dynamic scheduler*, and (v) an *execution engine* that implements DNN inference.

In a first stage, an offline progressive inference model generator decides about an appropriate placement of exit classifiers on a given DNN backbone architecture. The resulting DNN model is then trained by employing a joint-training approach that is similar to the one proposed by BranchyNet [TMK16] (see Section 2.2.1). Even though SPINN proposes a mechanism to place exit branches on a given DNN architecture, the described prototype implementation does not seem to include an *automated* process to generate and train multi-exit DNN models.

Next, the model splitter analyzes the underlying execution graph of the model to identify all possible split points in the given DNN model. In order to restrict the search-space, SPINN only considers ReLU layers as candidate split points.

SPINN's profiler operates in an offline as well as in an online stage. During the offline stage, the profiler determines device-independent attributes such as size of transmitted data at candidate split points, exit-rates and accuracy of the DNN's side-exits for different confidence threshold configurations. Also, initial device-dependent estimations are performed prior to deployment, e.g., to determine the execution time of each layer. Later, at run-time, the profiler improves its initial estimates on the basis of the system's monitored state, such as measured network conditions or server load.

At its core, SPINN operates a scheduler that decides the model partition and early exit policy based on the outputs of the profiler. The scheduler supports a number of metrics

[3] https://chainer.org/

(inference latency, throughput, server cost, device cost, and accuracy) that can either be used as part of a set of soft optimization targets (e.g., maximizing throughput) or as part of a set of hard constraints (e.g. a maximally allowed latency, or minimal level of accuracy). At run-time, the scheduler is triggered whenever the output of the profiler changes by a factor that exceeds a predefined threshold. The dynamic scheduler runs on the *end-device*, hence, in the case of less powerful end-devices, it is required to limit its resource utilization accordingly.

The execution engine handles classification requests and orchestrates inference of the partitioned model on the device and the server. At this point, SPINN also uses the intermediate results of early exits to improve fault tolerance during execution of co-inference. In particular, the scheduler ensures that one exit is always present on the end-device. This guarantees, that an inference result will always be available on the end-device even if the cloud server is unreachable. If the server is not available (e.g., due to a network failure) the execution engine can then fall back to the classification result at earlier exit branches at the price of reduced accuracy, instead of aborting inference altogether.

A prototypical implementation of the proposed design is based on PyTorch. In an experimental study, the prototype's performance is evaluated in comparison to related research, in particular, Neurosurgeon [KHG+17] and Edgent [LZZC20]. Compared to these systems, SPINN achieves an increase in inference throughput by a factor of 2, while reducing server cost by up to 6.8× and increasing accuracy by up to 20.7%.

3.7 EdgeML

Zhao et al. introduce EdgeML [ZWLX21], which also combines model partitioning with progressive DNN architectures to maximize inference accuracy, while respecting user-specified bounds on task latency and end-device energy consumption. The design of EdgeML proceeds in three stages: (i) model transformation, (ii) dynamic model execution control, and (iii) the actual execution of DNN inference over an edge device and a cloud server.

During the offline model transformation stage, the given DNN model will be adapted by inserting exit branches and possible split points will be identified. Exit branches and candidate split points are placed at equidistant locations in the underlying model. The inserted exit branches only consist of fully-connected layers, with decreasing depth across the neural network, i.e., earlier side branches contain more fully-connected layers while later side exits contain less layers.

At runtime, the dynamic model execution control stage utilizes a reinforcement learning (RL) based approach, to decide about an optimal *execution policy*. An execution policy consists of the particular split point for the model partition and confidence thresholds for the exit branches. In contrast to SPINN [LVA+20], that uses a *single* confidence threshold for *all* exit branches, EdgeML assigns separate confidence thresholds to each

exit branch. The prototypical source code used for the experiments has been published on github[4].

3.8 Feature Comparison

Framework	Year of Publication	DNN Framework	Communication	Model Partitioning	Early Exit DNNs	Exit Threshold Tuning	Runtime Adaptivity	Open Source	APIs/Extensibility
Neurosurgeon	2017	Caffe	Thrift	✓			✓		
DNN Surgery	2019	Caffe	gRPC	✓			✓		
ADDA	2019	PyTorch	ZeroMQ	✓	✓				
Couper	2019	TensorFlow	SAF	✓					
Edgent	2020	Chainer	n/a	✓	✓		✓		
SPINN	2020	PyTorch	n/a	✓	✓	✓	✓		
EdgeML	2021	TensorFlow	n/a	✓	✓	✓	✓	✓	
This Work	2022	PyTorch	gRPC	✓	✓		✓	✓	✓

Table 3.1: Feature comparison of distributed DNN inference frameworks

Table 3.1 outlines a comparative overview of the discussed works. With respect to implementation aspects, the table details the employed DNN frameworks as well as communication technologies. Note that Caffe as well as Chainer are now part of PyTorch, which, together with Tensorflow, accounts for the most important framework in the area of deep learning. While DNN model partitioning is employed by all of the presented works, the adoption of early-exit architectures accounts for major differences. Early exits are supported by ADDA , Edgent, SPINN, and EdgeML, the tuning of thresholds however, is only addressed by SPINN and EdgeML. Even though our work does not employ threshold tuning at the moment, this mechanism can easily be integrated by means of (i) extending the existing APIs to support the configuration of threshold levels and (ii) providing algorithms to tune these thresholds based on application-specific requirements. While the work on Couper [HBG19] provides the most comprehensive study addressing systems aspects, it lacks an implementation that is available as open-source. However, as can be seen in Table 3.1, this property is also shared by most of the other studies. In contrast to our work, Couper is dependent on the containerization of its components, which might conflict with resource limits when employing the system on more constrained devices. Furthermore, another aspect that sets our work apart, is its extensibility based on comprehensive APIs, which cover all aspects of managing, orchestrating, and executing a DNN on top of a device-to-cloud compute hierarchy. This API-driven design, based on the

[4] https://github.com/Kyrie-Zhao/EdgeML/tree/master

open-closed principle, allows for a high degree of modularity and facilitates independent implementation, replacement, and operation of individual components.

In summary, recent research is mainly focused on algorithmic aspects for model partitioning and tuning of progressive DNN architectures. To the best of our knowledge, apart from Couper, our work provides the most comprehensive study on systems and implementation aspects in this line of research.

3.9 Other approaches for distributed inference

In contrast to the above approaches that investigate compute hierarchies spanning the edge-to-cloud continuum, Hadidi et al. [HCRK20] propose a method that uses a collaborative network of IoT devices to perform in-the-edge DNN inference in a collaborative manner. With a focus on visual computing applications, they present methods to distribute the computations in a convolutional neural network across different compute nodes. The proposed techniques are based on *model parallelism* to exploit the independence of certain intra-layer computations. DeepThings [ZBG18] employs a similar approach to distribute DNN inference over a cluster of resource-constrained IOT devices.

CHAPTER 4

System Design

One contribution of this thesis is the design of a system for the distributed execution of DNN inference on top of a computing hierarchy that consists of the cloud, edge, and end devices. The system should support the deployment of an already trained DNN model over a network of distributed compute resources and provide means for monitoring and runtime management of the deployed DNN. This chapter gives an overview of a generic system design and its key characteristics. It intentionally omits implementation details. A concrete implementation of the proposed design will be covered separately in Chapter 5.

The remainder of this chapter is organized as follows. Section 4.1 defines the relevant stakeholders of the system. The requirements that need to be met by the proposed system design and its implementation are introduced in Section 4.2. Finally, Section 4.3 introduces the proposed system architecture that adheres to these requirements.

4.1 Stakeholders

In the following, the key roles, that are involved when operating the system, are identified. Depending on the specific application scenarios, these roles may overlap or may be executed by the same entity.

- **Model Developer/Owner:** The model developer is responsible for the design of a DNN architecture as well as the implementation of the chosen architecture for a concrete DNN framework technology. Furthermore, this role comprises the training of the implemented DNN on a dataset.
- **Infrastructure Provider:** The infrastructure provider is responsible for (i) the maintenance of the compute infrastructure, (ii) the operation of the system's components, and (iii) providing the trained DNN model to the system.

- **Data Provider/End-User:** In case the application scenario involves devices that are operated by an end-user (e.g. a smartphone owner), this role is responsible for generating the input data (e.g. images captured by a smartphone camera, voices that are recorded by a smartphone's microphone, etc.). Furthermore, since parts of a DNN model might be deployed to the end-device that serves the input data, this role may also assume parts of the functionality of an edge infrastructure provider.

4.2 Requirements and Desirable Properties

4.2.1 Operational Environment

The system should operate in an environment that is characterized by a distributed compute hierarchy consisting of the following vertically organized tiers:

- An *end-device* or *IoT tier* that consists of sensors or other kinds of devices that are connected to a source of input data. This tier typically suffers from resource constraints leading to restricted computing power or network bandwidth.
- One or multiple *edge* (or *fog*) *tiers* which can include servers or small data centers.
- A *cloud tier* that represents the final tier whose resources typically account for a major part of the available computing power in this hierarchy.

In addition to this vertical dimension, each of those tiers in turn may comprise a horizontal dimension covering multiple compute nodes. Furthermore, exactly one node, that is part of the end-device tier, is assumed to act as *input source* that generates the data to be processed by the DNN.

4.2.2 DNN architectures with early exits

The system should provide means to define and deploy DNN models that include multiple exit classifiers, as described in Section 2.2.

4.2.3 Model Partitioning

Instead of deploying and assigning DNN models to a single compute node, the system's placement mechanism should allow for a more-fine grained assignment. In particular, placement decisions should be made on a per-layer basis and possibly partition a model over multiple compute nodes, as described in section 2.3.

4.2.4 Runtime APIs

The system should provide APIs for the purpose of orchestration of system components that are deployed in its environment. In particular, this includes interfaces and application-layer protocols for:

- deployment as well as dynamic redeployment of DNN components to nodes in the compute hierarchy (e.g., for deployment of DNN layers and their dynamic migration between compute nodes),
- configuration of system properties such as confidence thresholds at exit classifiers,
- exchange of intermediate results of DNN inference between layers (e.g., for communicating outputs of intermediate DNN layers to upper DNN layers, e.g., *tensor* data in the case of TensorFlow or PyTorch),
- monitoring of system state and operational conditions of system components (e.g., CPU/GPU load, memory utilization, battery level, available bandwidth).

4.2.5 Interoperability with state-of-the-art DNN frameworks

The system should integrate with at least one state-of-the-art deep learning framework. Prominent frameworks that are employed by current research include *TensorFlow*,[1] which was initially introduced by Google, and *PyTorch*,[2] which is developed at Facebook.

4.2.6 Extensibility

From a software architecture perspective, the system should provide a framework/APIs for implementing different deployment, resource allocation, and adaptation strategies. This way, different optimization goals can be pursued.

4.2.7 Adaptivity

The system should aim for dynamic redeployment and reconfiguration of DNN components in response to changing operational conditions. At its core, such a mechanism would be driven by agents that operate either centrally or at each host.

The observable environment of such a mechanism would include the state of compute nodes (e.g., battery level, CPU load, available bandwidth, etc.). An agent interacts with its environment and aims to take the best decision at any given time according to its objectives. The action space is determined by the system APIs and could include, for example, migration of layers between hosts and the adaption of confidence thresholds at local exits. In the presence of an appropriate feedback signal, the agents might also apply reinforcement learning to learn the efficacy of decisions in order to tune their behavior.

4.3 Architecture

An overview of a generic system architecture, based on the requirements outlined above, is illustrated in Figure 4.1. In the following sections, we introduce the key components that are parts of the system.

[1] https://www.tensorflow.org/
[2] https://pytorch.org/

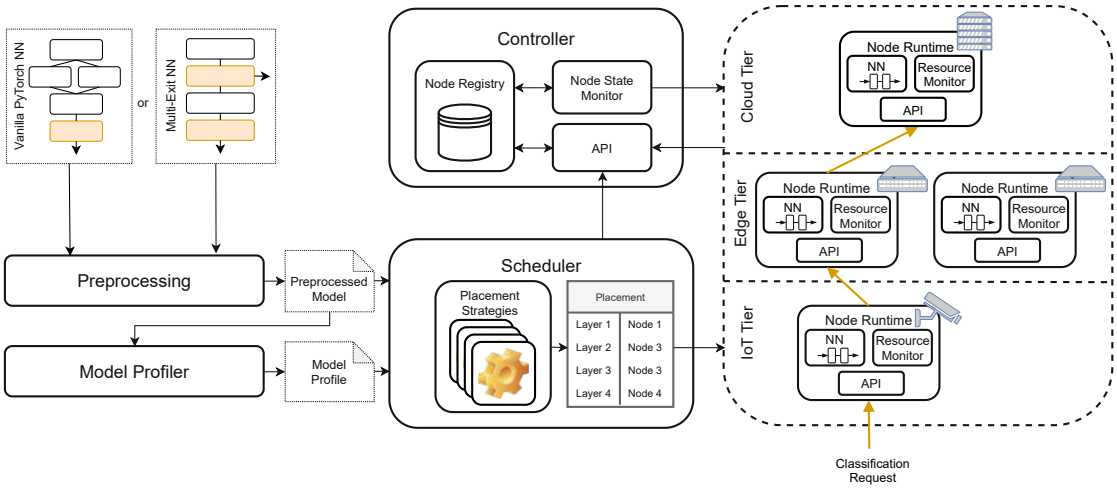


Figure 4.1: System Design

4.3.1 Preprocessing

Before a model can be handed to the core system components, it might be necessary to preprocess the given DNN. This includes, for example, the translation into a representation that is amenable for analysis tasks. Furthermore, at this state, the structure of the model would be determined to identify candidate split points, to enable the partitioning of a model over multiple compute nodes.

4.3.2 Profiler

After a model has been prepared for deployment and candidate split points have been detected, it is necessary to determine a model's resource requirements. This information is needed by the scheduler (see Section 4.3.4) to be able to determine an appropriate assignment of layers to compute nodes. In particular, a model's profile could include the following information on a per-layer basis:

- **Memory requirements:** Models of modern DNN architectures may comprise several hundreds of MBs when loaded in-memory. On powerful servers this would hardly pose any problems. On more resource constrained devices, however, it might be necessary to restrict the amount of memory that is allocated to a node's runtime. In order for the scheduler to be able to respect these limitations, it is necessary to have a conservative estimate of each layer's memory overhead. The amount of memory that is required for a layer is characterized, for example, by the size of its parameters (i.e. weight matrices that were learned during training).
- **Compute requirements:** In order to estimate the time it takes to obtain a layer's intermediate result, it is necessary to quantify the compute workload that is associated with each layer. Specifically, since computations in a neural network are

based on manipulation of matrices that contain floating point values, the number of *floating point operations (FLOPs)*, that are executed when a single input sample is processed by a layer, may serve as a baseline for estimations.

- **Size of intermediate results:** This is necessary in order to estimate the communication overhead that would occur if adjacent layers would be deployed to different compute nodes, based on partitioning decisions.

4.3.3 Controller

The *controller* is the central management and monitoring component that maintains a global view on the compute hierarchy. Any host, that wants to provide its compute resources to the system, initially needs to register its node runtime with the controller. It maintains a global view on the compute hierarchy, that can be accessed through according API services. In particular, the controller comprises three key components which are described in the following.

Node Registry

The node registry represents a database of all available compute nodes, associated metadata, and monitoring information. In particular, it stores the resource state of each compute node such as CPU levels, memory utilization, and network quality e.g. in the form of bandwidth and latency matrices.

API

The controller provides southbound as well as northbound APIs. A southbound API exposes entry points for the compute nodes to announce their availability and register their node-runtime within the compute hierarchy. A northbound API provides access to information about the registered compute nodes and their resource states.

Node State Monitor

After compute nodes registered themselves with the controller, it is necessary to track their availability and monitor their resource levels. A dedicated monitoring agent contacts the registered compute nodes in regular intervals to obtain their current runtime state.

4.3.4 Scheduler

At the heart of the orchestration mechanism sits the scheduler component. Upon startup and during run-time (e.g., in regular intervals) it contacts the controller to obtain the current state of the compute hierarchy. Together with the information about resource requirements of the DNN layers, which it obtained from the model profiler, it then computes a placement and deploys each DNN layer to its assigned compute host.

Furthermore, the scheduler should reevaluate its placement decisions, in response to changing environmental conditions.

From a software architecture perspective, the scheduler would provide a plugin framework for implementing, e.g., different deployment, resource allocation, and adaptation strategies. This way, different optimization goals can be pursued.

Action Space

The action space of the scheduler, for tuning the performance of the system, covers multiple dimensions. Possible actions include:

- **Placement of Layers:** Based on the resources requirements of a model as well as application requirements, the scheduler decides about the placement of a given model on available compute nodes. These decisions could be made on a per layer basis, which could hence involve partitioning a model over multiple hosts.
- **Confidence Thresholds Tuning**: By reconfiguring confidence thresholds of early exit branches, the scheduler is able to adjust the exit rates as well as accuracy of side exit classifiers, depending on application-specific constraints.
- **Cloud Resource Allocation**: In response to varying system load levels, it might be profitable to scale resources (e.g., number of vCPUs) of available cloud nodes.

4.3.5 Node Runtime

Compute nodes are organized in several tiers that form a compute hierarchy. Each compute node implements a runtime environment that allows to receive DNN layers, perform the respective computations, communicate with other nodes in the compute hierarchy, trigger inference, and perform monitoring. Nodes initially register with the controller, and the latter is responsible for contacting them periodically to determine their availability, as described earlier. Communication tasks between compute nodes are carried out over appropriate APIs. This includes exchanging serialized intermediate results of DNN inference between layers.

In particular, the node's runtime system comprises multiple components which are discussed in the following.

Resource Monitor

Each node's runtime system is responsible for monitoring various environmental conditions which are relevant for orchestration. For example, periodically, each node measures the latency to other known hosts, as well as the available bandwidth.

API

A compute node's runtime environment is equipped with an API server that exposes three kinds of APIs for (i) monitoring availability of resources, (ii) deploying parts of a DNN to the node, and (iii) for triggering DNN inference by sending an input sample to the layers that are hosted at the node.

CHAPTER 5

System Implementation

Based on the design that was outlined in Chapter 4 a prototypical system is implemented. It is a proof of concept of the proposed design that serves as a basis for the feasibility studies in Chapter 7. Figure 5.1 provides an overview of the system implementation — in contrast to Figure 4.1 it concretizes implementation aspects, which will be discussed in detail in the following sections.

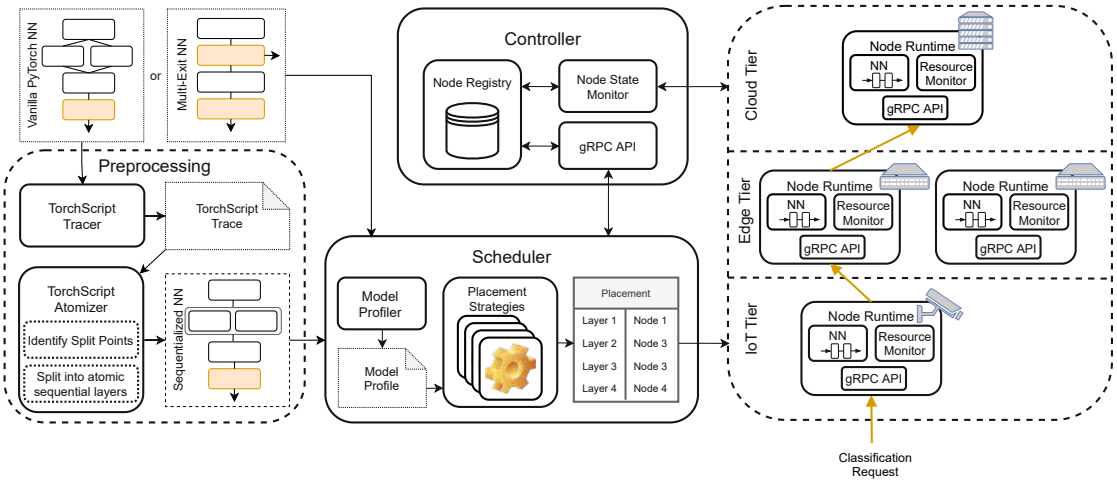


Figure 5.1: System Implementation

5.1 DNN Modeling

As part of the system, we provide a programming model built on top of PyTorch, which allows developers to compose and train neural network models, potentially with early exits, and explicitly define their distributable layers. A DNN produced this way can be

passed on by the service provider to the orchestration system which takes care of the distribution of its layers to compute hosts.

The system provides interfaces as part of a framework and library that allow a user to compose DNN models for distribution over a hierarchy of compute nodes, possibly including multiple side-exits. The PyTorch API defines a `torch.nn.Module` [1] class that acts as base-type for all DNN models. Similarly, the system provides a class-hierarchy that acts as the main interface to a DNN model for both the training phase and for serving the trained model to production.

