# Fast Edge Resource Scaling with Distributed DNN

Theodoros Giannakas[1], Dimitrios Tsilimantos[1], Apostolos Destounis[1], Thrasyvoulos Spyropoulos[2, 3]
[1] Paris Research Center, Huawei Technologies, France
[2] EURECOM, France, [3] Technical University of Crete, Greece

***Abstract*—Network slicing has been proposed as a paradigm for 5G+ networks. The operators slice physical resources from the edge all the way to the datacenter, and are responsible to micro-manage the allocation of these resources among tenants bound by predefined Service Level Agreements (SLAs). A key task, for which recent works have advocated the use of Deep Neural Networks (DNNs), is tracking the tenant demand and scaling its resources. Nevertheless, for the edge resources (e.g. RAN), a question arises on whether operators can: (a) scale them fast enough (often in the order of ms) and (b) afford to transmit huge amounts of data towards a remote cloud where such a DNN model might operate. We propose a Distributed DNN (DDNN) architecture for a class of such problems: a small subset of the DNN layers at the edge attempt to act as fast, standalone resource allocator; this is complemented by a mechanism to intelligently offload a percentage of (harder) decisions to additional DNN layers running at a remote cloud. To implement the offloading, we propose: (i) a Bayes-inspired method, using dropout during inference, to estimate the confidence in the local prediction; (ii) a learnable function which automatically classifies samples as "remote" (to be offloaded) or "local". Using the public Milano dataset, we investigate how such a DDNN should be trained and operated to address (a) and (b). In some cases, our offloading methods are near-optimal, resolving up to $50\%$ of decisions locally with little or no penalty on the allocation cost.**

***Index Terms*—Distributed inference, slicing, machine learning methodologies, resource provisioning.**

## I. Introduction

The advent of 5G+/6G networks has been characterized by a number of radical architectural transformations. Virtualization and slicing of communication, computation and infrastructure resources allow operators to co-host multiple services and tenants, with a large variety of performance requirements and Service Level Agreements (SLAs). What is more, 5G networks and beyond (5G+) will be characterized by increased programmability through the use of composable Virtual Network Functions (VNFs), executable at various network locations (edge/core/fog). This creates a great opportunity for an algorithmic optimization approach, towards (re-)designing modern cellular networks to cope with the daunting complexity of multi-service, multi-domain, multi-SLA emerging environments.

Thrasyvoulos Spyropoulos has been supported in part by the Hellenic Foundation for Research & Innovation (HFRI) project 8017: "AI4RecNets: Artificial Intelligence (AI) Driven Co-design of Recommendation and Networking Algorithms" and in part by the H2020 MonB5G Project (grant agreement no. 871780). This research was conducted while Theodoros Giannakas and Apostolos Destounis were with Huawei Technologies, Paris Research Center, France.

Traditionally, the various network components that affect the overall performance of a service (e.g., MAC scheduling, transport, core computation resources, etc.) have been optimized using proprietary algorithms, heuristics and simplified models to facilitate tractability. The literature abounds with such problem formulations that are based on several modeling assumptions, like knowledge of key inputs and/or stationarity (among others), which are often not satisfied in practice. As a result, flexible *model-free* algorithms based on modern Machine Learning (ML) methods have received significant attention as an alternative way to tackle wireless network optimization problems arising in 5G+ networks [1]. Such methods can learn to optimize various networks tasks, operating directly on offline (e.g., supervised learning) or online (e.g., online convex optimization) training data, without the need for a priori limiting modeling assumptions.

An important 5G task that has been recently addressed with data-driven methods is that of traffic prediction and slice resource allocation with Deep Neural Networks (DNN). In [1]–[4], the authors use a DNN to predict the Base Station (BS) traffic, based on past traffic demand samples. In particular, DeepCog [3], uses a DNN placed at a datacenter, which directly predicts the amount of resources, needed by a slice that is associated with BSs, in order to avoid both *underprovision* and *overprovision* of the user-generated demand—the former results in costly SLA violations, while the latter implies the wasting of valuable resources that another slice/tenant could have potentially used. As a consequence, said regression-like problems can be tackled by training popular DNN architectures, using an objective with appropriately tuned under- and over-provision terms. The main novelty in [3] (which we will also use) is the use of a 3D-ConvNet for resource allocation. The intuition for their choice is that time-series corresponding to different BS demands can be highly correlated (e.g., traffic in BSs near tram, metro, and other "commute" spots); and thus, a 3D-CNN, receiving as input an appropriately pre-processed image-like representation of past demands, will be able to exploit these correlations maximally.