Such an *adaptive distributable deep neural network (ADDNN)* is represented by instances of the class `Model`, shown in listing 5.1. This is the main interface of a neural network for the orchestration and runtime system.

```
1  class Model(torch.nn.Module):
2      """Represents the model of an Adaptive Distributed Deep Neural Network.
       """
3      def __init__(self, layers: Iterable[Layer]):
4          ...
5
6      @property
7      def layers(self) -> List[Layer]:
8          """Returns the layers of this ADDNN model."""
9          ...
10
11     def forward(self, x: torch.Tensor) -> List[torch.Tensor]:
12         """Returns the predictions of all exit classifiers in this model."""
13         ...
```

Listing 5.1: Model class

Furthermore, `Model` instances are made up of a linear sequence of distributable layers. Each layer, in turn, is represented by an instance of the `Layer` class, which is depicted in Listing 5.2. Each layer holds a portion of the neural network's main branch. Additionally, the main branch may be accompanied by an exit branch that can be used to obtain a classification result at this layer. Finally, it provides an implementation of the abstract `forward()` method that passes the given tensor to the layer's main branch and optionally classifies the corresponding intermediate result via the layer's exit branch, if available. It returns a tuple containing both the intermediate result of the main branch, as well as the classification result of the layer's exit branch.

```
1  class Layer(torch.nn.Module):
2      """Represents a layer of an Adaptive Distributed Deep Neural Network."""
3      def __init__(self, main_branch: Optional[torch.nn.Module], exit_branch:
       Optional[Exit] = None):
4          ...
5
6      @property
7      def main_branch(self) -> Optional[torch.nn.Module]:
```

[1] https://pytorch.org/docs/stable/generated/torch.nn.Module.html

```
        """Returns the main branch of this ADDNN layer."""
        ...

    @property
    def exit_branch(self) -> Optional[Exit]:
        """Returns the exit branch of this ADDNN layer."""
        ...

    @property
    def input_size(self) -> Optional[torch.Size]:
        """Returns the input size of the layer's main branch."""
        ...

    @property
    def number_of_exited_samples(self) -> int:
        """Returns the number of samples that took the exit at this layer."""
        ...

    def forward(self, x: torch.Tensor) -> Tuple[Optional[torch.Tensor],
    Optional[torch.Tensor]]:
        """Returns the output of the main branch and the prediction of the
    exit branch."""
        ...
```

Listing 5.2: Layer class

DNN exits are represented by instances of the class `Exit` which is depicted in listing 5.3. Each `Exit` holds a classifier in the form of a `torch.nn.Module`. The `Exit`'s confidence threshold is used by the inference mechanism at runtime. It defines the maximal, desirable entropy of classification results at an `Exit`. During training, the confidence thresholds of the according `Exit`s can be ignored. Finally, it provides an implementation of the abstract `forward()` method that simply passes the supplied tensor to the exit's classifier and returns the corresponding classification result.

```
class Exit(torch.nn.Module):
    """Represents an exit branch of a neural network."""
    def __init__(self, classifier: torch.nn.Module, confidence_threshold:
    float):
        ...

    @property
    def classifier(self) -> torch.nn.Module:
        """Returns the classifier of this ADDNN exit."""
        ...

    @property
    def confidence_threshold(self) -> float:
        """Returns the confidence threshold of this ADDNN exit."""
        ...

    @property
    def number_of_exited_samples(self) -> int:
        """Returns the number of samples that took this exit."""
```

```
19          ...
20
21      @number_of_exited_samples.setter
22      def number_of_exited_samples(self, number_of_exited_samples) -> None:
23          """Set the number of samples that took this exit."""
24          ...
25
26      def forward(self, x: torch.Tensor) -> torch.Tensor:
27          ...
```

Listing 5.3: Exit class

5.2 Split Point Detection

DNN models that have been implemented via the proposed programming model, outlined in the previous section, have an inherent sequential structure. Therefore, in such models, each layer represents a valid candidate split point and will not be subject to further preprocessing, as can be seen in Figure 5.1. However, the system also supports the orchestration of arbitrary models that are instances of PyTorch's `torch.nn.Module` class. Such models are not restricted to a certain structure and therefore the search for candidate split points in such models is more involved.

For that purpose, we provide an automated model slicing mechanism that operates on a pre-trained, vanilla PyTorch model (in particular, on TorchScript[2], an intermediate representation of serialized PyTorch modules), scans its computational graph, and automatically identifies split points. The processed model can be submitted for serving and is treated by the runtime system in an identical manner as the ones built using our developer facilities described in Section 5.1. Therefore, our system can orchestrate existing, pre-trained models, without modifications.

5.2.1 Transforming models into TorchScript

TorchScript refers to an intermediate representation of PyTorch code, that is used for optimization and efficient execution of DNN inference in production environments. For example, a `torch.nn.Module`, that has been transformed into TorchScript, can be embedded into C++ programs without any dependencies on a Python runtime or any of PyTorch's Python-based components. There are two ways to obtain a TorchScript representation of a `torch.nn.Module`:

- **Scripting:** This refers to the compilation of any Python function or `torch.nn.Module` to its equivalent TorchScript representation. For this purpose, PyTorch offers a `torch.jit.script` function, which is the entry point to the TorchScript compiler. It is able to translate any `torch.nn.Module`, Python callable, or generic Python class to TorchScript.

[2] `https://pytorch.org/docs/stable/jit.html`

- **Tracing:** The execution trace of a model can be created via PyTorch's `torch.jit.trace` function. In contrast to scripting, tracing does not employ compilation. Instead, the TorchScript trace of a module is obtained by executing the module's `forward` method and recording the specific operations that are executed by the TorchScript interperter.

A TorchScript program that was generated via the scripting API might contain arbitrarily complex control flow. A TorchScript trace, on the other hand, only contains sequential control flow since it only represents the computations that have been executed at the time the trace was captured. For the purpose of this thesis, we focused on TorchScript traces, due to their limited control flow. For future extensions of this work, it might be preferrable to extend the presented mechanisms to generic TorchScript graphs, that have been compiled via PyTorch's scripting facilities.

To demonstrate the usage of the tracing API, consider the exemplary model in Listing 5.4. It shows the source code of a simple model, which extends the `torch.nn.Module` class, that serves as base type for all models in PyTorch, as described earlier. The `MyModel` class has a single `layer` member of type `torch.nn.Linear`, which represents a fully-connected layer. Inference is implemented by overriding the abstract `forward` method of `torch.nn.Module`. In this example, inference involves the execution of the fully-connected layer, followed by a `relu` operation.

```
1 class MyModel(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.layer = torch.nn.Linear(in_features=1, out_features=5)
5
6     def forward(self, x):
7         x = self.layer(x)
8         x = torch.nn.functional.relu(x)
9         return x
```

Listing 5.4: Transforming a torch.nn.Module into TorchScript

A TorchScript trace for this model can be obtained as shown in Listing 5.5. The `torch.jit.trace` function returns a `ScriptModule`, which is the TorchScript-equivalent to PyTorch's generic `torch.nn.Module` class. An instance of `MyModel` and a concrete input sample are passed to `torch.jit.trace`, in order to extract an execution trace.

```
1 model = MyModel()
2 x = torch.rand(1, 20, 20, 1)
3 model.eval()
4 trace = torch.jit.trace(model, x)
```

Listing 5.5: Transforming a torch.nn.Module into TorchScript

Internally, a TorchScript program is represented in the form of an intermediate representation (IR) graph that encodes the control flow and data flow of the translated PyTorch code. A human-readable representation of the trace from above, is shown in Listing 5.6.

```
graph(%self.1 : __torch__.MyModel,
      %input.1 : Float(1, 20, 20, 1, strides=[400, 20, 1, 1], requires_grad
    =0, device=cpu)):
  %2 : __torch__.torch.nn.modules.linear.Linear = prim::GetAttr[name="layer"
    ](%self.1)
  %5 : int = prim::Constant[value=1]()
  %6 : Tensor = prim::GetAttr[name="bias"](%2)
  %7 : Tensor = prim::GetAttr[name="weight"](%2)
  %8 : Float(1, 5, strides=[1, 1], requires_grad=1, device=cpu) = aten::t(%7)
  %9 : Float(1, 20, 20, 5, strides=[2000, 100, 5, 1], requires_grad=1, device
    =cpu) = aten::matmul(%input.1, %8)
  %input : Float(1, 20, 20, 5, strides=[2000, 100, 5, 1], requires_grad=1,
    device=cpu) = aten::add_(%9, %6, %5)
  %4 : Float(1, 20, 20, 5, strides=[2000, 100, 5, 1], requires_grad=1, device
    =cpu) = aten::relu(%input)
  return (%4)
```

Listing 5.6: Transforming a torch.nn.Module into TorchScript

The TorchScript language is strongly typed and comes with support for primitive as well as complex types. Primitive types, as well as operations that yield primitive values, are characterized by their `prim::` prefix. For example, primitive types are `bool`, `int` but also compound types such as arrays (e.g. `int[]`), as well as a `Tensor` type. IR Nodes, that represent tensor operations, are characterized by their `aten::` prefix.

5.2.2 Finding candidate split points

The identification of candidate split points is based on traversing the data flow graph of a model's TorchScript trace. The Python API of PyTorch provides a limited API to inspect the TorchScript representation of a program. However, PyTorch provides more extensive APIs for C++, which are available in the form of a static library `libtorch.a`. It provides means for inspecting and manipulating TorchScript programs.

`torch::jit::script::Module` is the C++ representation of `ScriptModule`. The central entry point to a module is its `forward` method. Methods of `torch::jit::script::Module` instances are represented by `torch::jit::Method`. The body of a method is the actual TorchScript, which is represented by `torch::jit::Node`s that together form a `torch::jit::Graph`. Each node represents a certain TorchScript operation that may produce values that in turn are represented by `torch::jit::Value`.

Based on the described API components, our system implements a mechanism to identify the candidate split points in a given `torch::jit::script::Module`. This procedure is outlined in Algorithm 5.1 and is available as a standalone C++ application named `torchscript-atomize`.

The mechanism operates on the topographically sorted list of nodes that are part of a TorchScript trace and returns a list of all possible split points. In particular, the trace represents the operations of the `forward` method of the original `torch.nn.Module`. This topographical order corresponds to the execution order of the nodes when the trace was

recorded and can be obtained via the `torch::jit::Graph::nodes()` method. The split points are represented by the positions of the nodes, where the `torch::jit::Graph` of the trace can be split. The identification of those split points is based on the traversal of the data flow edges in the graph. Intuitively, a TorchScript graph is split at all positions, where only a single data flow edge is active. Therefore, the algorithm keeps track of the unconsumed data flow edges while iterating over the graph's nodes in a single forward pass.

The algorithm starts from an empty set of split points on line 1. Initially, on line 2, the only unconsumed data flow edges originate from the tensor argument that is passed to the module's `forward` method. Starting from line 3, all nodes are traversed in order of the control flow to inspect their data flow edges. Each node might consume a value that has been computed by one of its predecessors. The consumption of a value corresponds to an incoming data flow edge. Therefore, the loop at line 4 inspects the node's incoming data flow edges and updates the `unconsumedDataFlowEdges` accordingly. Likewise, based on the consumed inputs, each node might compute a value which in turn could be consumed by its successors. After the number of corresponding outgoing data flow edges is determined at line 10, the `unconsumedDataFlowEdges` are again updated accordingly. Finally, if only a single data flow edge is active at the current point in the graph, the position of the current node is recorded as candidate split point on line 14.

After all split points are identified, the TorchScript graph is partitioned into according sub-graphs which are then exported as separate `torch::jit::script::Module` instances. As indicated in Figure 5.1, the outlined preprocessing of TorchScript-based models is not an integrated part of the system. The recording of a TorchScipt trace as well as the `torchscript-atomize` utility have to be initiated manually by the system operator.

5.3 Controller

This section details the implementation of the system's controller in accordance with the design proposed in Section 4.3.3. From an architectural point of view, the controller's components are operated in the form of two Python processes. Namely, a gRPC server and a node state monitoring server, each accounting for one process. The details of these components are introduced in the following sections.

5.3.1 Node Registry

For reasons of simplicity, we abstained from hosting a fully-fledged database management system to maintain the node registry data. Instead, metadata of the registered compute nodes and their corresponding monitoring information are just held in the form of in-memory data-structures by the controller's gRPC API server, which is introduced next in Section 5.3.2.

Algorithm 5.1: Finding candidate split points in a TorchScript trace

Input: a TorchScript trace
Result: all candidate split points in the given TorchScript trace

splitPoints ← ∅;
unconsumedDataFlowEdges ← number of outgoing data flow edges of initial tensor argument;
foreach *node ∈ topographically sorted nodes in the TorchScript trace* **do**
  foreach *incoming data flow edge of the current node* **do**
    if *incoming data flow is the result of a tensor operation* **then**
      unconsumedDataFlowEdges ← unconsumedDataFlowEdges - 1;
    end
  end
  if *node is a tensor operation* **then**
    numberOfOutputs ← number of outgoing data flow edges of current node;
    unconsumedDataFlowEdges ← unconsumedDataFlowEdges + numberOfOutputs;
    if *unconsumedDataFlowEdges = 1* **then**
      nextSplitPoint ← position of current node;
      splitPoints ← splitPoints ∪ nextSplitPoint;
    end
  end
end
return *splitPoints*

5.3.2 gRPC API

The controller's gRPC API comprises both southbound API endpoints for the communication with compute nodes, as well as northbound endpoints for higher-level components to monitor the current state of the compute hierarchy. The gRPC service definition of the controller's gRPC API is given in listing 5.7.

```
service Controller {
    rpc RegisterNode(RegisterNodeRequest) returns (RegisterNodeResponse) {}
    rpc DeregisterNode(DeregisterNodeRequest) returns (google.protobuf.Empty) {}
    rpc ListNodes(google.protobuf.Empty) returns (ListNodesResponse) {}
    rpc UpdateNodeState(UpdateNodeStateRequest) returns (google.protobuf.Empty) {}
}
```

Listing 5.7: gRPC service definition of the controller API

For a host to provide its resources to the compute hierarchy, it initially has to register itself with the controller, using the `RegisterNode` method of the `Controller` service. The

request body for that service method is defined as `RegisterNodeRequest` which is outlined in Listing 5.8.

As part of that request, the node includes information about its runtime state which is represented by the `Node` message. This includes the following information:

- The host address as well as the port where the gRPC services of the node's runtime are exposed.
- Since the compute nodes are organized in tiers, the node also specifies the particular tier (as an integer index) it belongs to.
- The node's resource state upon registration is included in the `state` field in the form of a `NodeState` message.
- Finally, the port where the node's bandwidth monitoring server is listening is included (the node runtime's resource monitoring mechanism is discussed in more detail in Section 5.4).

Upon successful registration, the controller responds with a `RegisterNodeResponse` message. The body of this message only includes a UUID that uniquely identifies this node within the compute hierarchy.

A node's compute resources can be withdrawn from the system via the `DeregisterNode` endpoint. As outlined in Listing 5.8, a `DeregisterNodeRequest` comprises the UUID of the node that should be deregistered from the compute hierarchy. The controller concludes the deregistration by responding with an empty (`google.protobuf.Empty`) message.

```
message Node {
    // The host name or IP of the node.
    string host = 1;

    // The port at which the node's APIs can be reached.
    uint32 port = 2;

    // The tier that the node is part of.
    uint32 tier = 3;

    // Whether the node is the input source for the neural network (i.e., the
    // node represents an end device or sensor).
    bool is_input = 4;

    // The current resource state of the node.
    addnn.grpc.node_state.NodeState state = 5;

    // The port at which the node's iperf server is exposed
    uint32 iperf_port = 6;
```

```
}

message RegisterNodeRequest {
    // The node that should be registered.
    Node node = 1;
}

message RegisterNodeResponse {
    // The UUID that the controller assigned to the new node.
    string uuid = 1;
}

message DeregisterNodeRequest {
    // The UUID of the node that should be deregistered.
    string uuid = 1;
}
```

Listing 5.8: Protobuf messages for node registration

The current runtime state of a compute node is represented by instances of the `NodeState` message, as used as part of the `RegisterNodeRequest`. As outlined in Listing 5.14, a node's state is characterized by its resource state, as well as the state of the neural network model that is hosted by the node's runtime. While the runtime state and resource levels of registered compute nodes are monitored decentrally at each node's host, they are also tracked and collected centrally by the controller's node state monitor, which is introduced in Section 5.3.3.

As soon as the node state monitor obtains an updated resource state of a certain compute node, it uses the `UpdateNodeState` endpoint to notify the controller about the new resource levels. Concretely, this endpoint accepts a `UpdateNodeStateRequest` which is outlined in Listing 5.9. The payload contains the UUID of the respective node, alongside the updated resource levels in the form of a `NodeState` message.

```
message UpdateNodeStateRequest {
    // The UUID of the node of interest.
    string uuid = 1;

    // The updated state of the compute node.
    addnn.grpc.node_state.NodeState node_state = 2;
}
```

Listing 5.9: Protobuf messages for node monitoring

Finally, the controller's view on the compute hierarchy can be requested via the `ListNodes` endpoint. As detailed in Listing 5.10, it responds with the complete list of compute nodes that are currently registered with the controller.

```
// Represents a compute node that is registered at the controller.
message RegisteredNode {
```

```
3     // The UUID that the controller assigned to the node.
4     string uuid = 1;
5
6     // The node that is registered at the controller.
7     Node node = 2;
8 }
9
10 message ListNodesResponse {
11     // The nodes that are currently registered at the controller.
12     repeated RegisteredNode nodes = 1;
13 }
```

Listing 5.10: Protobuf messages for node monitoring

5.3.3 Node State Monitor

As soon as a compute node announced its availability to the system, based on the `RegisterNode` endpoint, the controller starts tracking the node's runtime state and resource levels. For that purpose, the controller runs a *node state monitor* in the form of a separate Python process.

In a configurable interval, this monitoring process retrieves the list of currently active nodes via the controller's `ListNodes` endpoint. For each active node, it then uses the respective host and port information, which is part of the `RegisteredNode`'s `node` field, to connect to the respective node's `Node` gRPC service (the service definition is outlined in Listing 5.11 and is introduced in more detail in Section 5.4).

Upon successful connection, the node's current state is obtained via its `ReadNodeState` endpoint. The updated state is then persisted in the controller's in-memory node registry based on the `UpdateNodeState` endpoint.

In case of a failed connection attempt, the node state monitor automatically deregisters the node using the controller's `DeregisterNode` endpoint, to make sure that it will not be considered for future placement decisions of the scheduler.

5.4 Node Runtime

For a host to provide its compute resources to the system it is required to operate a runtime environment, as introduced in Section 4.3.5. From an architectural point of view, the node's runtime system consists of two independent Python processes. One process acts as gRPC server and hosts the node's API services. A second process is responsible for monitoring the node's resource levels as well as connectivity to other nodes in the system.

5.4.1 gRPC API

A compute node's runtime environment is equipped with a gRPC server that exposes three kinds of APIs. They provide endpoints for (i) monitoring availability of resources, (ii) deploying parts of a DNN to the node, and (iii) for triggering DNN inference by sending an input sample to the layers that are hosted at the node.

DNN deployment and runtime monitoring

From an implementation point of view, these APIs are organized as two separate gRPC services. The APIs for configuration and monitoring of the node runtime (i and ii) are available in the form of a consolidated `Node` service, that is outlined in Listing 5.11.

```
service Node {
    rpc DeployModel(stream LocalLayer) returns (google.protobuf.Empty) {}
    rpc DeleteModel(google.protobuf.Empty) returns (google.protobuf.Empty) {}
    rpc ActivateLayers(ActivateLayersRequest) returns (google.protobuf.Empty) {}
    rpc DeactivateLayers(google.protobuf.Empty) returns (google.protobuf.Empty) {}
    rpc ReadNodeState(ReadNodeStateRequest) returns (ReadNodeStateResponse) {}
    rpc UpdateResourceState(UpdateResourceStateRequest) returns (google.protobuf.Empty) {}
    rpc ReadNeighbourNodes(google.protobuf.Empty) returns (ReadNeighbourNodesResponse) {}
}
```

Listing 5.11: gRPC service definition of the node API

In case a node does not yet host any DNN layers, it is possible to deploy a DNN model to the node's runtime by transmitting a `stream` of `LocalLayer` messages to the node's `DeployModel` endpoint. Using a gRPC `stream` provides several advantages over conventional *unary* RPCs, where a request consists only of a single message. DNN models may comprise several hundreds of MBs, hence sending all the layers of a DNN model at once, would involve sending one, possibly big, protobuf message. For resource constrained devices, the reception of such messages could incur memory overheads that might exceed the available memory. Instead, sending each layer as a separate message as part of a stream, allows the node's runtime to handle each layer separately. In particular, upon receipt of a `LocalLayer`, the layer will be saved directly to disk and will not be held in memory. Hence, during deployment, at most one layer will reside in memory at a time, which helps ensure that memory limits of resource constrained nodes will not be exceeded.

As outlined in Listing 5.12, a `LocalLayer` represents a serialized layer of a DNN model which may consist of both a `main_branch` as well as an `exit_branch`, that forms an optional side-exit classifier. This exit branch is represented by the `Exit` message,

which encapsulates both a serialized `torch.nn.Module` instance, as well as a configurable `confidence_threshold`, that determines the exit behavior for the respective classifier.

The node runtime is able to operate DNN models that conform to PyTorch's Python-based APIs as well as native TorchScript. Since the unmarshalling of a `torch.jit.ScriptModule` requires a different deserialization mechanism than unmarshalling of a pickled Python-based `torch.nn.Module` (namely, `torch.jit.load()` vs `torch.load()`), the flag `is_torchscript` identifies the type of the layer to help determine which deserialization mechanism to use upon receipt of the layer.

```
// Represents the exit branch of a ADDNN layer.
message Exit {
    // A pickled torch.nn.Module that represents the classifier of this exit.
    bytes classifier = 1;

    // The confidence threshold of this side exit (a value in the closed interval [0.0, 1.0]).
    float confidence_threshold = 2;
}

// Represents an locally available ADDNN layer.
message LocalLayer {
    // A pickled torch.nn.Module that represents this layer's portion of the DNN's main branch.
    bytes main_branch = 1;

    // An optional exit that is placed at this layer.
    Exit exit_branch = 2;

    // Whether the layer's branches are instances of `torch.jit.ScriptModule` instead of `torch.nn.Module`.
    bool is_torchscript = 3;
}
```

Listing 5.12: Protobuf message for model deployment

Note that the `DeployModel` endpoint only ensures that the complete model is made available to a node's runtime system, by saving each of its layers to an on-disk cache. Only after all the layers of a model have been deployed to a node's runtime environment, the compute node can be considered by the scheduler's placement decision processes. For this purpose, the `Node` service offers the `ActivateLayers` endpoint. By transmitting an `ActivateLayersRequest`, which is defined in Listing 5.13, a node is instructed to load a certain range of DNN layers from its on-disk cache, so that they can readily be executed upon receipt of an inference request. The respective range of layers is represented by the `active_layers` field in the form of a `LayerRange` message. Depending on the partitioning decisions of the scheduler, this range might not include the final layer of the DNN model. In this case, the `ActivateLayersRequest` also contains continuation details in the form of a `RemoteLayer`, which specifies where to reach the compute node that hosts the next

layers. The deployment and activation process, which is performed by the scheduler, is described in more detail in Section 5.5.

```
message ActivateLayersRequest {
    // The range of layers that should be active on this node.
    LayerRange active_layers = 1;

    // Determines where to reach the next layer, if existing (i.e. `active_layers.end_index + 1`).
    RemoteLayer remote_layer = 2;
}

// References a range of layers in a model.
message LayerRange {
    // The 0-based index of the first layer referenced by the range.
    uint32 start_index = 1;

    // The 0-based index of the last layer referenced by the range.
    uint32 end_index = 2;
}

// Represents an ADDNN layer at a remote host.
message RemoteLayer {
    // The host name or IP of the node that hosts the layer.
    string host = 1;

    // The port at which the layer can be reached.
    int32 port = 2;
}
```

Listing 5.13: Protobuf messages for layer activation

The `Node` service also provides a `ReadNodeState` endpoint, in order to obtain the current state of its runtime environment, in the form of a `NodeState` message, as defined in Listing 5.14.

```
message NodeState {
    // The resource state of the node.
    ResourceState resource_state = 1;

    // The state of the neural network that is hosted by the node.
    NeuralNetworkState neural_network_state = 2;
}
```

Listing 5.14: Protobuf messages for node monitoring

The node's state is characterized by its `ResourceState` (which will be introduced in more detail in Section 5.4.2) as well as the state of the DNN layers that it hosts:

```
message NeuralNetworkState {
    repeated LayerState layer_states = 1;
```

```
}

message LayerState {
    // The 0-based index of the layer.
    uint32 layer_index = 1;

    // Whether the layer is currently active at that node.
    bool active = 2;

    // The number of samples that exited at this layer.
    uint32 number_of_exited_samples = 3;
}
```

Listing 5.15: Protobuf messages for node monitoring

The `NeuralNetworkState` specifies the runtime state for all DNN layers that have been deployed to a node. Namely, the `active` flag determines if a layer has been activated by a prior `ActivateLayers` request. Furthermore, if a side-exit classifier is attached to a given layer, its state also comprises the number of samples that exited at the layer's classifier. Concretely, the `number_of_exited_samples` field describes the exit behavior of all the samples that have been seen by a node's inference mechanism, which is introduced in the following.

DNN inference

Finally, the entrypoint for triggering neural network inference for an input sample is exposed as a separate gRPC service:

```
service NeuralNetwork {
    rpc Infer(InferRequest) returns (InferResponse) {}
}
```

Listing 5.16: gRPC service definition of the nodes' neural network API

The `NeuralNetwork` service exposes a single endpoint named `Infer`, that can be used to obtain a classification result for a given tensor. Its request payload is defined by the `InferRequest` message in Listing 5.17. It has a single `bytes` field that represents a `torch.Tensor` instance that has been serialized to a byte array using PyTorch's `torch.save()` utility. After the node successfully finishes the inference procedure, it responds with a `InferResponse` message, which contains a single `classification` field that represents a class label in the form of an integer index.

```
message InferRequest {
    // A pickled torch.Tensor that should be classified.
    bytes tensor = 1;
}

message InferResponse {
```

```
7     // The classification result that was inferred for the input tensor.
8     int32 classification = 1;
9 }
```

Listing 5.17: Protobuf messages for DNN inference

The inference mechanism, which is performed upon invocation of the `Infer` endpoint, is shown in Algorithm 5.2. It is an extension of the mechanism that is outlined for multi-exit networks in Algorithm 2.1 in Section 2.2.

Algorithm 5.2: Performing classification on a compute node

Input: Tensor $\mathbf{x}$
Result: Class label
Function Infer($\mathbf{x}$) **is**
  firstLayerIndex ← the index of the first active layer;
  lastLayerIndex ← the index of the last active layer;
  for $i \leftarrow$ *firstLayerIndex* **to** *lastLayerIndex* **do**
    $\mathbf{x} \leftarrow l_i(\mathbf{x})$;
    if *layer i has a local exit* **then**
      $\mathbf{y} \leftarrow c_i(\mathbf{x})$;
      $\mathbf{s} \leftarrow softmax(\mathbf{z})$;
      $e \leftarrow \eta(\mathbf{s})$;
      if $e \leq \mathbf{T}_i$ **then**
        increment exit counter for layer i;
        return $argmax(\mathbf{y})$
      end
    end
  end
  return *result of remote invocation of* Infer($\mathbf{x}$) *at next node*
end

The input tensor $\mathbf{x}$ represents a deserialized tensor instance, that was submitted as part of an `InferRequest` message. In contrast to the original algorithm in 2.1, the adapted inference mechanism needs to consider that the DNN model might be partitioned over multiple compute nodes. The inference process is bound to execute only those DNN layers, that have been activated by a prior `ActivateLayers` request. Hence, inference starts at the first active layer and proceeds until it reaches a layer with an attached side-exit classifier. If the node is confident about the classification result at the respective exit classifier, inference stops at this point of the neural network. The node's runtime tracks the exit behavior of each classifier. For each sample, exiting at a certain layer involves the incrementation of a dedicated counter, that is used to track the number of samples that were classified successfully by a layer's exit classifier, as shown on line

11. This counter corresponds to the aforementioned `number_of_exited_samples` field of a layer's `LayerState`, when requesting a node's runtime state, as described earlier.

Note that the inference mechanism assumes that each classifier has an associated exit threshold. In particular, the threshold of the DNN's final layer is always set to 1.0, which ensures that inference always exits at this layer, in case none of the earlier exits was taken. If inference does not exit on line 12, this means that the DNN's final layer is not active in the node's runtime. Hence, inference has to proceed at another node, according to the scheduler's partitioning decision. For this purpose, the node that hosts the next layer is determined based on the endpoint information of the `RemoteLayer` that was specified as part of the latest invocation of the node's `ActivateLayers` endpoint. This information is used to connect to the `NeuralNetwork` service of the respective node. Finally, line 16 hands the current intermediate result to the corresponding node and waits until the remotely triggered inference concludes.

5.4.2 Resource Monitor

As described earlier, the monitoring of compute nodes is realized in a decentralized manner. While the controller maintains a global view on the compute hierarchy, it relies on each node's runtime to perform the actual monitoring operations in an independent manner. For that purpose, each node runs a dedicated *resource monitor* that keeps track of the resource levels of its host platform.

In regular intervals, this monitoring process performs the necessary tasks in order to determine its node's current memory, storage, compute capacity, as well as network performance to the other nodes that are managed by the controller.

For each pair of compute nodes in the compute hierarchy, network quality is determined by observing the current throughput and latency levels. For that purpose, the resource monitor regularly obtains the list of currently active nodes from the controller by the use of its `ListNodes` gRPC endpoint.

Throughput levels between nodes are estimated based on `iperf3`. Therefore, each node hosts an `iperf` server, whose lifecycle is managed by the node's resource monitor. Network latency measurements are based on ICMP round-trip-delays.

Finally, the resource monitor announces the obtained state by invoking the `UpdateResourceState` endpoint of its node's runtime. Any subsequent invocations of the `Node` service's `ReadNodeState` endpoint will hence observe the latest resource state as part of the runtime's `NodeState`. Monitoring results are represented as `ResourceState` messages:

```
1 message ResourceState {
2     // The available RAM in bytes.
3     uint64 memory = 1;
4
5     // The available storage in bytes.
6     uint64 storage = 2;
7
```

```
8     // The available compute capacity in FLOPS.
9     uint64 compute = 3;
10
11    // The network throughputs to the node's neighbours.
12    repeated NetworkThroughput network_throughputs = 5;
13
14    // The network latencies to the node's neighbours.
15    repeated NetworkLatency network_latencies = 6;
16 }
```

Listing 5.18: Protobuf message representing a node's resource state

The `NetworkThroughput` and `NetworkLatency` specify the most recent throughput and latency that has been observed between the current node and another `host` that operates another compute node.

```
1  message NetworkThroughput {
2      // Host or IP of the neighbour node.
3      string host = 1;
4
5      // The throughput in bits/second.
6      uint64 throughput = 2;
7  }
8
9  message NetworkLatency {
10     // Host or IP of the neighbour node.
11     string host = 1;
12
13     // The latency in milliseconds.
14     float latency = 2;
15 }
```

Listing 5.19: Protobuf messages representing a node's connectivity

5.5 Scheduler

As described in Section 4.3.4, the scheduler component implements the core functionality of the system's deployment and orchestration operations. The actions that are performed by the scheduler are given in algorithm 5.3.

Upon startup, it performs an offline profiling step in order to estimate the resource requirements of the DNN layers. Namely, the memory, compute, and communication overhead are estimated on a per-layer basis on lines 2 to 4 (these profiling steps will be described in more detail in Section 5.5.2). In regular, configurable intervals, it then computes and applies a placement policy which assigns each individual DNN layer to a specific compute node.

Algorithm 5.3: Scheduler main loop

```
Input:
{l_i | i = 0 ... L − 1}: layers of the DNN
s: a user-chosen placement strategy
for i ← 0 to L − 1 do
    m_{l_i} ← f_m(l_i);
    c_{l_i} ← f_c(l_i);
    b_{l_i} ← f_b(l_i);
end
currentPlacement ← uninitialized;
while true do
    node ← get list of nodes from controller including monitored state;
    foreach node ∈ nodes do
        if node does not yet host the DNN model then
            deploy all layers to the node;
        end
    end
    placementPolicy ← computePlacement(s, m, c, b, nodes);
    if placementPolicy ≠ currentPlacementPolicy then
        foreach node ∈ nodes do
            assignedLayers ← getAssignedLayers(placementPolicy, node);
            if assignedLayers ≠ ∅ then
                nextNode ← uninitialized;
                if l_{L−1} ∉ assignedLayers then
                    nextLayer ← get next layer that is not part of assignedLayers;
                    nextNode ← getAssignedNode(placement, nextLayer);
                end
                activate assignedLayers on node, considering that next layer is
                 assigned to nextNode ;
            end
            else
                if node is input source then
                    nextNode ← getAssignedNode(placement, l_0);
                    activate proxy layer on the node which redirects to nextNode;
                end
                else
                    deactive currently active layers on the node;
                end
            end
        end
    end
    currentPlacementPolicy ← placementPolicy;
    wait for configured time interval;
end
```

For that purpose, on line 8, it contacts the controller in order to obtain the list of currently active compute nodes in the hierarchy, together with the most recent monitoring information about their runtime state. Next, the scheduler needs to determine whether the active compute nodes already host the DNN model. Modern DNN models may comprise several hundreds of MBs in size, hence, deploying a model to a node may take a considerable amount of time, depending on the network conditions. Therefore, if the scheduler encounters that a new node has joined the hierarchy, it deploys all layers upfront, using the node's `DeployModel` endpoint, introduced in Section 5.4.1.

Next, it invokes a user-specified placement strategy in order to compute a placement based on the DNN model's profile information together with the state information about the compute hierarchy. In case the placement policy has not changed, no operation has to be performed. If the placement changed, however, the nodes have to be notified about the new policy accordingly.

On line 17, the scheduler determines the layers that are assigned to a given node, according to the latest placement policy. If at least one layer has been assigned to the node, then the scheduler activates the assigned layers in the node's runtime by issuing an `ActivateLayers` request, which is represented by the actions on line 24. In case the node is not assigned the DNN's final layer, this implies that the model is partitioned over multiple nodes. Therefore, before activating the layers, it is necessary to determine the node that is assigned to the next layer, which is expressed by the condition starting at line 20. In this case, the `ActivateLayers` request, that is issued at line 24, would also need to contain a `RemoteLayer`, representing the layer that is hosted by the next node.

If no layers are assigned to the given node, it is necessary to determine whether the node acts as input source to the compute hierarchy. As described earlier, the system design assumes that a compute hierarchy always contains exactly one node, where input data for the DNN originates. In case the given node acts as input source but the latest placement decision does not assign any layers to this node, it is necessary to activate a proxy layer at the node's runtime, which ensures that the input samples are redirected to the node which hosts the first layer of the DNN model. For the remaining nodes, their runtime is instructed to deactivate previously active layers by issuing a `DeactivateLayers` request, as expressed on line 32.

Finally, the scheduler waits for a configurable amount of time before repeating the above steps. Alternatively, instead of running repeatedly, it is also possible to execute the scheduler as a one-shot mechanism. In that case, the above actions would be performed only once.

5.5.1 Plugin System for Placement Strategies

From a software architecture perspective, we provide a plugin framework, that allows system users to implement and apply their own algorithms for custom placement decisions. This way, different optimization goals can be pursued.

In particular, the scheduler can be equipped with placement strategies that extend the abstract `Strategy` class shown in Listing 5.20. This class provides two abstract methods that have to be overridden by its descendants. By overriding the `name` method, each strategy has to specify a name, that uniquely identifies the strategy within the system. This name is exposed via user-facing APIs, for example, for choosing which strategy to use when starting the scheduler via its CLI.

```
1  NodeIndex = int
2  Placement = List[NodeIndex]
3
4  class Strategy(ABC):
5      """
6      Base class for placement strategies.
7      """
8      @abstractmethod
9      def name(self) -> str:
10         """
11         A name that uniquely identifies this strategy.
12         """
13         raise NotImplementedError
14
15     @abstractmethod
16     def compute_placement(self, nodes: List[Node], layer_profiles: List[
       LayerProfile]) -> Placement:
17         """
18         Compute a placement that assigns each layer to a compute node.
19         """
20         raise NotImplementedError
```

Listing 5.20: Placement strategy base class

The strategy's placement decisions are implemented by overriding the `compute_placement` method. Placement decisions can be based on the current state of the compute nodes (represented by the `nodes` argument) and on the profiling information about the DNN model (represented by the `layer_profiles` argument). The runtime state of each compute node, including its resource levels, is available in the form of `Node` instances (the `Node` type was introduced in Listing 5.8 in Section 5.3.2). The profile of each layer is represented by a `LayerProfile` which summarizes a layer's characteristics, such as the memory, communication, and computation overhead. The result of the `compute_placement` method is the strategy's placement decision, which is an assignment of layers to compute nodes.

Plugin discovery is implemented by means of Python's *namespace packages* [3] [4] [5]. This plugin mechanism allows users, that operate the scheduler, to extend the system with their custom placement strategies, without having to modify the scheduler's source code.

Namespace packages enable a Python application to split its sub-packages and modules, that would normally reside in a single package, across multiple packages, that can then be

[3] https://packaging.python.org/guides/creating-and-discovering-plugins/
[4] https://packaging.python.org/guides/packaging-namespace-packages/
[5] https://www.python.org/dev/peps/pep-0420

distributed separately. Python provides three different ways to create namespace packages: *native* namespace packages, `pkgutil`-style namespace packages, and `pkg_resources`-style namespace packages.

Concretely, the scheduler expects placement strategies to be part of the `addnn.serve.placement.strategies` package. Upon startup, the scheduler traverses all sub-packages of this namespace package, that are available on the current `PYTHONPATH`, and loads all descendants of the `Strategy` class that it finds within this namespace.

Our system readily provides a few placement strategies that aim to minimize end-to-end inference latency, which we review in detail in chapter 6.

5.5.2 Offline Profiling of DNN Models

When the scheduler is started, the user specifies the DNN model that should be served over a given compute hierarchy. Before any placement decisions are carried out, the scheduler performs an offline profiling step, to estimate the model's resource requirements. As shown earlier in Listing 5.20, the result of this profiling step is made available to the placement strategies in the form of a `LayerProfile`, which summarizes the profiling results and important properties of each layer in a DNN model.

We classify this profiling as *offline*, since the profiling information is obtained without performing any *online* measurements directly on the compute nodes. These layer profiles are obtained as follows.

Memory Overhead

The memory overhead of a layer is determined by the size of its input, its parameters (such as learned weight matrices), and the size of its output. The parameters of any `torch.nn.Module` instance can be inspected via its `parameters()` method, which provides access to the parameters of the module as well as all of its sub-modules. However, the profiler's estimation of the memory overhead is only an approximation, since the actual in-memory size of a layer is dependent on the allocation and effective memory layout of the loaded values at run-time.

Computational Overhead

The operations that are carried out by a layer incur a certain compute overhead that determines how long it takes to obtain a classification result for an input sample. If profiling would be performed in an online manner, this compute latency could be determined, for example, by measuring the execution time for each layer on each device, in order to obtain an appropriate execution profile for each host platform. However, since the profiler is run offline, an estimation based approach is applied, that is based on the floating point operations (FLOPS) that are executed by each layer.

In general, the operations that are performed by a layer are well-defined. For example, fully connected layers or convolutional layers perform certain transformations and matrix

operations. If the size of an input sample and other parameters, such as weight matrices, are known upfront, it is possible to calculate the number of operations that will be carried out accordingly. For that purpose, the profiler relies on the `fvcore`[6] library, which provides functionality to estimate the number of FLOPS that are performed by common tensor operations in PyTorch.

Communication Overhead

If a model is partitioned over multiple nodes, this requires the corresponding nodes to exchange intermediate results between layers. This exchange of information incurs an overhead that is dependent on the size of those intermediate results. As mentioned earlier, in PyTorch, each layer is a descendant of the `torch.nn.Module` class, which implements its inference operations by overriding the abstract `forward` method. Therefore, in order to determine the amount of data that is exchanged between layers, the profiler computes the size of the `torch.Tensor` instances that are the result of each layer's `forward` method. The size of the resulting tensor instances are then used to estimate the corresponding communication overhead.

[6] `https://pypi.org/project/fvcore/`

CHAPTER 6

Placement Strategies

In the previous chapters we introduced the framework design, gave an overview of its prototype implementation, and described its APIs, that allow to extend the system with custom mechanisms for the deployment, orchestration, and management of distributed DNNs. In particular, Sections 4.3.4 and 5.5 outline the scheduler which decides about the placement of DNN layers on participating compute nodes. Its flexible plugin architecture allows framework users to implement and apply their own placement strategies, that align with their respective application-specific constraints.

In this chapter we introduce a number of such strategies. Section 6.1 introduces an exact algorithm, that solves the problem of assigning layers to nodes to optimality, with respect to end-to-end inference latency. Due to the intractability of finding an exact solution for the placement problem, heuristic approaches are required for bigger problem instances. For this purpose, Section 6.2 introduces a strategy that is based on a evolutionary algorithm. Finally, sections 6.3 and 6.4 introduce simplistic strategies based on a first-fit-decreasing algorithm and for cloud-only placement.