While the previous approach is promising, the standard assumption of a *centralized* implementation of the DNN architecture faces the following two challenges, when used to control key 5G+ network functions: *First,* unlike the use of DNNs for some application-layer tasks (e.g., image classification on a mobile phone device) that can be "lazily" offloaded to a central computational cloud, the use of DNNs for controlling 5G edge resources (e.g., allocation of RAN resource blocks among tenants, CPU allocation

for CRAN processing) requires significantly lower latency; sending all required data to a central DNN, making the decision there, then sending back the actuation message to the desired edge components (resources for an edge VNF), can violate these requirements. *Second,* constantly sending raw monitored data, over possibly already congested links, towards a DNN architecture lying deep in the core network has a prohibitive network footprint and is a *costly* operation. Hence, we are interested in how such DNN architectures could be appropriately *implemented in a distributed fashion* towards resolving both of the above concerns, yet without compromising the performance advantages of the DNN.

In the context of image classification, the seminal work [5] introduced the concept of Distributed DNN (DDNN). The key idea behind DDNNs is to split the layers of a DNN between different (in the geographical sense) locations, where predictions can be taken at each location: e.g., *locally* (edge) if the latency requirement for a decision/prediction is stringent or the network links to the core are congested; or *remotely* if additional accuracy is needed. To achieve both tasks well, one needs to *jointly* train both the local and remote layers. For a first view of the architecture we will be discussing throughout this paper, please see Fig. 1(a).

To that end, the main goal of this paper is to propose, train and study a *distributed* architecture for a data-driven edge resource allocation problem that generalizes the one considered in recent state-of-the-art work [3], [4]. Our main contributions are summarized below:

**(C.1) DDNNs for regression problems.** We propose a *Distributed DNN* architecture that can be applied to any regression task, thereby extending the seminal work [5] that focused on image classification tasks. We apply this idea to a resource allocation problem that arises in slicing for 5G.

**(C.2) Edge offloading mechanism for regression tasks.** The key component of our DDNN is the offloading mechanism, placed at the edge, which decides *online* whether to perform the inference instantly at the edge or offload to the remote core cloud for additional processing. This mechanism aims to fulfill the stringent inference time requirements by not compromising too much the accuracy of the regression task. To address this, we propose the following two methods:

***—Uncertainty Rule*:** We estimate the uncertainty of the local exit by making multiple forward-passes, using random dropout, of the same input sample during inference. To generate the offloading decision, the sample uncertainty is compared to an *operator-defined threshold*, which is selected based on the inference time requirements. This methodology first appeared in our preliminary work [6].

***—Optimized Rule*:** After training the DDNN layers, we do a ranking of the training samples based on the resource allocation costs of the local and remote DNN exits. This key *meta*-data allows us to *cast the optimization of the offloading mechanism as supervised learning problem*. To solve this, we train a classifier which can directly differentiate easy from hard decisions, based on the features of the sample. It is worth mentioning that, although we apply this approach on a regression task, the idea is more generally applicable. For example, it could be also used on DDNNs answering image classification queries, such as the seminal DDNN [5][1]. Our simple yet effective methodology, essentially, bypasses the need of uncertainty estimation at the local exit.

**(C.3) Extensive evaluation of DDNNs on real data.** We demonstrate that our DDNN architecture (DDNN layers + offloading) is able to resolve a large percentage of its decisions using the local lightweight model with little (resource allocation) cost increase. In particular, the *Uncertainty Rule* can operate at a regime where 40% of the decisions are taken locally with a penalty of $\approx 3\%$ of cost. Moreover, our *Optimized Rule* almost always outperforms the *Uncertainty Rule*, and often performs close to an offline optimal *Oracle*, which is a baseline we defined. Importantly, we show by means of simulation that the *Optimized Rule* policy manages to distinguish correctly the "hard" samples for which the more powerful remote layers can offer a big resource allocation cost reduction.