6.1 Exact Placement

6.1.1 System Model

In order to formalize the exact placement of DNN layers as an integer-linear program, we first define the system model, which in turn, is characterized by three components: (i) the *resource landscape* that comprises the compute hierarchy and its nodes, (ii) the *properties* and *profile information* of the DNN layers, and (iii) the *decision variables*.

The compute nodes are organized in the form of tiers, denoted by $t \in \{1 \dots T\}$, where T is the total number of tiers in the compute hierarchy that is managed by a controller. The complete set of nodes that are registered with the controller is denoted by N, whereas

$N_t \subseteq N$ refers to the nodes that are placed in a certain tier $t \in T$. Furthermore, the system assumes that exactly one node in the lowermost tier serves the input data, that is subject to DNN inference. This node is referred to as n_{input}. Each compute node's host platform is characterized by certain hardware capabilities. Specifically, as described earlier in Section 5.4.2, the resources and environmental conditions of the compute nodes are subject to constant observation by a resource monitor that is run by each node's runtime. The currently available amount of memory for a node $n \in N$ — according to the latest monitoring results — is denoted by m^n. Similarly, c^n denotes an estimate of the node's compute capacity in the form of floating point operations per second (FLOPS), based to the node's current CPU load. Furthermore, a bandwidth and latency matrix describe the network conditions between the compute nodes — for each pair of compute nodes $n, n' \in N$, the bandwidth (or maximal throughput according to measurements) is denoted by $b^{nn'}$ and is indicated in the form of bits/second. The estimated network latency, which is based on the measurement of ICMP round-trip-delays, is given as $e^{nn'}$ and specifies the latency in seconds. The complete notation, that describes the resource landscape, is summarized in Table 6.1.

Notation	Definition
T	Number of tiers in the computing hierarchy
$t \in \{1 \dots T\}$	A specific tier in the computing hierarchy
N	Set of available compute nodes
$N_t \subseteq N$	Compute nodes that are part of tier t
$n \in N$	A specific compute node
$n_{input} \in N_1$	The node where input data originates
m^n	RAM capacity of compute node n (in bytes)
c^n	Compute capacity of compute node n (in FLOPS)
$b^{nn'}$	Bandwidth/throughput between compute nodes n and n' (in bits/second)
$e^{nn'}$	Network latency between compute nodes n and n' (in seconds)

Table 6.1: Resource landscape

Furthermore, placement decisions are based on the properties and profiling information of layers in the given DNN model. A DNN model may consist of L layers, that are arranged in a sequential structure. They can be uniquely identified by their position in the DNN model in the form of an integer index $i \in \{1 \dots L\}$. Each layer is characterized by a certain resource overhead when it is assigned to a compute node. This overhead is estimated by the system's profiler, as described in Section 5.5.2. m_{l_i} denotes the estimated memory overhead in bytes, that is incurred when assigning a certain layer l_i to a node. Additionally, the compute workload that is induced by a layer, when it is executed for a single input sample, is given by the number of performed floating point operations c_{l_i}. Due to a partitioning decision, layer l_i might be assigned to a different node than its successor. This means, that the intermediate results of layer l_i would have

to be transferred between these two nodes, which occurs a communication overhead which is determined by the size b_{l_i} of layer l_i's output. Likewise, if the DNN's first layer is not assigned to the data source n_{input}, then n_{input} would have to transmit each input sample to the node that is assigned to the first layer. Under the assumption that input samples are of equal size, b_{input} denotes the size of a single input sample.

Each layer might also be accompanied by an exit classifier. The corresponding exit rates are given in the form of a probability distribution: p_{l_i} specifies the probability that an input sample might exit at the ith layer, with $p_{l_i} \geq 0, i \in \{1 \dots L\}$ and $\sum_{i \in 1 \dots L} p_{l_i} = 1$.

Notation	Definition
L	Number of layers in the DNN model
$l_i, i \in \{1 \dots L\}$	A specific layer of the DNN model
m_{l_i}	Memory overhead of layer l_i (in bytes)
c_{l_i}	Compute overhead of layer l_i (in FLOPs)
b_{l_i}	Output size of layer l_i (in bytes)
b_{input}	Transfer size (bandwidth consumption) of the input to the first layer (in bytes)
p_{l_i}	Probability that the exit at layer l_i is taken, with $p_{l_i} = 0$ if layer l_i has no exit and $\sum_{i \in 1 \dots L} p_{l_i} = 1$

Table 6.2: Profile information & properties of layers

Finally, the decision variables of the ILP formulation are listed in Table 6.3. The placement decision, which is the final result of the strategy, is encoded by the variable matrix $x_{nl_i} \in \{0,1\}, n \in N, i \in 1 \dots L$. Concretely, $x_{nl_i} = 1$ if and only if layer l_i is assigned to node n.

The second group of decision variables in Table 6.3 is necessary in order to linearize the ILP's optimization target. Intuitively, they represent the partitioning decisions and specify the position of split points in the DNN model.

Notation	Definition
$x_{nl_i} \in \{0,1\}$	Determines whether node n hosts layer l_i
$y_{nn'l_i} \in \{0,1\}$	Denotes whether there is a DNN split point between compute nodes n and n' after layer l_i (i.e., node n hosts layer l_i and node n' hosts layer l_{i+1})

Table 6.3: Decision variables

6.1.2 Constraints

The operation of the node's runtime environment is bound by the resource capacities of the node's host platform. This also means, that assignment decisions, which are made by

placement strategies, have to respect limitations of hardware resources that are provided by compute nodes.

Modern DNN models might comprise several hundreds of MB when loaded into memory. For example, Resnet 152 might account for an in-memory size of more than 300 MB and models such as VGG 19 might even consume up to one GB of memory during inference. While powerful cloud machines might be able to allocate larger amounts of memory to the node runtime, these models might easily exceed the memory capacity of more constrained devices.

Therefore, placement decisions have to respect each node's memory capacity when assigning layers to compute nodes:

$$\sum_{i \in 1 \dots L} x_{nl_i} m_{l_i} \leq m^n, \quad \forall n \in N \tag{6.1}$$

Furthermore, the following constraint is necessary for consistency of the ILP's placement results by ensuring that each layer is assigned to exactly one host:

$$\sum_{n \in N} x_{nl_i} = 1, \quad \forall l \in \{1 \dots L\} \tag{6.2}$$

The next constraint ensures that a partitioning decision does not cause a loop in the inference chain. More formally, non-adjacent layers must not be hosted by the same compute node, i.e., node n must host layer l_i, if both its predecessor (l_{i-1}) and its successor (l_{i+1}) are hosted by n:

$$x_{nl_{i-1}} + x_{nl_{i+1}} - 1 \leq x_{nl_i}, \quad \forall n \in N, \forall i \in \{2 \dots (L-1)\} \tag{6.3}$$

Furthermore, placement decisions must respect the stack-hierarchy of the tiers. This means, for each layer, none of its successors can be assigned to an earlier tier:

$$\sum_{n \in N_t} x_{nl_i} \leq \sum_{t' \in t \dots T} \sum_{n' \in N_{t'}} x_{n'l_{i+1}}, \quad \forall t \in 1 \dots T, \forall i \in \{1 \dots (L-1)\} \tag{6.4}$$

The remaining constraints define the semantics of the decision variables that are needed to linearize the optimization target. There cannot be any split point at the final layer because it does not have a successor:

$$y_{nn'l_L} = 0, \quad \forall n \in N, \forall n' \in N - \{n\} \tag{6.5}$$

Furthermore, split points only exist between two different compute nodes, i.e., if two layers are assigned to the same node, then there occurs no split between these two layers:

$$y_{nnl_i} = 0, \quad \forall n \in N, \forall i \in \{1 \dots L\} \tag{6.6}$$

Finally, there is a split point after layer l between nodes n and n' if and only if node n hosts layer l_i and node n' hosts the successor layer l_{i+1}. So the final group of constraints corresponds to this equivalence: $y_{nn'l_i} \iff x_{nl_i} \wedge x_{n'l_{i+1}}, \forall n \in N, \forall n' \in N - \{n\}, \forall i \in \{1 \dots L-1\}$.

$$\begin{aligned} &y_{nn'l_i} \leq x_{nl_i}, \\ &y_{nn'l_i} \leq x_{n'l_{i+1}}, \\ &y_{nn'l_i} \geq x_{nl_i} + x_{n'l_{i+1}} - 1, \quad \forall n \in N, \forall n' \in N - \{n\}, \forall i \in \{1 \dots L-1\} \end{aligned} \tag{6.7}$$

6.1.3 Optimization Target

The aim of this strategy is to find, amongst all valid solutions, the placement that induces the minimal *end-to-end inference latency*. The end-to-end inference latency accounts for the time it takes to obtain a classification result for an input sample. This time frame spans the submission of an input sample to the runtime on the input source node n_{input}, the traversal of all layers on their respective hosts, as well as the receipt of the final classification result, when it has been communicated back to n_{input}. It is characterized by two components, namely (i) the compute latency, which covers the execution time of the layers, as well as (ii) the communication latency, which covers the transmission of tensors and results between nodes.

The compute workload, that is induced by each layer $l_i, i \in 1 \dots L$, is given as number of floating point operations c_{l_i} that are executed for a single input sample, whereas the compute capacity of each node $n \in N$ is characterized as the number of floating point operations it is able to execute per second. Hence, the time in seconds it takes to execute some layer l_i on a compute node n is:

$$\frac{c_{l_i}}{c^n} \tag{6.8}$$

Furthermore, with C_{l_i} we denote the total computation time of all layers up to the ith layer:

$$C_{l_i} = \sum_{j \in 1 \dots i} \sum_{n \in N} x_{nl_j} \frac{c_{l_j}}{c^n}, \quad i \in 1 \dots L \tag{6.9}$$

Next, we consider the time it takes to transmit each layer's output to the next layer. B_{l_i} denotes the total transmission time between the layers up to the ith layer.

$$B_{l_i} = \sum_{j \in 1 \dots i-1} \sum_{n \in N} \sum_{n' \in N - \{n\}} y_{nn'l_j} \left(\frac{b_{l_j} \times 8}{b^{nn'}} + e^{nn'} \right), \quad i \in 1 \dots L \tag{6.10}$$

Finally, if n_{input} does not host any layers, we also have to consider the time it takes to transmit the application's input data from n_{input} to the node that hosts the first layer. This is given by B_{input}:

$$B_{input} = \sum_{n \in N - \{n_{input}\}} x_{nl_1} \left(\frac{b_{input} \times 8}{b^{n_{input}n}} + e^{n_{input}n'} \right) \tag{6.11}$$

For network architectures that comprise multiple exit classifiers, we also have to take the exit probabilities of each layer into account. This leads to the following optimization target:

$$\min \left(\sum_{i \in 1 \dots L} p_{l_i} \left(C_{l_i} + B_{l_i} \right) + B_{input} \right) \tag{6.12}$$

For traditional network architectures that only contain a single classifier at the final network layer, the exit probabilities of the final exit layer would be 1. In this case the target would correspond to:

$$\min \left(C_{l_L} + B_{l_L} + B_{input} \right) \tag{6.13}$$

6.1.4 Computational Complexity

As we show in this section, the DNN layer placement we have formulated is intractable. In the following, we provide a formal proof of this claim.

Proposition 1. *The* layer placement problem *is an NP-hard optimization problem.*

Proof. The proof is done by providing a reduction of the traveling salesman problem (TSP), which is known to be NP-complete, to the outlined layer placement problem. The intuition of the subsequent reduction can be understood as follows. Compute nodes of the layer placement problem correspond to the set of vertices in the TSP and the communication latency between compute nodes corresponds to the edge weights (or distance) between vertices. The placement of the layers on the compute nodes then determines the order in which the corresponding vertices are visited — since each layer has a unique index that identifies its position in the DNN, the placement of a layer on a compute node hence determines the position of the corresponding vertex in the traveling salesman's tour.

Now, let an arbitrary instance of the traveling salesman problem (TSP) be given by the complete directed weighted graph $G = (V, E, w, v_{start})$, with vertices V, edges $E \subset V \times V$, the weight relation $w : E \to \mathbb{N}$, and the start point of the route $v_{start} \in V$.

Next, the traveling salesman problem, which is the equivalent of finding the shortest hamiltonian cycle in the given graph, can be reduced to the problem of finding the shortest hamiltonian path by introducing an additional node v_{end} that duplicates v_{start} together with its corresponding edges and their weights:

$$
\begin{aligned}
G' &= (V', E', w', v_{start}, v_{end}) \\
V' &= V \cup \{v_{end}\} \\
E' &= E \cup \{(v, v_{end}) | (v, v_{start}) \in E\} \cup \{(v_{end}, v) | (v_{start}, v) \in E\} \\
w'(v_i, v_j) &= w(v_i, v_j), \quad \forall (v_i, v_j) \in E \\
w'(v, v_{end}) &= w(v, v_{start}), \quad \forall (v, v_{start}) \in E \\
w'(v_{end}, v) &= w(v_{start}, v), \quad \forall (v_{start}, v) \in E
\end{aligned}
$$

Then, from G' we construct an instance of the *layer placement problem* as follows. Let the resource landscape be defined as follows:

$$
\begin{aligned}
T &= 1 \\
N &= V' \\
m^{v_{start}} &= 3 \\
m^{v_{end}} &= 2 \\
m^{v} &= 1, \quad \forall v \in V' - \{v_{start}, v_{end}\} \\
c^{v} &= 1, \quad \forall v \in V' \\
b^{v_i v_j} &= 1 \\
e^{v_i v_j} &= w'(v_i, v_j), \quad \forall (v_i, v_j) \in E' \\
n_{input} &= v_{start}
\end{aligned}
$$

Let the layers be defined as follows:

$$\begin{aligned} L &= |V'| \\ m_{l_1} &= 3 \\ m_{l_L} &= 2 \\ m_{l_i} &= 1, \quad \forall i \in \{2 \dots L-1\} \\ c_{l_i} &= 1, \quad \forall i \in \{1 \dots L\} \\ p_{l_i} &= 0, \quad \forall i \in \{1 \dots L-1\} \\ p_{l_L} &= 1 \\ b_{l_i} &= 0 \quad \forall i \in \{1 \dots L\} \\ b_{input} &= 0 \end{aligned}$$

Due to $m^{v_{start}} = 3$ and $m_{l_1} = 3$ as well as $m^{v_{end}} = 2$ and $m_{l_L} = 2$, any solution to the layer placement problem must assign layer 1 to v_{start} and layer L to v_{end} respectively, since any other assignment would violate the RAM constraints.

The communication delays between the compute nodes directly correspond to the edge weights of the corresponding edges in the TSP instance. Without loss of generality, let $v_i, v_j \in V$ (with $v_i \neq v_j$) be two arbitrary nodes of the TSP and let l_k, $k \in \{1 \dots L\}$, be an arbitrary layer. Then the communication delay between v_i and v_j would be:

$$\begin{aligned} \frac{b_{l_k} \times 8}{b^{v_i v_j}} + e^{v_i v_j} &= \frac{0 \times 8}{1} + w'(v_i, v_j) \\ &= w'(v_i, v_j) \end{aligned}$$

Hence, for this problem instance, the objective of the layer placement problem corresponds to finding a path of length L from v_{start} to v_{end} with minimal communication delay, which is equivalent to finding the hamiltonian path with the minimal sum of edge weights in G'. □

6.2 Genetic Placement

The placement of layers on hosts shares similarities with the more general *service placement problem*. In their work, Skarlat et al. [SNS+17] address service placement in the context of fog computing and provide a general framework to optimize placement decisions. Similar to the algorithm presented in Section 6.1, they employ an exact approach based on CPLEX and propose a heuristic based on a genetic algorithm to tackle larger problem instances.

In the previous section we showed that, similar to service placement, the layer placement problem is NP-hard in the general case. While the linear programming based approach

guarantees to find an exact solution, its computational complexity might lead to impractical execution times for bigger problem instances. Therefore, as an alternative, this section presents a heuristic based on a genetic algorithm, that is inspired by the approach studied by Skarlat et al. [SNS+17]. The implementation of this strategy is done on the basis of [FDG+12], a Python framework for evolutionary algorithms.

In general, genetic algorithms are characterized by (i) their chromosome representation, (ii) a fitness function, and (iii) the genetic operators, that are applied to each generation of chromosomes. In addition, our proposed approach employs two domain-specific adaptions to ensure that chromosomes adhere to the constraints that are outlined in the previous section. In summary, the genetic algorithm operates according to the procedure outlined in Algorithm 6.1, which is described in more detail in the following sections.

Algorithm 6.1: Simple Genetic Algorithm

```
1  N ← 1000;
2  create a random initial population consisting of N chromosomes;
3  calculate the fitness of all chromosomes in the initial population;
4  for generation ← 1 ... numberOfGenerations do
5  |  select N chromosomes from population based on their fitness;
6  |  create N new chromosomes by performing crossover on the current
   |   population;
7  |  mutate random genes in the chromosomes;
8  |  foreach chromosome ∈ population do
9  |  |  EnforceTierConstraint(chromosome);
10 |  |  EnforceAdjacencyConstraint(chromosome);
11 |  end
12 |  calculate the fitness of all chromosomes in the population;
13 |  fittestChromosome ← find fittest chromosome in population;
14 |  if fittestChromosome did not change for 10 generations then
15 |  |  return fittestChromosome
16 |  end
17 end
18 return fittestChromosome
```

6.2.1 Chromosome Representation

The length of the chromosome corresponds to the number of layers in the DNN model. The genes in the chromosome represent a certain placement of a layer on a specific compute node. Each gene is an integer value that corresponds to a unique index that identifies a node in the compute hierarchy. Hence, a chromosome can be characterized as an integer vector $\mathbf{x}$ with L elements, where each element $x_l, l \in 1 \dots L$ encodes the placement decision for its corresponding layer. By design, the structure of a chromosome ensures that each layer is assigned to exactly one node.

6.2.2 Fitness Function

The fitness of a chromosome determines whether it is considered for evolution of the population by breeding the next generation of chromosomes. Furthermore, the genetic algorithm has to ensure that only those chromosomes survive, that correspond to valid placements. In particular, chromosomes are subject to the same constraints that have been introduced in Section 6.1.2.

If any of these constraints is violated, a penalty P is added to the fitness of a chromosome, which ensures that the individual chromosome will not be considered for further evolution, due to its decreased fitness. For that purpose, the penalty is set to a value that is bigger than any valid solution of the layer placement problem:

$$P = L \times \left(\frac{\max\limits_{i \in 1 \dots L} c_{l_i}}{\min\limits_{n \in N} c^n} + \frac{\max\left(\max\limits_{i \in 1 \dots L-1} b_{l_i}, b_{input} \right)}{\min\limits_{n \in N} b^n} + \max_{n \in N, n' \in N} e^{nn'} \right) \tag{6.14}$$

The term $\delta(\mathbf{x})$ is used to denote whether one of the constraints, that are formulated in the previous section, is violated:

$$\delta(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \text{ violates a constraint} \\ 0 & \text{otherwise} \end{cases} \tag{6.15}$$

Finally, the actual fitness F of a placement $\mathbf{x}$ is then computed as:

$$F(\mathbf{x}) = \sum_{i \in 1 \dots L} p_{l_i} \left(C_{l_i} + B_{l_i} \right) + B_{input} + \delta(\mathbf{x}) P \tag{6.16}$$

The terms C_{l_i} and B_{l_i}, (with $i \in 1 \dots L$), as well as B_{input} are semantically equivalent to the terms introduced in Section 6.1.3.

6.2.3 Genetic Operators

The evolution process of a genetic algorithm is defined by a set of genetic operators. A *selection* operator is used to determine which chromosomes should be considered for breeding chromosomes of the next generation. *Crossover* controls how selected chromosomes of the previous generation are combined to create their offspring. Finally, *mutation* introduces an additional source of evolutionary diversity by randomly changing some of the genes, that chromosomes inherit from their parents.

The used genetic operators as well as their parameterization are summarized in Table 6.4. Pre-experiments have shown that the chromosomes, that are generated by means of these

Genetic Operator	Value
Selection	Tournament of size 3
Crossover	80%, uniform, with 50% individual gene crossover probability
Mutation	2%, uniform, with 50% individual gene mutation probability

Table 6.4: Genetic Operators

traditional genetic operators, are likely to violate Constraints 6.4 and 6.3, which would prevent the genetic algorithm from finding valid solutions. Therefore, our algorithm employs two domain-specific mutations on top of the presented genetic operators, to enforce the compliance of the generated chromosomes with those constraints.

Algorithm 6.2 outlines how chromosomes are mutated such that they adhere to Constraint 6.4. Each gene in a chromosome corresponds to a placement decision for a particular layer. For example, the ith gene x_i of a chromosome $\mathbf{x}$ encodes the algorithm's placement decision for the DNN's ith layer. If the chromosome assigns layer i to an earlier tier than layer $i-1$, then the value of the chromosome's gene x_i is changed such that it corresponds to the value of gene x_{i-1}. This ensures that layers are never assigned earlier tiers than their successors.

Algorithm 6.2: Mutation for enforcing Constraint 6.4

Input: Chromosome $\mathbf{x}$
Result: Changed chromosome that is compliant with Constraint 6.4

```
1 Function EnforceTierConstraint(x) is
2     for i ← 2 to L do
3         if x_i ≠ x_{i−1} ∧ getTier(x_i) < getTier(x_{i−1}) then
4             x_i ← x_{i−1};
5         end
6     end
7     return the changed chromosome x
8 end
```

A second adaption to the chromosomes is outlined in Algorithm 6.3. It prevents placement decisions that would assign non-adjacent layers to the same node. To illustrate this, consider the exemplary placement decisions of the following chromosome:

$$\mathbf{x} = [1, 3, 2, 2, 3] \tag{6.17}$$

Here, the first layer would be placed on the node represented by the integer index 1. However, the second as well as the uppermost layer would be assigned to node 3, while the two interjacent layers are assigned to a different node. Hence, these placement decisions clearly violate Constraint 6.3.

After applying the adaptions described in Algorithm 6.3, this would result in the following updated chromosome, that evades this violation by assigning the third and fourth layer to node 3 as well:

$$\mathbf{x}' = [1, 3, 3, 3, 3] \tag{6.18}$$

Algorithm 6.3: Mutation for enforcing Constraint 6.3

Input: Chromosome $\mathbf{x}$
Result: Changed chromosome that is compliant with Constraint 6.3

```
1  Function EnforceAdjacencyConstraint(x) is
2      lastLayerMap ← ∅;
3      for i ← L to 1 do
4          if x_i ∉ lastLayerMap then
5              lastLayerMap ← lastLayerMap ∪ {x_i ↦ i};
6          end
7      end
8      i ← 1;
9      while i ≤ L do
10         currentNode ← x_i;
11         while i ≤ lastLayerMap(x_i) do
12             x_i ← currentNode;
13             i ← i + 1;
14         end
15     end
16     return the changed chromosome x
17 end
```

6.3 First-Fit Decreasing Placement

The optimization model that is introduced in Section 6.1 outlines classical resource constraints to ensure that layers do not exceed their host's resource capacities. In compute hierarchies that consist of powerful nodes with plenty of resources, these constraints are likely to be met implicitly. However, in usage scenarios that feature homogeneous IoT nodes that suffer from limited resources, the problem of placing layers on hosts is more related to the classical bin-packing problem.

The bin-packing problem has been shown to be NP-complete, however a number of heuristics have been proposed. Algorithm 6.4 outlines a strategy that is based on the *first-fit decreasing* heuristic. With this approach, the available compute nodes are first sorted in descending order, with respect to their compute power. Then, starting from

n_{input}, it proceeds by placing as many DNN layers as possible on more powerful compute nodes, in conformance with each node's memory constraints.

Algorithm 6.4: First-Fit Decreasing Placement

Input: Nodes N, Layers L
Result: Mapping of layers to nodes

```
placement ← ∅;
nextNode ← n_input;
sortedNodes ← sort nodes N by compute power;
availableMemory ← getMemory(n_input);
for l ∈ L do
    if getInMemorySize(l) > availableMemory then
        nextNode ← popLeft(sortedNodes);
        availableMemory ← getMemory(nextNode);
    end
    availableMemory ← availableMemory − getInMemorySize(l);
    placement ← placement ∪ {l ↦ nextNode};
end
return placement
```

6.4 Cloud-Only Placement

As described earlier, the traditional approach of deploying a DNN model simply places all layers in the cloud, ignoring variations in latency that arises from always having to send the DNN's input data all the way to the cloud. The strategy that is outlined in Algorithm 6.5 follows a similar approach and places all layers of the DNN model on a single node in the compute hierarchy.

First, the strategy determines the total compute demand that is induced by the computations of all layers. Then, it iterates over all nodes in the cloud tier in order to find the node that achieves the minimal estimated end-to-end latency. For this purpose, for each compute node it determines the compute latency for executing all layers, as well as the communication latency for sending the DNN's input data from the data source to the particular node. Finally, it nominates the compute node that minimizes the end-to-end inference latency.

Algorithm 6.5: Cloud-Only Placement

Input: Nodes N, Layers L
Result: Mapping of layers to nodes

$totalComputeDemand \leftarrow \sum_{l \in L} getExecutionProbability(l) \times getComputeDemand(l)$;
$candidateNode \leftarrow 0$;
$minLatency \leftarrow \infty$;
for $n \in N$ **do**
  if *node n is part of the cloud tier* **then**
    $computeLatency \leftarrow \frac{totalComputeDemand}{getComputeCapacity(n)}$;
    $inputLatency \leftarrow \frac{getInputSize(l_1) \times 8}{getNetworkThroughput(n_{input}, n)} + getNetworkLatency(n_{input}, n)$;
    $latency \leftarrow computeLatency + inputLatency$;
    if $latency < minLatency$ **then**
      $candidateNode \leftarrow n$;
      $minLatency \leftarrow latency$;
    end
  end
end
$placement \leftarrow \{l \mapsto candidateNode | l \in L\}$;
return *placement*

CHAPTER 7

Evaluation

In this chapter we present the evaluation of the proposed system based on different application scenarios. The experimental studies are conducted in a twofold approach. First, a simulation-based procedure is adopted to study the placement strategies, that are described in Chapter 6, in isolation. This will also help understand their performance on large-scale topologies that would otherwise go beyond the scope of a physical test-bed. Second, a feasibility study is performed on an actual test-bed that provides a heterogeneous environment covering an ARM-based Raspberry Pi node and a powerful x86-based cloud machine.

The remainder of this chapter is organized as follows. First, Section 7.1 gives an overview of the DNN models and datasets that are used for the experiments. Section 7.2 outlines the results of evaluating the placement strategies in different simulated scenarios. Finally, Section 7.3 describes the experiments that were conducted on a physical test-bed.

7.1 Models

The experiments are done on a set of DNN models that are based on state-of-the-art DNN architectures. For example, Resnet [HZRS16] and VGG [SZ15], which are popular architectures for image classification tasks, are part of the official PyTorch model zoo which contains a number of popular DNN models. For the purpose of the experimental studies, these models have been adapted to include an additional side-exit classifier.

Furthermore, these models also already come pretrained on the ImageNet [FFDL10] dataset. ImageNet is one of the largest and most popular datasets in the area of computer-vision and contains more than 14 million labeled images. The images that are contained in the dataset have been labeled by hand using *Amazon Mechanical Turk*[1].

[1] https://www.mturk.com/

For the purpose of the presented experiments the classification accuracy is not essential, therefore, the adapted model variants were only trained on a limited subset[2] of ImageNet covering approximately 400 images, in order to decrease training time. Since the backbone networks, that we used for our multi-exit variants, are already trained on the full ImageNet dataset, we only had to retrain the classifiers of the adapated models. In particular, the side-exit and final classifier were trained separately from each other, which corresponds to the training strategy outlined in Section 2.2.1. Hence, parameters of the backbone networks are retained, while only the exit classifiers are trained on the smaller dataset. In general, this is a common training strategy referred to as *transfer learning*, which applies a trained network's "knowledge" to a similar dataset [PSY+18]. For the purpose of the experiments, training of the classifiers follows a scheme that is similar to the one outlined in PyTorch's transfer learning tutorial[3]. It uses stochastic gradient descent with a learning rate of 0.001 and a momentum of 0.9 and as optimization criterion we chose cross entropy loss. Learning rate is decayed by a gamma factor of 0.1 every 7 epochs.

Finally, three different side-exit threshold configurations are chosen for the trained models, to be able to study different exit rate behaviors. The chosen configurations, together with the resulting exit rate behavior and classification accuracy, are summarized in tables 7.1 and 7.2. For both models, variant 0 serves as a baseline, that does not employ any side exit classifications due to its threshold being set to 0. The threshold configuration of variant 1 enables the evaluation of exit rates at a medium level. Finally, the thresholds of variant 2 have been chosen to study the effects of a high level of side exit classifications.

	Side exit threshold	Exit rate at side exit	Exit rate at main exit	Accuracy
Variant 0	0.0	0%	100%	0.9739
Variant 1	0.995	37%	63%	0.9346
Variant 2	0.9995	76%	24%	0.8366

Table 7.1: Resnet 152 variants

	Side exit threshold	Exit rate at side exit	Exit rate at main exit	Accuracy
Variant 0	0.0	0%	100%	0.9216
Variant 1	0.995	37%	63%	0.9281
Variant 2	0.9995	75%	25%	0.9216

Table 7.2: VGG 13 variants

As described in sections 4.3.2 and 5.5.2, the system performs an offline profiling step in order to estimate the resource requirements and other properties of the DNN model prior to the initial deployment. Table 7.3 summarizes the corresponding profiling information for these models. For each model it lists the number of sequential layers, the positions of

[2] https://download.pytorch.org/tutorial/hymenoptera_data.zip
[3] https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

its side-exit, its in-memory size and on-disk size, as well as its compute demand in the form of GFLOPs.

Model	# Layers	Side-Exit Position	In-Memory Size (in MB)	On-Disk Size (in MB)	GFLOPs
Resnet 152	57	15	593	234	11.6
VGG 13	35	20	1204	994	11.4

Table 7.3: Profiling summary of models for experiments

The profiler collects the according profiling information on a per-layer basis. Figures 7.1 and 7.2 illustrate the profile of each layer of the used variants of Resnet 152 and VGG 13.

Figure 7.1a shows the size of the intermediate results that are passed between layers of Resnet 152. In particular, each bar represents the size of the input tensor that is passed to each layer i.e, the bar of the first layer represents the size of the input images to be classified. These correspond to the images from the ImageNet dataset which are scaled to a size of 256x256 and contain 3 color channels, each accounting for a tensor size of 0.6 MB. After the first layer, which performs a convolution, the size of the intermediate results, that are passed to the next layers, increases by a factor of more than 5. After a series of layers that alternate between convolutions and ReLU operations, the size of the intermediate results decreases at layers 8 and 17. The intermediate results that are passed between the next layers reach 0.8 MB and are finally downsampled to 0.4 MB before layer 53.

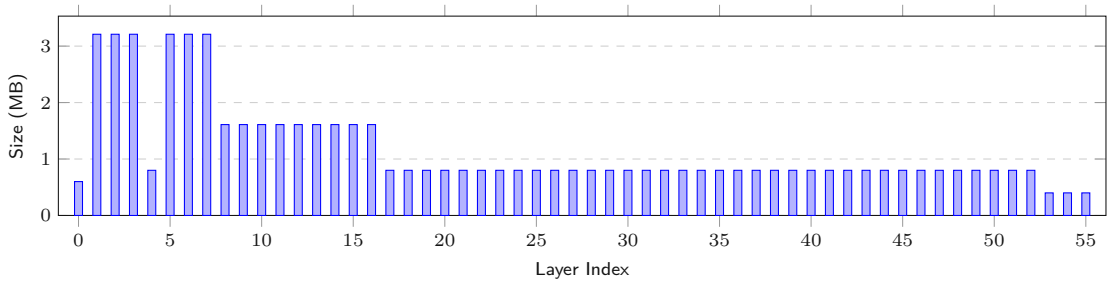
Figure 7.1b depicts the compute requirements at each layer. The number of floating point operations that are performed at each layer is distributed quite evenly over the neural network. Interestingly, the side-exit classifier at layer 15 as wells as the final classifier at layer 55, which correspond to pooling operations followed by a fully connected layer, seem to account for a minor fraction of the performed compute workload, averaging at 0.01 MFLOPS.

Likewise, the memory requirements of the layers (i.e. the size of the layers' learned parameters) are distributed relatively evenly as well, as can be seen in Figure 7.1c.

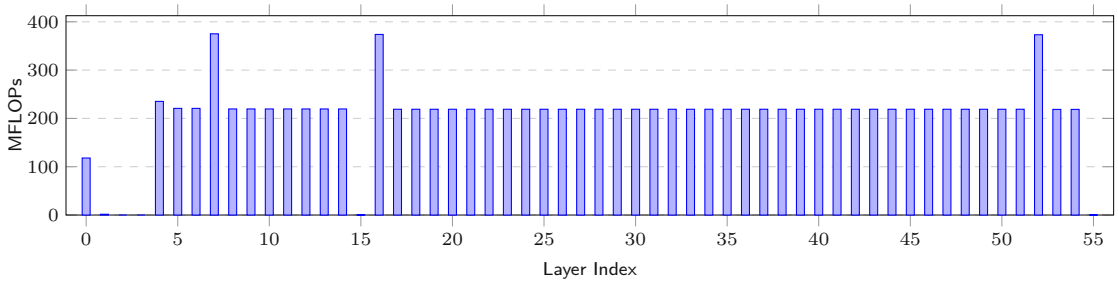
Figure 7.2a illustrates the size of the intermediate results in VGG 13. Again, the input to the first layer is 0.6 MB, as determined by ImageNet. Pooling operations at layers 4, 9, 14, and 19 significantly reduce the size of intermediate results that are passed to upper layers.

As shown in Figure 7.2b, the compute workload is not distributed as equally between layers as in Resnet. The major part of the computation is performed by the lower half (i.e. layers 0 to 17) of the neural network.

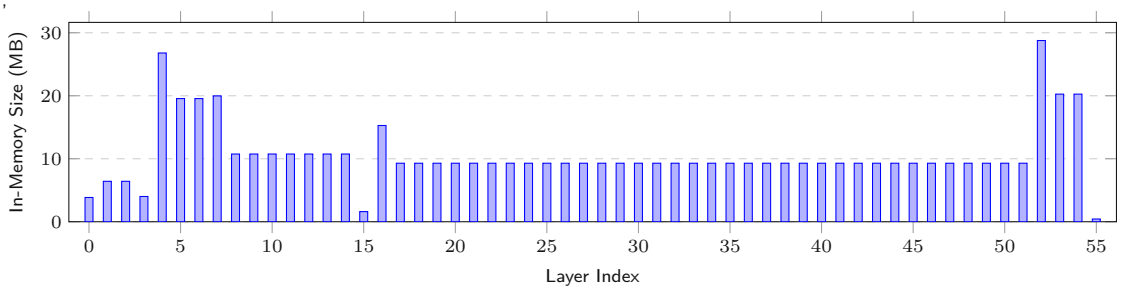
Finally, Figure 7.2c illustrates the memory overhead of the layers in VGG 13. Clearly, the learned parameters in the side-exit classifier at layer 20, as well as the layers leading



(a) Size of intermediate results



(b) Estimated compute requirements of layers



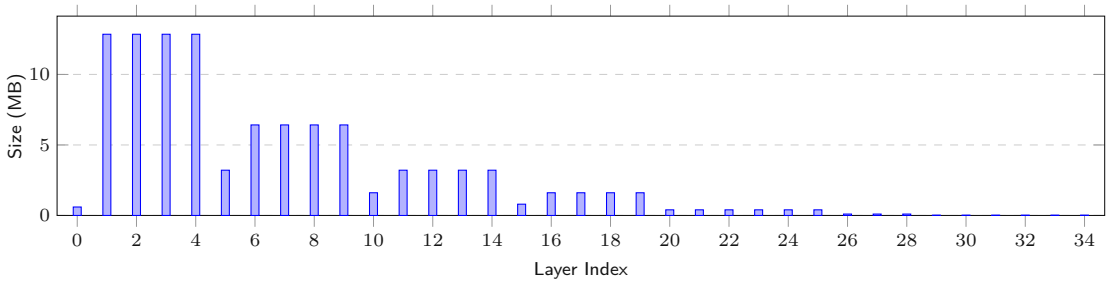
(c) Estimated memory requirements of layers

Figure 7.1: Profiling output for Resnet 152

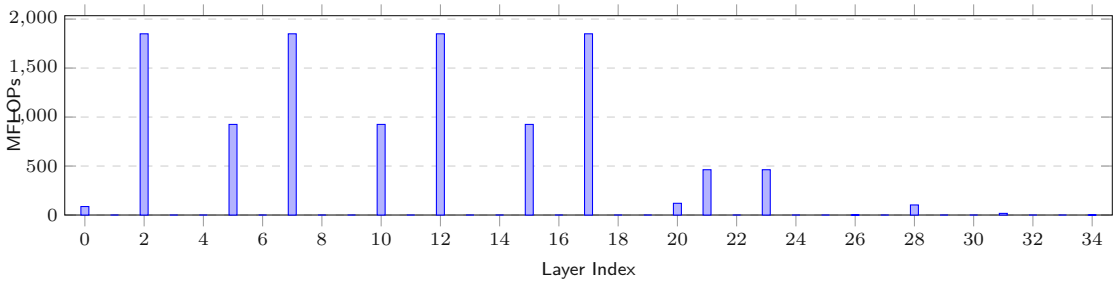
up to the final classifier starting at layer 28, account for the largest portion of the model's memory requirements.

7.2 Placement Strategies in Isolation

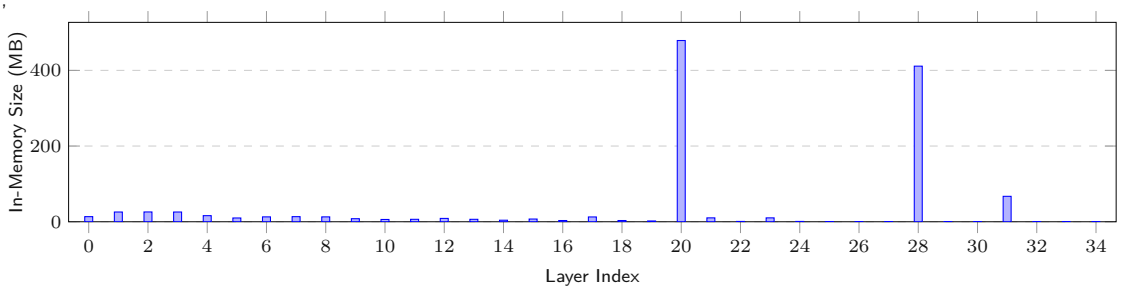
As described in sections 4.3.4 and 5.5, the system's scheduler drives the deployment of DNN models and decides the placement of layers on compute hosts based on a user-chosen placement strategy. In Chapter 6, we presented placement strategies, that aim at minimizing end-to-end inference latency. Furthermore, in Section 6.1, we outlined that exact placement is NP-hard, which motivated the design of a heuristic based on a genetic algorithm in Section 6.2. The computational complexity of the layer placement problem



(a) Size of intermediate results



(b) Estimated compute requirements of layers



(c) Estimated memory requirements of layers

Figure 7.2: Profiling output for VGG 13

suggests that execution time increases for bigger problem instances which is likely to exceed the scope of a physical test-bed. Therefore, in order to evaluate the characteristics of the proposed placement strategies, this section focuses on a simulation-based approach that evaluates the strategies in isolation from other system components. This is done by simulating compute hierarchies of varying sizes.

7.2.1 Simulated compute hierarchies

In particular, the designed evaluation scenario assumes a compute hierarchy that consists of three tiers: an IoT or end-device tier, an edge tier, and a cloud tier. Each tier is assumed to comprise a number of homogeneous compute nodes. The according node

types as well as their resource characteristics are listed in Table 7.4.

We assume that the compute nodes are distributed evenly among tiers, i.e., the tiers are assumed to contain equal amounts of compute nodes. The assumed maximal network throughput between nodes in each tier is listed in Table 7.5. For each pair of nodes, the throughput is varied randomly between 10 and 100 % of the listed values. Finally, the assumed network latencies between tiers are listed in Table 7.6.