Below, we list the new technical contributions of this work which are not present in our conference paper [6].

- The *Optimized Rule* for offloading, which is a learnable ML model, optimized using principled gradient descent.
- The definition of the *Oracle* baseline, which serves as an upper bound for the DDNN performance.
- The significantly extended evaluation section, that includes more experiments and baselines.

**Roadmap.** Section II setups the problem and the objectives of the network infrastructure provider (slice host). Section III introduces our DDNN architecture and the variables involved; explains DDNN inference (how it takes the resource allocation decisions at real time) and training; and finally showcases the targets of the early (local) exit. Importantly, in Section IV we present our two proposed offloading methodologies, specifically tailored to address regression-like tasks. Section V presents our results and insights on DDNN system performance. Section VI and VII discuss related and future work respectively; and Section VIII concludes the paper.

## II. Resource Allocation with DNNs

In this section, we revisit how a (centralized) DNN can be used for accurate and "safe" slice resource allocation.

### A. Data-driven Edge Resource Allocation

We consider a provider whose infrastructure is a set of interconnected servers, from which some reside at the edge of the network and others at the core, see [7]. The provider hosts a set of slices, where each slice consists of VNFs. Typically, there are two types of VNFs: i) the ones executed at the edge (e.g., baseband unit or edge caching) and others at the core, such as user/control plane [7]. At each timeslot, a VNF is associated to a demand value that corresponds to the amount of computational resources (e.g., CPU, memory) it needs from the provider. Said resource demand is a result of the traffic generated by the users. *We are interested in allocating resources for the edge VNFs.* The set of edge VNFs is denoted by $\mathcal{M} = \{1, \dots, M\}$, and the true resource demand of VNF

[1]Uncertainty is quantified via entropy of the DNN output in [5].

$m$ at $t$ as $d_{t,m}$. The VNFs we consider do not necessarily belong to the same slice; they could be viewed as generic network elements for which we allocate resources.

By monitoring the network activity, we keep track of the $N$ past traffic samples of each VNF; we denote the $n$-th past demand of the $m$-th VNF as $d_{t-n,m}$. Our goal is to use past demands to help us allocate (schedule) resources for all the VNFs in $\mathcal{M}$ (e.g. a real number indicating some kind of resource for each), before the true demand is revealed. The provider would like our allocated resources to satisfy an SLA related to the unknown true demand $d_{t,m}$ of each VNF. The resources are allocated by a DNN, denoted by $\mathbf{F}$, and parameterized by $\boldsymbol{\theta}$. Below, we define the input-output pair and the loss of $\mathbf{F}$.

***Input*:** At time $t$, the DNN receives $\mathbf{D}_t = (d_{t-n,m}) \in \mathbb{R}^{M\times N}$, where $m \in \mathcal{M}$ and $n = 1, \ldots, N$. This is a snapshot of past traffic of all $M$ VNFs with a window of size $N$.

***Output*:** We denote with $\mathbf{y}_t \in \mathbb{R}^M$ the DNN output; $y_{t,m}$ represents the resources allocated for VNF $m \in \mathcal{M}$.

***Operator cost*:** At every timestep $t$, the following three events take place sequentially. First, the amount of resources that will be allocated is computed as $\mathbf{y}_t = \mathbf{F}(\mathbf{D}_t; \boldsymbol{\theta})$, where $\mathbf{F} : \mathbb{R}^{M\times N} \to \mathbb{R}^M$ is any "prediction" architecture (to be elaborated shortly). Then, the users' activity generates the true resource demand $\mathbf{d}_t \in \mathbb{R}^M$. And finally, the network operator pays a scalar cost $f(\mathbf{y}_t, \mathbf{d}_t)$.

**Remark:** It is worth stressing that with our variables $\mathbf{D}, \mathbf{d}, \mathbf{y}$ defined, the subscript $t$, as we will see next, is neither relevant for the computation of loss, nor for the DNN execution, since all these equations *involve variables of the same timestep*. Hence, to make the notation lighter, we will drop subscript $t$ in the remainder of the paper. We will only use $k$ to indicate *available data* in later sections, when we want to sum them across a dataset.