Tier	Type	CPU	Compute Power (GFLOPS)	RAM (GB)	Disk Size (GB)
IoT	Raspberry Pi 3A+	ARM Cortex A53	4.93	0.5	0.5
Edge	Raspberry Pi 4B	ARM Cortex A72	13.5	4	16
Cloud	Generic Server	Intel Xeon E5-2620 v4	184	16	100

Table 7.4: Node Types (GFLOPS estimated based on Linpack benchmarks [WCERG])

	IoT	Edge	Cloud
IoT	1000	50	10
Edge	50	1000	100
Cloud	10	100	1000

Table 7.5: Network throughput (in MBit/s) between tiers

	IoT	Edge	Cloud
IoT	10	25	50
Edge	25	10	25
Cloud	50	25	10

Table 7.6: Network latency (in milliseconds) between tiers

7.2.2 Performed Measurements

The aim of this scenario is to compare the performance of the exact (ILP-based) algorithm, the heuristic genetic algorithm, and the traditional cloud-only placement. The following measurements are performed for each placement strategy on each simulated problem instance:

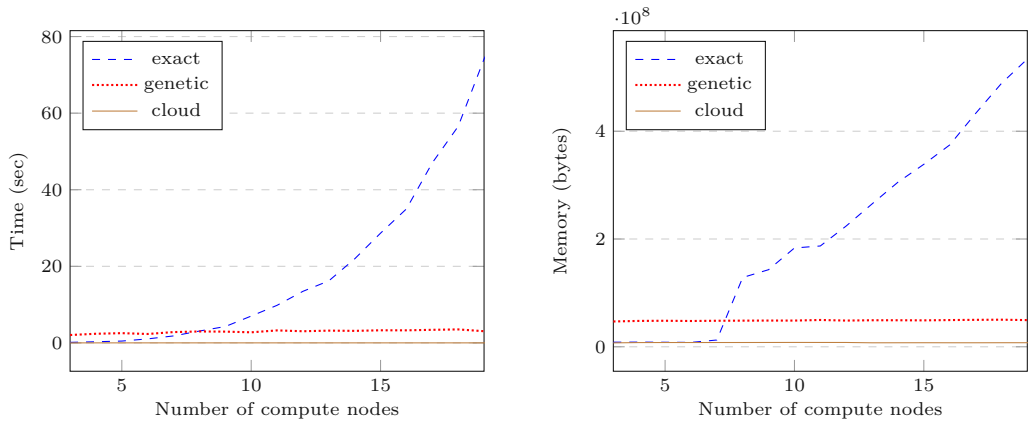
- **Execution time:** This corresponds to the amount of time it takes for a strategy to decide on a placement of layers on hosts. It is measured by summing the amount of time the scheduler and each of its child processes spends in user mode and system mode.
- **Memory consumption:** The memory consumption is estimated via `psutil` by measuring the *unique set size (USS)* of the scheduler and each of its child processes. According to the documentation of `psutil`, the USS "is probably the most representative metric for determining how much memory is actually being used by a process. It represents the amount of memory that would be freed if the process was terminated right now." [Com].
- **Predicted inference latency:** The inference latency corresponds to the strategies' optimization target, i.e., the end-to-end inference latency of the deployed DNN

with respect to the applied placement policy. In this simulation-based scenario, we can only obtain each strategy's *prediction* of the end-to-end inference latency which is based on the profiling and monitoring data they obtain from the system, as opposed to the *actual* inference latency, that can only be measured by performing actual inference requests on a physical compute hierarchy.

- **Predicted inference latency loss:** By definition, the exact algorithm obtains the placement that is characterized by the minimal predicted inference latency, i.e., the other strategies cannot provide a better placement without violating correctness constraints. Hence, the placements that are obtained via the exact algorithm, are used as a baseline to compare the quality of the placements suggested by each placement strategy. The latency *loss* is the relative performance of a strategy's predicted inference latency compared to the latency that was predicted by the exact placement strategy.

- **Number of split points:** A placement strategy might decide to split a model into partitions in order to respect correctness constraints or to minimize inference latency. Hence, instead of assigning all the layers of a DNN model to a single node, a strategy's placement decision might assign layers to different nodes. For example, if a strategy decides to distribute its layers over two distinct compute nodes, then the placement is characterized by one split point. Different strategies might lead to placements with different split points.

7.2.3 Results



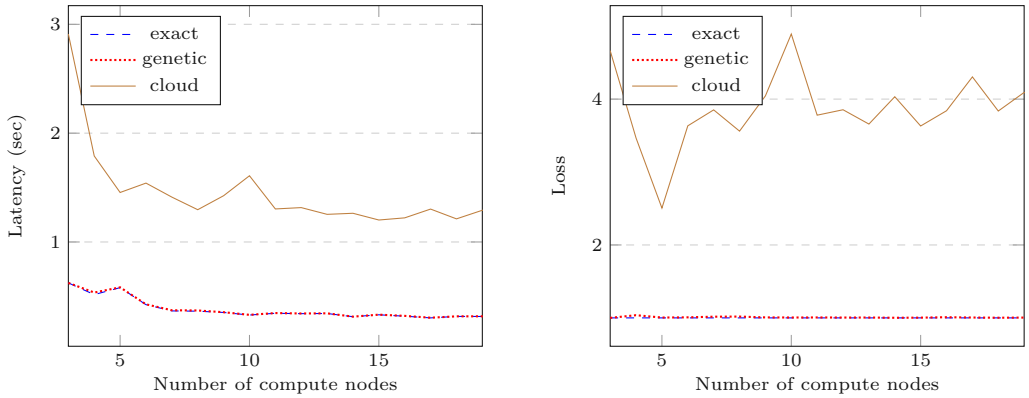
(a) Execution time (b) Memory consumption

Figure 7.3: Resource overhead for Resnet 152 variant 1

As suggested by the computational complexity of the exact algorithm, its execution time increases with the size of the problem instances. Also, the time it takes the genetic algorithm to provide a solution increases slightly, but at a much slower rate than for

the exact algorithm. Furthermore, the genetic algorithm is bound by a fixed amount of generations, therefore at a certain point its execution time is not expected to increase.

Memory consumption of the exact algorithm also increases with the size of the simulated compute hierarchies. For the genetic algorithm, memory consumption does not increase at a significant rate.



(a) Predicted inference latency (b) Predicted latency loss

Figure 7.4: Comparison of solution qualities for Resnet 152 variant 1

For all models and all tested algorithms, the predicted end-to-end latency decreases for bigger problem instances. This is due to the characteristics of the presented scenario, where problem instances and their resource properties are generated randomly. Hence, for bigger problem instances there is a higher probability that high-performing compute nodes are part of the hierarchy which entails that the algorithms tend to find better placements for bigger problem instances.

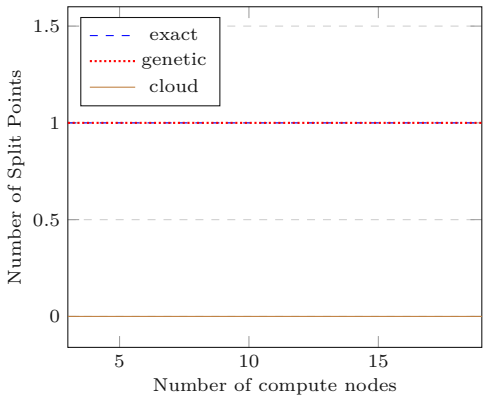


Figure 7.5: Average number of split points for Resnet 152 variant 1

Notably, the solutions of the exact and the genetic algorithm are characterized by similar partitioning decisions. On average, both algorithms place a single split point to distribute

the DNN inference workload over two distinct compute nodes. By design, the cloud-only placement does not place any split points.

Consequently, for bigger problem instances the genetic algorithm provides an acceptable trade-off between solution quality on the one hand and execution time and memory overhead on the other hand.

7.3 End-to-end Experiments

In the previous section, we presented experiments that aimed at studying the placement strategies in isolation from the rest of the system. In order to understand the system behavior on real-world scenarios, this section studies the system's performance on a physical test-bed.

7.3.1 Experimental Platform

The experimental setup that is used for the evaluation is depicted in Figure 7.6. The compute hierarchy comprises two nodes - a cloud node and a device node. The device node is represented by a Raspberry Pi 4B running on an ARM Cortex A72 CPU. The node runtime is installed on both node platforms, however, official PyTorch builds only exist for x86-based platforms. Therefore a manual build[4,5] of PyTorch for ARM had to be installed on the Raspberry Pi. The scheduler and the controller are hosted on a separate VM node.

We chose *Amazon Web Services (AWS)*[6] as a cloud platform to host the scheduler and the controller as well as the cloud-based compute node. For both cloud instances we chose `c5.xlarge` as EC2 type, which provides an acceptable trade-off between resource capabilities (compute performance and networking resources) and pricing.

The basic characteristics of the compute nodes in the physical test-bed are outlined in Table 7.7.

Tier	Type	CPU	RAM (GB)	Disk Size (GB)	Operating System
Device	Raspberry Pi 4B	ARM Cortex A72	4	32	Ubuntu Server 20.04
Cloud	AWS EC2 c5.xlarge	N/A	8	100	Ubuntu Server 18.04

Table 7.7: Types of compute nodes in the physical test-bed

In addition to the compute nodes, Table 7.8 lists further hosts and their role in the test-bed. In the addition to the compute nodes, the test-bed comprises a dedicated host that runs the controller as well as the scheduler component of the system runtime.

[4] https://qengineering.eu/install-pytorch-on-raspberry-pi-4.html
[5] https://github.com/Qengineering/PyTorch-Raspberry-Pi-64-OS
[6] https://aws.amazon.com

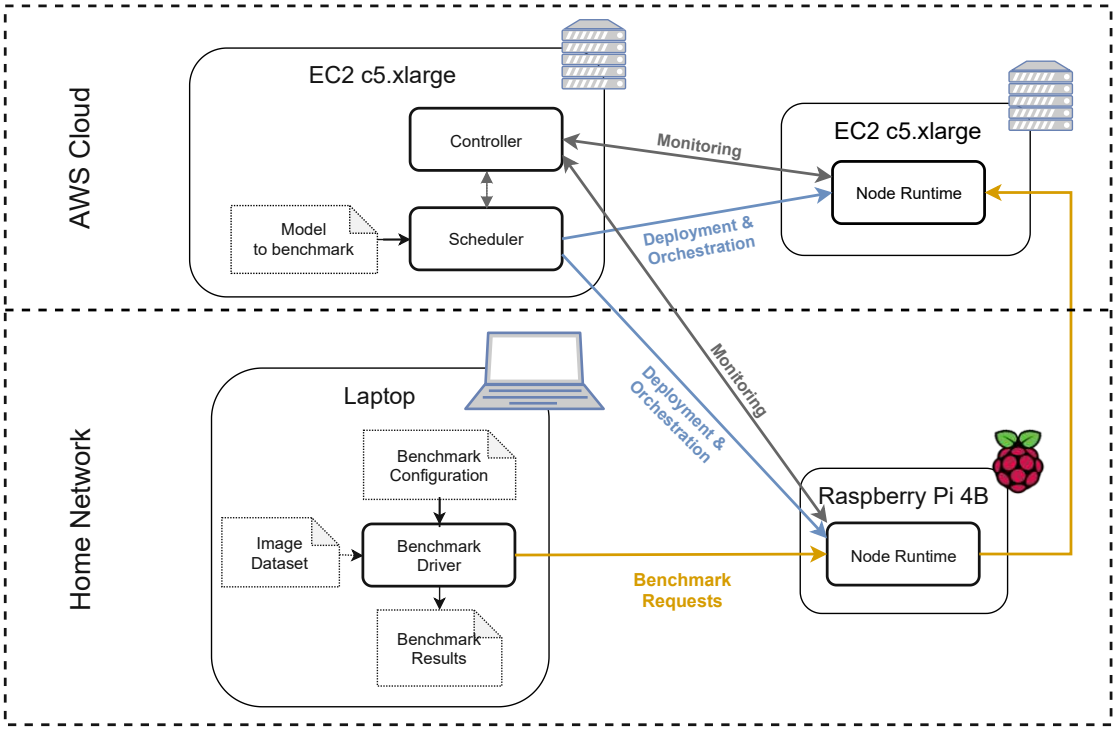


Figure 7.6: Experimental Setup

The actual benchmark functionality (i.e. performing inference requests and according measurements) is executed on a separate host.

Role	Type	CPU	Cores	RAM (GB)	Operating System
Controller/Scheduler	AWS EC2 c5.xlarge	N/A	4	8	Ubuntu Server 18.04
Benchmark Driver	Lenovo ThinkPad X1 Carbon 5th Gen	Intel® Core™ i7-7500U	4	16	Arch Linux

Table 7.8: Other test-bed components

Estimation of FLOPS

The scheduler and the employed placement rely on profiling and monitoring data in order to provide meaningful placement decisions. In particular, an accurate estimate of the compute capabilities in the form of floating point operations per second (FLOPS) is essential to be able to forecast the amount of time it takes to execute the computations that are performed by each layer in a given DNN model. Unfortunately, determining the FLOPS of a certain platform is far from trivial.

For the purpose of these experiments, we rely on the following procedure to estimate

the FLOPS for our experimental platform. The CLI of our system prototype provides a number of commands that can be used to provide such an estimation.

As a first step, this involves estimating the number of floating point operations (*FLOPs*) that are executed by a reference model when performing inference for a single input sample. For example, the following command can be used to execute the system's offline profiler for a given model and print the profile information:

```
addnn profile show $PATH_TO_MODEL
```

As part of the reported profiling estimation, this command also provides an estimate of the number of floating point operations (*FLOPs*) that are executed by the given model when it classifies a single input sample.

Next, we determine the throughput (as number of classified samples per second) when performing inference on the compute nodes. For that prupose, the following command can be issued on each compute node (in our case on the Raspberry Pi and on the cloud node), which will report the measured inference throughput:

```
addnn profile local-random-throughput --iterations=10 $PATH_TO_MODEL
```

More concretely, this command will repeatedly classify a specified number of randomly generated input samples using the given model (the number of classified samples is determined by the `--iterations` parameter which is 10 in the given example). Based on the number of classified samples and the elapsed time, it reports the estimated inference throughput on the current host.

Given the estimated *FLOPs* and actual throughput for the reference model, the floating point operations per second (*FLOPS*) for the target platform could then be estimated as follows:

$$FLOPS = FLOPs * throughput \tag{7.1}$$

Concretely, the Resnet 152 variants that are used in the experiments account for 11.557193728 GFLOPs according to our profiler. The measured throughput on the cloud instance is 4.6 samples/sec (based on the above command using 100 iterations). Throughput on the Raspberry Pi is 0.5 samples/sec. Hence the estimated compute capacity of the cloud instance is approximately 53 GFLOPS. The estimated compute capacity of the Raspberry Pi is approximately 5.8 GFLOPS.

These estimations of the FLOPS of our experimental platform are used as inputs for the subsequent experiments.

7.3.2 Overhead of System Components

The operation of the system components comes at a certain cost. For example, the overhead of operating the scheduler depends on the employed placement strategy. The

corresponding execution time and memory consumption was already discussed extensively in Section 7.2.

The scheduler and the controller can be operated independently from the rest of the system. In particular, they can be run on independent cloud VMs, whose resource characteristics can be adapted flexibly to the actual needs of the components. The node-runtime, however, is bound by the resource constraints of the actual compute nodes. For powerful cloud nodes this is less of a problem, but constrained devices, such as IoT nodes, impose hard limits on the resources that are available to the system's operation.

Figure 7.7 shows the memory overhead of the node runtime both on the cloud node as well as on the device node of the employed test-bed. Measurements were done with the Resnet 152 variant presented in Section 7.1. The model has a size of 234 MB when serialized to disk. The memory overhead of its layers was estimated (by the profiler) as 593 MB.

The node runtime's overhead is 71 MB on the device node and 132 MB on the cloud node at idle state and when no layers are loaded into memory. On the other hand, the overhead is 436 MB and 482 MB respectively, when all layers have been activated and reside in the node's memory. From this it follows, that the model's actual in-memory size is at maximum 350 MB (i.e. $482 - 132$) when deployed on the cloud VM or 362 MB (i.e. $436 - 71$) when deployed on the Raspberry Pi 4B.

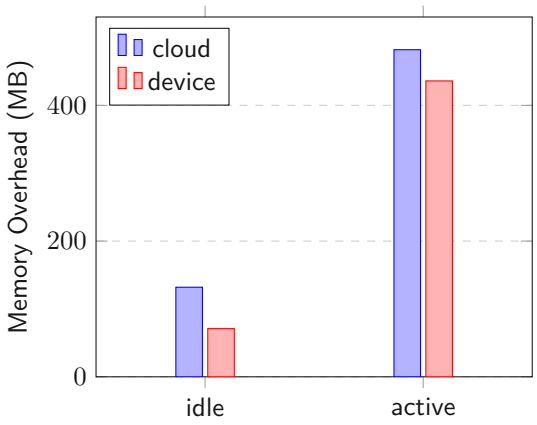


Figure 7.7: Memory overhead of node runtime

7.3.3 Compute Latency

In order to estimate how long it takes to execute compute workloads for the given models when no communication is involved, we studied the inference latency when executing all layers on a single compute node. Figure 7.8 shows the inference latency of the three variants of Resnet 152 that are used for the experiments. Variant 0 (which does not use the side-exit classifier and always classifies at its final layer) achieves an inference latency

of 0.2 s on the cloud node and 2.13 s on the Raspberry Pi. The other two variants, with side-exit rates of 37% and 71% respectively, achieve much lower inference latencies.

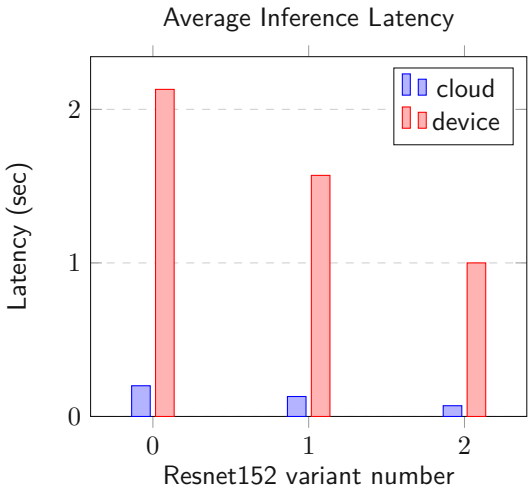


Figure 7.8: Compute latencies for Resnet152 variants

7.3.4 Inference latency at various static network conditions

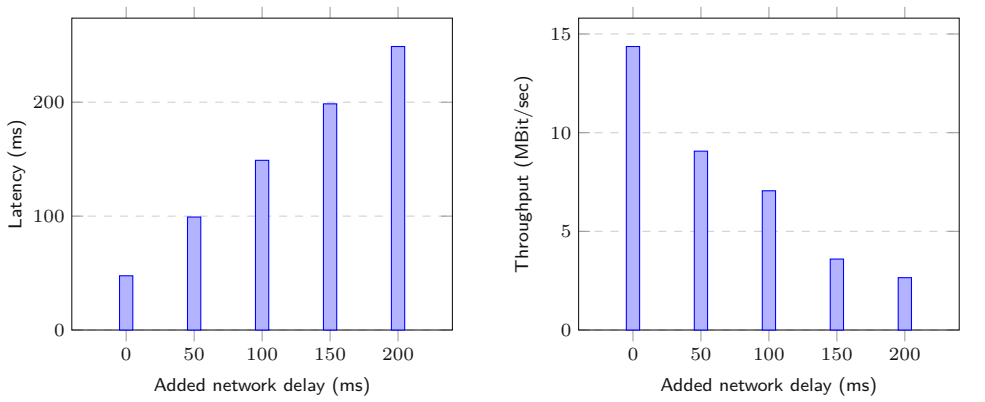
This scenario is focused on evaluating the system under constant environmental conditions. The goal of this scenario is to evaluate the inference latency of the system when using the exact placement strategy, compared to the traditional cloud-centric and device-only DNN execution. Measurements will be done for five different types of network conditions that are induced based on the `tc` tool. Specifically, we use `tc`'s `qdisc` sub-command to configure different levels of network delay on the device node, in order to simulate decreasing network quality between the compute nodes.

In a first step, the benchmark driver starts the controller in the cloud. It then configures the desired network delay on the device node and initializes the node runtime on each compute node. After the compute nodes successfully registered with the controller and submitted the necessary monitoring information, the benchmark driver starts the scheduler in order to deploy the model of interest to the compute nodes and activate the layers based on the exact placement strategy.

As soon as the scheduler deployed the DNN, applied its placement decision, and the nodes are ready to receive inference requests, the benchmark driver will start by issuing 5 sequential requests to warm up the system. This phase is then followed by the actual benchmark period during which the benchmark driver issues 20 sequential inference requests. For each inference request, the request timestamp as well as the response timestamp are recorded in order to calculate the respective inference latency per request.

Network Conditions

Since this scenario relies on a test-bed that covers components in a home network and the AWS cloud, the benchmarks rely on infrastructure that might not allow to reproduce environmental conditions in an exact manner for each of the performed benchmarks. Therefore, Figures 7.9a and 7.9b illustrate the network conditions that were observed at the beginning of the performed benchmarks. In particular, Figure 7.9a shows the network latency for each of the five `tc` configurations that are used for the benchmarks. When no additional network delay is configured, the measured network latency accounts for 50 ms, and for each of the network delay configurations it clearly shows how the latency increases accordingly. Likewise, Figure 7.9b shows the upload throughput from the Raspberry Pi device node to the cloud node for each of the configured network conditions. Starting from approximately 15 MBit/s, the upload throughput decreases significantly with the configured network delay levels.



(a) Network Latency (b) Network Throughput (device → cloud)

Results

The following figures illustrate the benchmark results for the Resnet 152 and VGG 13 variants for the described scenario. For each model variant, the figures show both the actual inference latency, as measured by the benchmark driver, as well as the predicted inference latency, as calculated by the scheduler, based on the model's profile and the monitoring data obtained from the controller and the compute nodes.

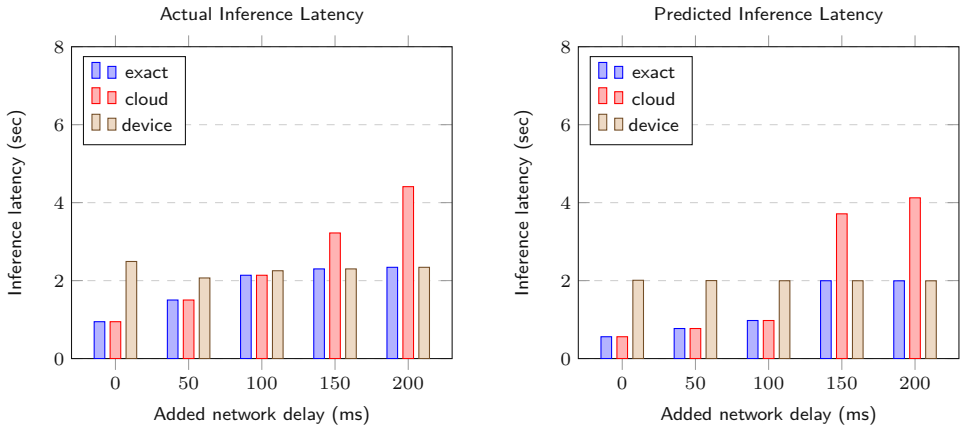


Figure 7.10: Resnet152 variant 0 – actual vs. predicted inference latency

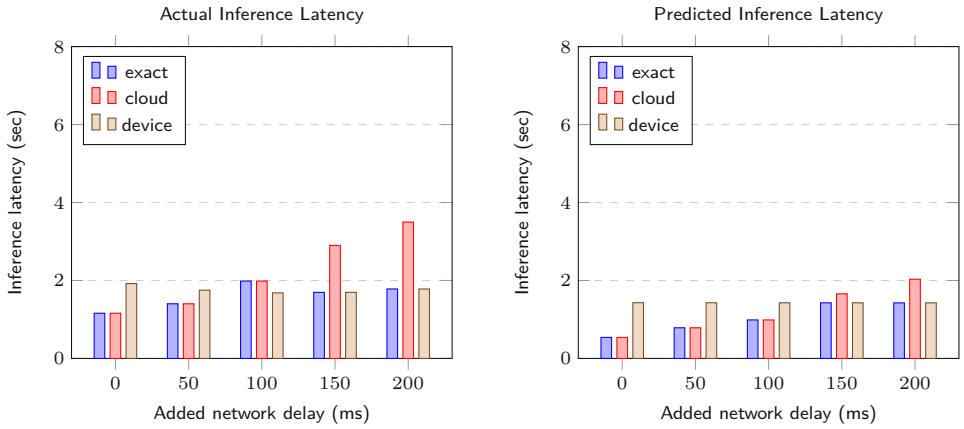


Figure 7.11: Resnet152 variant 1 – actual vs. predicted inference latency

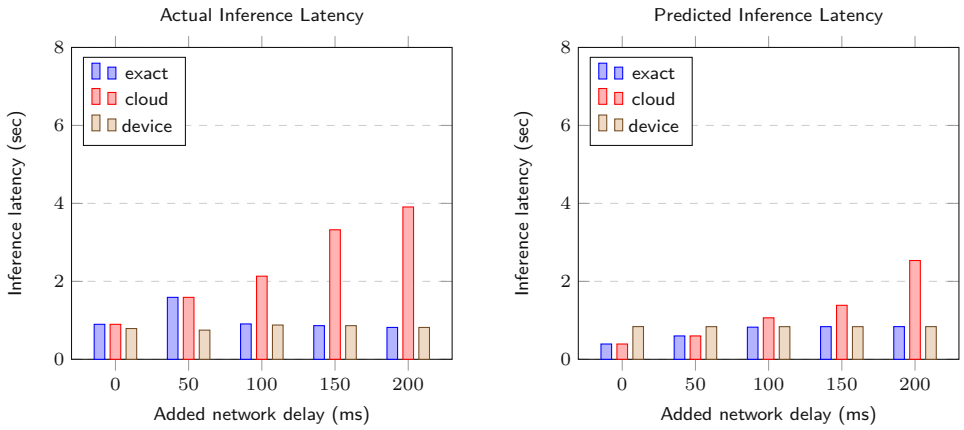


Figure 7.12: Resnet152 variant 2 – actual vs. predicted inference latency

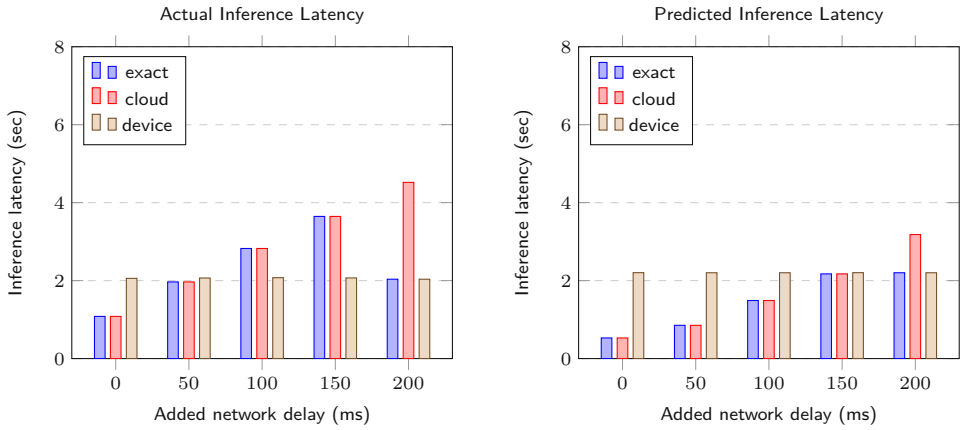


Figure 7.13: VGG13 variant 0 – actual vs. predicted inference latency

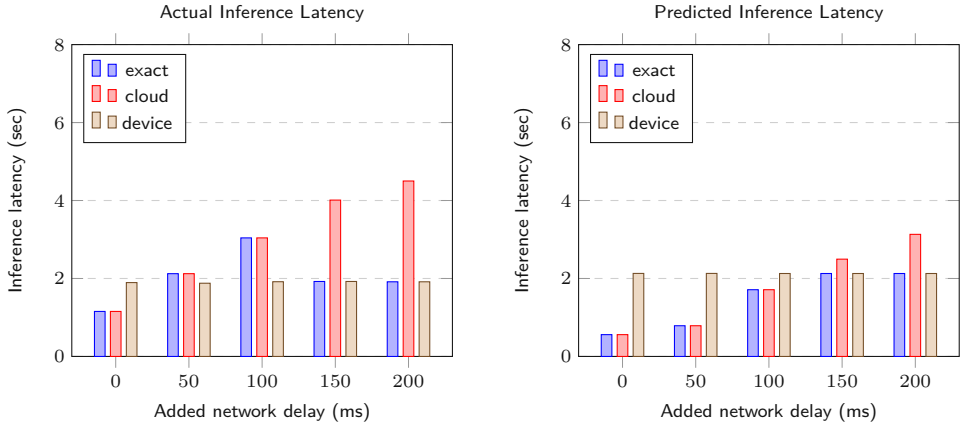


Figure 7.14: VGG13 variant 1 – actual vs. predicted inference latency

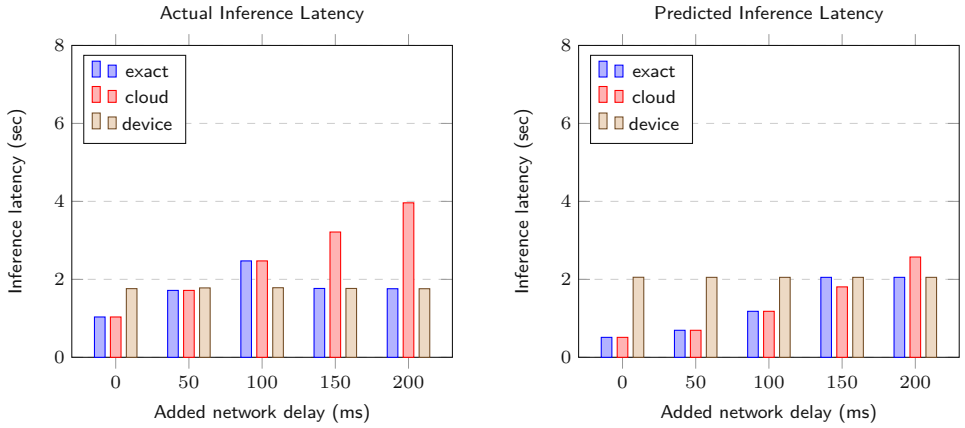


Figure 7.15: VGG13 variant 2 – actual vs. predicted inference latency

Discussion of Results

The system clearly identifies the different environmental conditions and the scheduler decides for according placements when using the exact placement strategy.

However, partitioning was performed for none of the used versions of Resnet 152 and VGG 13 under the tested network conditions, i.e., all layers of the models were either placed on the cloud node or on the device node. In the case of Resnet, this can be attributed to the structural properties of the DNN layers. As can be seen in Figure 7.1a, the input samples are upscaled significantly by the initial layers. The size of the intermediate results then exceeds the initial size of the network's input samples. Hence, the partitioning of the DNN at intermediate layers might actually increase the communication cost, which does not encourage partitioning decisions, since the placement of a split point could induce an increased inference latency. In the case of VGG, a split point at layer 20 could actually decrease the communication cost, however, Figure 7.2b suggests that the preceding layers induce a big part of the model's computational overhead. This in turn would impose a higher computation latency when executing these layers on the less powerful device node, which again could lead to an increased end-to-end inference latency.

From this we conclude, that the model architectures of Resnet and VGG are not eligible candidates for partitioning in the analyzed execution scenarios. However, the following two structural properties might facilitate the partitioning of other DNN architectures. First, the early layers of a DNN model should downscale the size of the intermediate results, in order to decrease communication cost for partitioning points at intermediate layers. Second, the computational overhead of executing layers that precede a split point should not outweigh its respective decrease in communication latency.

Furthermore, the figures indicate that the scheduler's latency predictions, which are based on the monitored environmental conditions, differ from the actual inference latency of the performed benchmark requests. In particular, the difference between the predicted inference latency and the latencies that were actually observed by the benchmark driver increases with degraded network performance. For certain network conditions, this might hence result in sub-optimal placement decisions. Moreover, this behavior suggests that the actual throughput and latency levels of the RPC-based communication differs from the raw, monitored network throughput and latency levels. Since this divergence appears to increase in a linear manner, with regard to decreasing network quality, a basic approach to improve the precision of the end-to-end inference predictions could realign the monitored network conditions by a constant (configurable) factor, in order to adjust the scheduler's estimations of the communication overhead.

7.3.5 Adaptive System Performance

As opposed to the experiments outlined in the previous section, the following scenario is focused on studying the system's dynamic behavior (based on Resnet 152 variant 0), when confronted with changing environmental conditions. Again, measurements are done for different networking conditions, configured via `tc`. Specifically, the benchmark

scenario involves three epochs that last for approximately 60 seconds each. During the first epoch, no additional network delay is configured. Only at the start of the second epoch, the benchmark driver configures an additional network delay of 300 ms in order to severely degrade the network conditions between the device and the cloud node. In the third and last period, the network delay is reset to its original state.

After initializing the controller and the compute nodes, the benchmark driver starts the scheduler in order to perform the initial DNN placement. It then starts the first epoch and continuously issues inference requests in a sequential manner until all epochs are finished. As opposed to the experiments in the previous section, the scheduler is now configured to trigger the placement algorithm repeatedly after a configurable time-interval. This means it continuously contacts the controller to obtain the most recent monitoring information about the state of the compute hierarchy. In response to changed conditions, the scheduler might then decide to change the placement. Furthermore, since the experiments involve measurements that are performed by separate components on separate hosts, all hosts were configured to use similar NTP servers.

Results

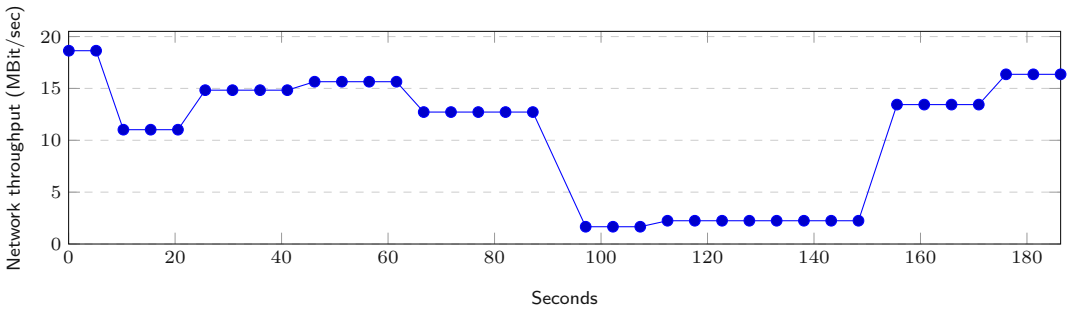


Figure 7.16: Network throughput over time

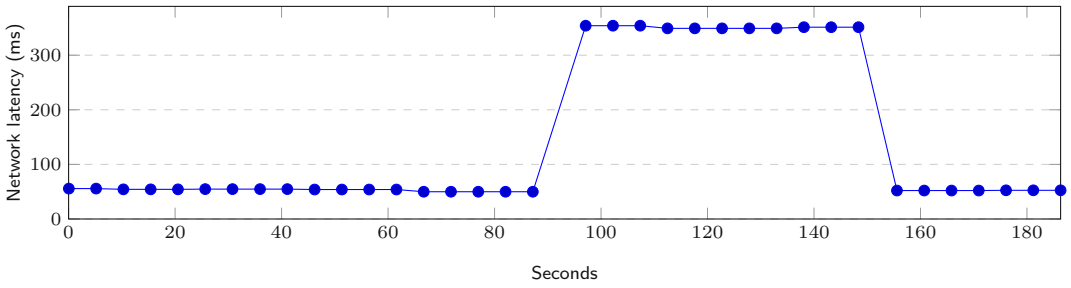


Figure 7.17: Network latency over time

The network conditions, as observed by the scheduler, are depicted in figures 7.16 and 7.17. Clearly, during the first epoch, the upload throughput between the device and the cloud node remains at relatively high levels between 15 and 20 MBit/s, with a constantly low latency of about 50 ms. After the start of the second epoch, when an additional

network delay of 300 ms is configured on the device node, the scheduler does not observe the changed network conditions immediately. The full decrease of the network throughput to approximately 2 MBit/s as well as the major increase in network latency is observed shortly before second mark 100, i.e., it took the system nearly 40 seconds to fully observe the degraded network conditions. Likewise, after the network conditions are reset at the start of epoch three at second 120, the improved network conditions are not observed immediately.

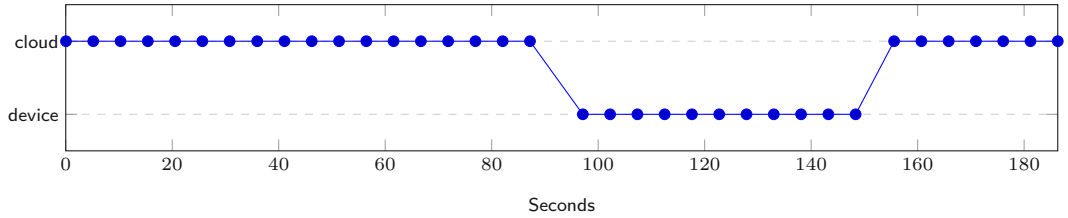


Figure 7.18: Placement over time

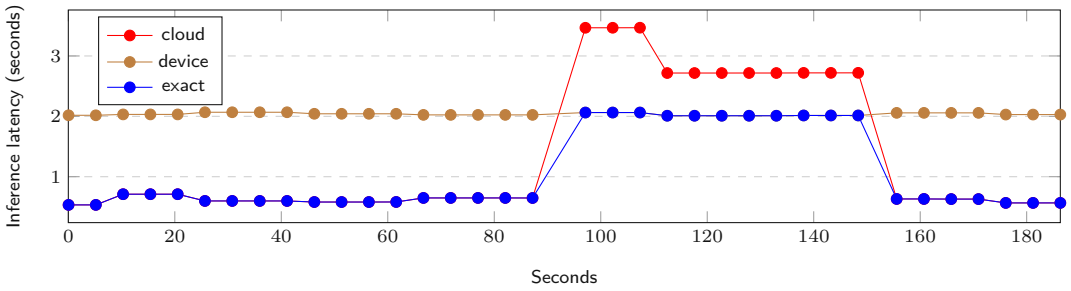


Figure 7.19: Predicted end-to-end inference latency over time

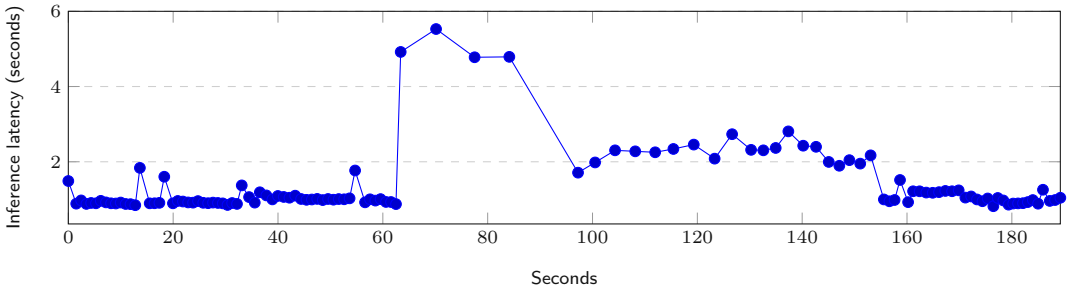


Figure 7.20: Measured end-to-end inference latency over time

The placement decisions of the scheduler, based on the exact placement strategy, are illustrated in Figure 7.18. In this scenario, the scheduler solely alternates between cloud-only and device-only placement and never partitions the model. The predicted inference latency for each placement decision, based on the observed environmental conditions, is shown in Figure 7.19. It compares the scheduler's predictions based on the exact placement strategy, to cloud- and device-only placements. Each point in these two

graphs represents a placement decision that was made by the scheduler, employing the exact placement strategy. Finally, Figure 7.20 shows how the *actual* inference latency evolved over time. Here, each point represents the issuance of an inference request by the benchmark driver.

CHAPTER 8

Conclusion

The use of edge computing as a platform for distributed DNN inference is an active area of research. Recent work proposes new neural network architectures that facilitate the distribution of DNN workloads in such environments. In addition to the classifier on a DNN's final layer, these architectures introduce side-exit classifiers at intermediate layers. With this approach it is possible to obtain inference results at earlier points in the network and thereby reduce the compute overhead, which is critical for the operation on more constrained devices.

This thesis follows a recent line of research, that uses this novel architecture to shift DNN computations towards less powerful devices at the edge of the network, to improve user experience, while limiting the sacrifice of inference accuracy. In contrast to related work, which is more focused on algorithmic aspects to optimize the distributed execution of DNNs, this thesis puts a focus on the design aspects that enable the implementation of an extensible system framework. In the following sections, we summarize the core contributions of this thesis and conclude by providing an outlook for possible future research directions and extensions of this work.

8.1 Contributions

This thesis is centered around a number of research questions, which are outlined in Chapter 1. The core contribution of this work is the design and implementation of a system for distributing inference of feed-forward DNNs, possibly with multiple exit classifiers, thereby addressing research question **RQ1**. Each host in the compute hierarchy operates a runtime environment that offers APIs for orchestration and execution of DNNs, as well as a component for monitoring the node's resource levels and network conditions. Compute nodes are required to register with a central controller, which maintains a global view on the compute hierarchy. Finally, a scheduler decides about the deployment and orchestration of a given DNN model over the available compute resources. With respect to

research question **RQ3**, the scheduler reorchestrates DNN inference in regular intervals, in response to changing environmental conditions. From a software architecture perspective, the scheduler offers a plugin framework, that allows system users to implement and apply their own algorithms for custom placement decisions. This way, different optimization goals can be pursued.