### B. Optimization Objective

Typically, the target in time-series problems is to give prediction $y_m$ for some demand $\mathbf{d}_m$, before the latter is revealed. The objective is evaluated as:

$$f(\mathbf{y}, \mathbf{d}) = \sum_{m\in\mathcal{M}} (y_m - d_m)^2 . \tag{1}$$

This objective is appropriate when we consider, for example, analytics applications. On the other hand, when allocating resources for slices, under- and over-provision costs are by nature *asymmetric*, as they capture different penalties. The former ($y_m < d_m$) is often governed by SLAs of the operator with the slice tenant, while the latter ($y_m > d_m$) stems from unused resources that, for example, the operator could have allocated to another VNF. Typically, the operator aims to "play it safe" striving to avoid underprovisions at the cost of wasting few resources (i.e., allocating slightly more than the true demand).

For every VNF $m$, we denote the following two events:

- "u": $y_m < d_m$ for underprovision;
- "o": $y_m \geq d_m$ for overprovision.

TABLE I: Main Notation

| | |
|---|---|
| $\mathbf{d}$ | VNFs demand vector, in $\mathbb{R}^M$ |
| $\mathbf{y}$ | VNFs resource allocation, in $\mathbb{R}^M$ |
| $\mathbf{D}$ | Past traffic snapshot, in $\mathbb{R}^{M\times N}$ |
| $\mathbf{F}_0$ | First NN block at the edge |
| $\mathbf{F}_1$ | Early exit NN block at the edge |
| $\mathbf{F}_2$ | NN block at the cloud |
| $\mathbf{z}$ | Latent variable, output of $\mathbf{F}_0$ |
| $\mathbf{y}_L$ | Res. allocation of local exit, output of $\mathbf{F}_1$, in $\mathbb{R}^M$ |
| $\mathbf{y}_R$ | Res. allocation of remote exit, output of $\mathbf{F}_2$, in $\mathbb{R}^M$ |
| $f(\mathbf{y}, \mathbf{d})$ | Cost of allocation $\mathbf{y}$, when demand is $\mathbf{d}$ |
| $C_L$ | Cost of local exit |
| $C_R$ | Cost of remote exit |
| $b^*$ | Critical benefit |

We denote with $I_{u,m} = \{1 \text{ if } y_m < d_m, 0 \text{ otherwise}\}$ the underprovision indicator, and by $I_{o,m} = 1 - I_{u,m}$ the overprovision one. The cost incurred by allocation $\mathbf{y}$ can be then expressed as follows:

$$f(\mathbf{y}, \mathbf{d}) = \sum_{m\in\mathcal{M}} \Big( I_{u,m} \cdot g_u(d_m - y_m) + I_{o,m} \cdot g_o(y_m - d_m) \Big), \tag{2}$$

where $g_{u/o}(.) : \mathbb{R} \to \mathbb{R}$ are generic non-decreasing functions. An example of such function, fitting well the needs of the slicing resource allocation problem, was defined in [3] as

$$f(\mathbf{y}, \mathbf{d}) = \sum_{m\in\mathcal{M}} \Big( c_u \cdot I_{u,m} + c_o \cdot I_{o,m} \cdot (y_m - d_m) \Big). \tag{3}$$

According to the above, a constant $c_u$ penalty is paid for any SLA violation (left term), and the overprovisioning cost increases linearly (right term) with $c_o$.

For a quick reference, we gather all the notation in Table I.

### C. Discussion on the Chain Embedding Problem

The Chain Embedding Problem (CEP), whose most variants have been shown to be NP-hard [8], [9], has attracted a lot of attention from the networking research community [10]–[15] during the last decade. Its decision variables can be decomposed as follows: (i) the placement of the VNFs and virtual links onto the physical infrastructure, (ii) the routing of the traffic, and (iii) the resources that need to be allocated (e.g., CPU or memory), also called scaling, for the processing of the continuously fluctuating traffic demand. This work focuses on (iii), given the solutions of (i) and (ii).