In response to research question **RQ2**, we propose a number of strategies that integrate with the plugin framework of the system's scheduler. First, we provide an exact algorithm that solves the placement problem to optimality. Based on CPLEX, the strategy is implemented in the form of an integer linear program, that aims to identify, amongst all valid solutions, a placement policy that minimizes end-to-end inference latency. In contrast to related work, which only focuses on distributing a DNN workload over a single device and single cloud node, this algorithm is able to find an optimal placement policy in a compute hierarchy of arbitrary size. We also show the exact placement of layers in such a system landscape to be an NP-hard combinatorial optimization problem, by providing a reduction for the travelling salesman problem. Due to this intractability, we also propose a heuristic approach for bigger problem instances, that is based on a generic genetic algorithm. Furthermore, we also provide a simplistic strategy that assigns all layers of a DNN to the most powerful host in a compute hierarchy, as well as a first-fit decreasing heuristic.

Finally, experimental studies evaluate the prototypical system implementation in simulation-based scenarios and on a physical test-bed. On simulated compute hierarchies, the exact placement clearly outperforms the traditional cloud-centric placement. A feasibility study on a physical test-bed confirms that the system is able identify efficient placements based on monitored environmental conditions.

The implementation of the system, including the proposed placement strategies, is done in Python and integrates with PyTorch. The complete code-base is made available as open-source[1].

8.2 Future Work

The proposed system features an extensible design based on flexible APIs and interfaces, that can serve as a platform for further research. This section describes certain aspects of the current work that can be improved and discusses possible directions for future work.

8.2.1 Multi-tenant use

At the moment, a basic level of multi-tenancy can already be achieved by deploying the runtime environment in the form of a container. With this approach, multiple instances of the system could be operated as separate containers, running in parallel on each compute node.

[1] https://github.com/MatthiasJReisinger/addnn

However, the proposed design and prototype implementation does not yet explicitly address the parallel use of the system by multiple tenants at the same time. Currently, it only handles placement and inference for a single DNN model on a given compute hierarchy. The parallel operation of multiple models requires the extension of the node runtime as well as enhanced scheduling and orchestration mechanisms. Ideally, such an extension considers fairness in terms of the allocation of available resources between the models of multiple tenants.

8.2.2 Confidence Threshold Tuning

Currently, the action space of placement strategies is restricted to the placement of layers on nodes and does not include the configuration of confidence thresholds at side-exits. In particular, the `Strategy` class defined in Listing 5.20, which is the base type for all placement strategies, only allows its descendants to override a `compute_placement()` method that, as its name suggests, can only be used to compute the placement of the layers in the compute hierarchy.

In order to illustrate how the action space could be extended in the context of the current system implementation, Listing 8.1 outlines a possible extension to the `Strategy` base class, that is part of the scheduler's plugin framework. In contrast to the original class in Listing 5.20, the `compute_policy()` method returns a list of `LayerPolicy` objects. In addition to the `placement` decision for the corresponding layer, a `LayerPolicy` also specifies an optional `confidence_threshold`, in case the layer is connected to an exit branch.

```
1  NodeIndex = int
2
3  class LayerPolicy:
4      def __init__(self, placement: NodeIndex, confidence_threshold: Optional[
       float]) -> None:
5          self._placement = placement
6          self._confidence_threshold = confidence_threshold
7
8      @property
9      def placement(self) -> NodeIndex:
10         return self._placement
11
12     @property
13     def confidence_threshold(self) -> Optional[float]:
14         return self._confidence_threshold
15
16
17 class Strategy(ABC):
18     @abstractmethod
19     def name(self) -> str:
20         raise NotImplementedError
21
22     @abstractmethod
23     def compute_policy(self, nodes: List[Node], layers: List[LayerProperties
       ]) -> List[LayerPolicy]:
```

```
24          raise NotImplementedError
```

Listing 8.1: Placement Strategy Base Class

8.2.3 Placement Strategies

The scheduler offers a flexible plugin infrastructure, which can serve as a platform to experiment with different placement algorithms. The algorithms, that are proposed by this thesis, demonstrate how to tune placement decisions with the aim to minimize DNN inference latency. Future work could focus on strategies that also target other aspects of DNN operation. For example, following an approach outlined by SPINN [LVA+20], placement strategies could employ multi-objective optimization for joint tuning of latency, accuracy, and energy-efficiency, as well as other user-defined application constraints.

8.2.4 Online profiling

As described in Section 5.5.2, the system currently relies on an offline approach to predict resource overhead that is induced by executing the layers of a given DNN model. For the purpose of estimating the compute overhead of each layer, our approach relies on the knowledge of each node's compute capabilities in terms of floating point operations per second (FLOPS). Unfortunately, manufacturing specifications of compute devices often lack indications of FLOPS and despite the existence of benchmark tool-chains such as LINPACK[2], determining the compute capacity for a specific target platform remains a difficult endeavor. Furthermore, the actual performance of a model might be dependent on further run-time mechanisms, which would not be covered adequately by this abstract estimation approach. For example, PyTorch's TorchScript interpreter might execute certain operations in parallel on a single CPU core based on the target platform's CPU threading capabilities[3].

Instead of relying on FLOPS as a measure of compute performance, a layer's run-time overhead might therefore be estimated more precisely by online profiling. In particular, this would be achieved by executing inference for a given model directly on each type of target platform that is used by the compute nodes. This could be done automatically by the controller or the scheduler at system startup or when a new compute node joins the compute hierarchy.

8.2.5 Fault tolerance

At the moment the proposed system only offers a very basic level of fault tolerance. If a compute node fails to continue operation, for example due to a loss of network connectivity, the controller's monitoring mechanism would detect that the node became unavailable and deregister it accordingly. As soon as the scheduler observes the changed

[2] http://www.netlib.org/linpack/

[3] https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html

compute hierarchy, it would reassign any DNN layers of the failed node to another host. Since the controller's monitoring agent as well as the scheduler are only retriggered in fixed time intervals, the system would be unresponsive for a certain period of time.

In order to respond more effectively to the failure of compute nodes, an approach similar to SPINN [LVA+20] could be adopted. In particular, this approach is based on the availability of side-exits at different stages of the neural network. The scheduler could decide to always place a side-exit at the input device node, which would enable to use earlier classification results at the expense of reduced accuracy, if the next layer at the next node cannot be reached in time. This mechanism would also require an adaption of the node runtime's DNN inference engine. Currently, the classification of a side-exit is neglected if it does not meet a user-specified exit threshold. For the purpose of fault tolerance, however, the node runtime could choose to remember the classification result of earlier exits. If the next layer does not respond in time, the node could use its local classification result, regardless of the corresponding threshold value.

This scheme can further be enhanced by a publish-subscribe mechanism. At the moment, DNN inference is implemented based on pull-based APIs in the node runtime. Instead, compute nodes could publish intermediate results of their layers as well as classification results of side-exits to a central message broker. Other nodes, can then subscribe to that broker to receive results asynchronously and choose to collaborate, if certain nodes do not respond in time.

8.2.6 Support for different compute devices

At the moment, the system is restricted to the use of CPUs when performing DNN operations. To improve performance, the use of additional compute devices should be considered. In particular, the node runtime could make use of GPU resources on more powerful host platforms. The generic design of the proposed APIs makes such an extension straight-forward, since it allows a transparent handling of different kinds of compute resources in an abstract manner. Besides adapting the DNN inference mechanism, this would also involve the extension of the resource monitoring agents, to track the availability and load levels of the GPU devices.

APPENDIX A

Running the Experiments

The implementation of the presented system implementation as well as the code for the experiments are available in the form of a Python-based command-line application. The CLI application is named `addnn`, its development is based on the dependency manager *Poetry*[1]. More information about the installation and the usage of this CLI tool-chain is available as part of the git repository at `https://github.com/MatthiasJReisinger/addnn`. The following sections shortly outline important commands that were used to run the experiments in Chapter 7.

A.1 Models

The DNN models that have been used for the experiments, as introduced in Section 7.1, have been created and trained via the commands outlined in Listings A.1 and A.2.

```
poetry run addnn example --model=resnet152_2exits --pretrained --dataset=
    imagenet-hymenoptera --dataset-root=./datasets/hymenoptera_data --epochs
    =25 --batch-size=4 resnet152_2exits_25epochs.pt
```

Listing A.1: Export & train Resnet 152

```
poetry run addnn example --model=vgg13_2exits --pretrained --dataset=imagenet
    -hymenoptera --dataset-root=./datasets/hymenoptera_data --epochs=25 --
    batch-size=4 vgg13_2exits_25epochs.pt
```

Listing A.2: Export & train VGG 13

The threshold levels of the side-exits, that are described in Section 7.1, have been configured for both models as outlined in Listing A.3 — in the concrete example, the side-exit threshold of the given model would be set to 0.99.

[1] `https://python-poetry.org/`

```
poetry run addnn thresholds set --exit 0 0.99 $PATH_TO_MODEL
```

Listing A.3: Exit threshold configuration

Next, after settings the treshold levels of a model's exit branches, the following command is used to learn the exit rates of each exit classifier:

```
poetry run addnn thresholds learn-exit-rates --dataset=imagenet-hymenoptera
    --dataset-root=$PATH_TO_DATASET --batch-size=1 $PATH_TO_MODEL
```

Finally, the profile information for a given model can be retrieved, for example, via the command in Listing A.4. This command was used to generate the profiles for Resnet 152 and VGG 13 listed in Figures 7.1 and 7.2.

```
poetry run addnn profile export --out=$CSV_FILE --format=csv $PATH_TO_MODEL
```

Listing A.4: Profiler

A.2 Placement Strategies in Isolation

Section 7.2 introduced experiments that study the placement strategies on a simulated compute hierarchy. The outlined exeriments were carried out via the command in Listing A.5.

```
poetry run addnn benchmark strategies vary-hierarchy-size --scenario=
    iot_edge_cloud --min-num-nodes=3 --max-num-nodes=20 --strategy=optimal --
    strategy=genetic --strategy=cloud --repetitions=10 --csv --seed=42 --step
    =1 $PATH_TO_MODEL
```

Listing A.5: Placement strategy benchmark

A.3 End-to-end Experiments

The testbed experiments that are outlined in Section 7.3 were performed in a completely automated manner. The corresponding functionality, which is referred to as *benchmark driver* in Figure 7.6, is made available as part of the implemented CLI tool-chain. In particular, the life cycle of the controller, scheduler, as well as the compute node runtimes are managed the benchmark driver.

Furthermore, the models that are used for the experiments are placed on the AWS EC2 instance that hosts the scheduler component, as outlined in Figure 7.6. The concrete directory were all models are stored is `/home/ubuntu/models/testbed/`, as can be seen in the command invocations that are outlined in Section A.3.2 below.

A.3.1 Node Runtime Configuration

As described above, the experiments that are introduced in Sections 7.3.4 and 7.3.5 manage the life cycle of the participating compute nodes. The corresponding information,

that is required to manage the node runtimes, is passed to the benchmark driver in the form of a configuration file.

Listing A.6 outlines the configuration that is used for the experiments regarding the different variants of Resnet 152, referred to as `rpi4-cloud-resnet152.yaml` in the commands below. The configuration for the experiments regarding VGG 13 is given in Listing A.7, referred to as `rpi4-cloud-vgg13.yaml` in the command invocations below.

```
nodes:
  - host: 2a02:8388:4240:c580:dea6:32ff:fe4d:d621
    port: 24242
    user: ubuntu
    compute_capacity: 5_800_000_000
    tier: 0
    is_input: true
    network_device: eth0
    addnn_executable: "/home/ubuntu/.local/bin/poetry run addnn"

  # c5.xlarge
  - host: 2a05:d018:d44:9100:659d:231:6406:f1b1
    port: 24242
    user: ubuntu
    compute_capacity: 53_000_000_000
    tier: 1
    is_input: false
    network_device: ens5
    ssh_key_path: ~/.ssh/aws-ec2.pem
    addnn_executable: "/home/ubuntu/.poetry/bin/poetry run addnn"
```

Listing A.6: Node runtime configuration for Resnet 152 benchmarks

```
nodes:
  - host: 2a02:8388:4240:c580:dea6:32ff:fe4d:d621
    port: 24242
    user: ubuntu
    compute_capacity: 5_200_000_000
    tier: 0
    is_input: true
    network_device: eth0
    addnn_executable: "/home/ubuntu/.local/bin/poetry run addnn"

  # c5.xlarge
  - host: 2a05:d018:d44:9100:659d:231:6406:f1b1
    port: 24242
    user: ubuntu
    compute_capacity: 68_500_000_000
    tier: 1
    is_input: false
    network_device: ens5
    ssh_key_path: ~/.ssh/aws-ec2.pem
    addnn_executable: "/home/ubuntu/.poetry/bin/poetry run addnn"
```

Listing A.7: Node runtime configuration for VGG 13 benchmarks

A.3.2 Inference latency at various static network conditions

In the following, we outline the commands that were used to execute the experiments that are introduced in Section 7.3.4.

```
poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-model-path=/home/ubuntu/models/testbed/resnet152_2exits_25epochs_0.pt --dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/hymenoptera_data --config=../masters-thesis/testbed/rpi4-cloud-resnet152.yaml --benchmark-duration=20 --num-layers=56 --seed=42 device_network_delay --delay=0 --delay=50 --delay=100 --delay=150 --delay=200
```

Listing A.8: Invocation for Resnet 152 variant 0

```
poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-model-path=/home/ubuntu/models/testbed/resnet152_2exits_25epochs_0995.pt --dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/hymenoptera_data --config=../masters-thesis/testbed/rpi4-cloud-resnet152.yaml --benchmark-duration=25 --num-layers=56 --seed=42 device_network_delay --delay=0 --delay=50 --delay=100 --delay=150 --delay=200
```

Listing A.9: Invocation for Resnet 152 variant 1

```
poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-model-path=/home/ubuntu/models/testbed/resnet152_2exits_25epochs_099.pt --dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/hymenoptera_data --config=../masters-thesis/testbed/rpi4-cloud-resnet152.yaml --benchmark-duration=25 --num-layers=56 --seed=42 device_network_delay --delay=0 --delay=50 --delay=100 --delay=150 --delay=200
```

Listing A.10: Invocation for Resnet 152 variant 2

```
poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-model-path=/home/ubuntu/models/testbed/vgg13_2exits_25epochs_0.pt --dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/
```

```
    hymenoptera_data --config=/home/matthias/projects/masters-thesis/testbed/
    rpi4-cloud-vgg13.yaml --benchmark-duration=25 --num-layers=35 --seed=42
    device_network_delay --delay=0 --delay=50 --delay=100 --delay=150 --delay
    =200
```

Listing A.11: Invocation for VGG 13 variant 0

```
1 poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:
    c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --
    controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-
    scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --
    scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-
    model-path=/home/ubuntu/models/testbed/vgg13_2exits_25epochs_1.pt --
    dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/
    hymenoptera_data --config=/home/matthias/projects/masters-thesis/testbed/
    rpi4-cloud-vgg13.yaml --benchmark-duration=25 --num-layers=35 --seed=42
    device_network_delay --delay=0 --delay=50 --delay=100 --delay=150 --delay
    =200
```

Listing A.12: Invocation for VGG 13 variant 1

```
1 poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:
    c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --
    controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-
    scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --
    scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-
    model-path=/home/ubuntu/models/testbed/vgg13_2exits_25epochs_2.pt --
    dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/
    hymenoptera_data --config=/home/matthias/projects/masters-thesis/testbed/
    rpi4-cloud-vgg13.yaml --benchmark-duration=25 --num-layers=35 --seed=42
    device_network_delay --delay=0 --delay=50 --delay=100 --delay=150 --delay
    =200
```

Listing A.13: Invocation for VGG 13 variant 2

A.3.3 Adaptive System Performance

The functionality of the experiments for studying the adaptive system behavior in Section 7.3.5 are also available as part of the implemented CLI tool-chain. The exact command invocation for the described experiment is outlined in Listing A.3.3.

```
1 poetry run addnn benchmark end-to-end --controller-host=2a05:d018:d44:9100:
    c0c:bca2:ec86:35f0 --controller-port=42424 --start-controller --
    controller-user=ubuntu --controller-ssh-key=~/.ssh/aws-ec2.pem --start-
    scheduler --scheduler-host=2a05:d018:d44:9100:c0c:bca2:ec86:35f0 --
    scheduler-user=ubuntu --scheduler-ssh-key=~/.ssh/aws-ec2.pem --scheduler-
    model-path=/home/ubuntu/models/testbed/resnet152_2exits_25epochs_0.pt --
    dataset=imagenet --dataset-root=/home/matthias/projects/addnn/datasets/
    hymenoptera_data --config=../masters-thesis/testbed/rpi4-cloud-resnet152.
    yaml --num-layers=56 --seed=42 dynamic --delay=0 --delay=300 --delay=0 --
    epoch-duration=60 --placement=optimal
```

List of Figures

List of Tables

List of Algorithms

Bibliography

[BMNZ14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[BSS+20] Enzo Baccarelli, Simone Scardapane, Michele Scarpiniti, Alireza Momenzadeh, and Aurelio Uncini. Optimized training and scalable implementation of conditional deep neural networks with early exits for fog-supported iot applications. *Information Sciences*, 521:107 – 143, 2020.

[Com] Python Community. psutil documentation on uss. URL: `https://psutil.readthedocs.io/en/latest/#psutil.Process.memory_full_info`.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[DMP+02] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[ETS20] Multi-access Edge Computing (MEC); Framework and Reference Architecture, December 2020.

[FDG+12] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[FFDL10] L. Fei-Fei, J. Deng, and K. Li. Imagenet: Constructing a large-scale image database. *Journal of Vision - J VISION*, 9:1037–1037, 08 2010.

[FMI83] Kunihiko Fukushima, Sei Miyake, and Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (5):826–834, 1983.

[HBG19] Ke-Jou Hsu, Ketan Bhardwaj, and Ada Gavrilovska. Couper: Dnn model slicing for visual analytics containers at the edge. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 179–194, New York, NY, USA, 2019. Association for Computing Machinery.

[HBWL19] C. Hu, W. Bao, D. Wang, and F. Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1423–1431, 2019.

[HCRK20] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim. Toward collaborative inferencing of deep neural networks on internet-of-things devices. *IEEE Internet of Things Journal*, 7(6):4950–4960, 2020.

[HDNQ17] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of Network and Computer Applications*, 98:27–42, 2017.

[HSC+19] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. Searching for mobilenetv3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1314–1324, 2019.

[HW62] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[HZC+17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[KAH+19] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[KH+09] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[KHD19] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. Shallow-deep networks: Understanding and mitigating network overthinking. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3301–3310. PMLR, 09–15 Jun 2019.

[KHG+17] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *SIGARCH Comput. Archit. News*, 45(1):615–629, April 2017.

[Kri12] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[LVA+20] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. Spinn: Synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.

[LZQ+19] Hao Li, Hong Zhang, Xiaojuan Qi, Ruigang Yang, and Gao Huang. Improved techniques for training adaptive deep networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[LZZC20] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2020.

[MDK+20] Francis McNamee, Schahram Dustadar, Peter Kilpatrick, Weisong Shi, Ivor Spence, and Blesson Varghese. A case for adaptive deep neural networks in edge computing, 2020.

[MG+11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[PGCB13] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.

[PSY+18] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and Sundaraja S Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Computing Surveys (CSUR)*, 51(5):1–36, 2018.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[Ros57] Frank Rosenblatt. *The perceptron — a perceiving and recognizing automaton, Project Para.* Cornell Aeronautical Laboratory, 1957.

[SAK07] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SD16] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[SHZ+19] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[SNS+17] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized iot service placement in the fog. *Serv. Oriented Comput. Appl.*, 11(4):427–443, December 2017.

[SSBU20] Simone Scardapane, Michele Scarpiniti, Enzo Baccarelli, and Aurelio Uncini. Why should we add early exits to neural networks? *Cognitive Computation*, 12(5):954–966, Jun 2020.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[TMK16] S. Teerapittayanon, B. McDanel, and H. T. Kung. BranchyNet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469, 2016.

[TMK17] S. Teerapittayanon, B. McDanel, and H. T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339, 2017.

[WCERG] University of Maine Weaver Computer Engineering Research Group. Linpack results for machines in the weaver computer engineering research group. URL: `http://web.eece.maine.edu/~vweaver/group/machines.html`.

[WCHD19] H. Wang, G. Cai, Z. Huang, and F. Dong. Adda: Adaptive distributed dnn inference acceleration in edge computing environment. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 438–445, 2019.

[WLC+17] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, and Joseph E. Gonzalez. IDK cascades: Fast deep learning by learning not to overthink. *CoRR*, abs/1706.00885, 2017.

[YFN+19] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[ZBG18] Z. Zhao, K. M. Barijough, and A. Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2348–2359, 2018.

[ZWLX21] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. *EdgeML: An AutoML Framework for Real-Time Deep Learning on the Edge*, page 133–144. Association for Computing Machinery, New York, NY, USA, 2021.