Furthermore, notice that CEP's decomposition into smaller, more manageable, problems can be based on the timescale that these subproblems are solved. For example, the placement is solved in a longer timescale (see [4], [16]), due to the fact that VNF migration is a costly operation that should not occur frequently. In contrast, the scaling, whose main target is to react on the fluctuating demands the VNF experiences, is solved more frequently. Evidently, the scaling and the placement/routing are interconnected problems whose solutions would benefit if they were solved by the same entity. To give a simplified example, a placement policy that assigns all VNFs into a single server, such that it can barely accommodate its assigned load, leaves the scaler little to

no room to increase/decrease its resources according to the fluctuating demand. Hence, in this example the placement policy ignores the scaler, essentially rendering it ineffective.

With that being said, technically our scaling mechanism does not require that the same entity controls all policies, since the physical server's resources are always capped by some budget. (Trivially our DNN scaling mechanism could give the maximum resources it has available). However, for performance reasons, having a centralized operator controlling all policies would be beneficial.

## III. Resource Allocation with Early-Exit DDNNs

In this section, we discuss how an early exit turns *a DNN into a DDNN* that spans two domains, edge and a remote cloud. To this end, we first describe DDNN inference; then explain how such a DDNN is trained to provide meaningful allocations both in its early (edge) local exit, as well as its remote one; finally, we discuss the role of the offloading mechanism in reducing the DDNN inference latency.

### A. Inference

We give a brief overview of the DDNN architecture during the inference phase, see Fig. 1(a). At the edge (left part of the figure), $N$ past traffic samples of edge VNFs are collected, which create the data sample $\mathbf{d}$. The sample $\mathbf{D}$ is forward-passed through the DNN block $\mathbf{F}_0$ which produces latent variable $\mathbf{z}$. Then $\mathbf{z}$ is sent to either:

- The early-exit layers $\mathbf{F}_1$, and the resource allocation is $\mathbf{y}_L$ (we have a local decision); or,
- The remote cloud layers $\mathbf{F}_2$, and the allocation is $\mathbf{y}_R$ (we have a remote decision).

In the example DDNN of Fig. 1(a), $\mathbf{z}$ is used to trigger the offloading decision. If the offloading algorithm decides that a quick resource decision must be taken, then resources are based on $\mathbf{y}_L$ (green path), otherwise on $\mathbf{y}_R$ (red path). Our main contribution is the offloading mechanism. We discuss in detail the challenges and our proposed offloading methodologies in Section IV.

We based our DNN models on 3D-Convolutional NNs (3D-CNN), as they have exhibited good performance on a similar time-series task [3]. However, note that here we need to "break" the 3D-CNN in two parts. At the edge, we have a 3D-CNN $\mathbf{F}_0(\cdot;\boldsymbol{\theta}_0)$, followed by Fully Connected (FC) layers $\mathbf{F}_1(\cdot;\boldsymbol{\theta}_1)$. At the remote cloud, there is another 3D-CNN with FC layers, denoted as $\mathbf{F}_2(\cdot;\boldsymbol{\theta}_2)$. For reference of the above, please see Fig. 1. The variables involved can be described by the following feed-forward equations:

$$\text{Latent variable:} \quad \mathbf{z} = \mathbf{F}_0(\mathbf{D};\boldsymbol{\theta}_0), \tag{4}$$

$$\text{Local exit allocation:} \quad \mathbf{y}_L = \mathbf{F}_1(\mathbf{z};\boldsymbol{\theta}_1), \tag{5}$$

$$\text{Remote exit allocation:} \quad \mathbf{y}_R = \mathbf{F}_2(\mathbf{z};\boldsymbol{\theta}_2). \tag{6}$$

From Fig. 1(a), during *inference*, the offloading mechanism, using only local information (in this example $\mathbf{z}$), decides whether local exit allocation $\mathbf{y}_L$ should be accepted (green path) or not (red path). The operator then pays one of the following two:

$$\text{Local exit cost:} \quad C_L = f(\mathbf{y}_L, \mathbf{d}), \tag{7}$$

$$\text{Remote exit cost:} \quad C_R = f(\mathbf{y}_R, \mathbf{d}). \tag{8}$$

Notice that (7) and (8) use the same $f(\cdot)$ and the same true demand $\mathbf{d}$; but they are evaluated on the corresponding $\mathbf{y}_L$ and $\mathbf{y}_R$. Next, we discuss the offline training of $\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2$.

### B. Training

The blocks $\mathbf{F}_0, \mathbf{F}_1$ and $\mathbf{F}_2$ must be *jointly* trained to achieve an intricate tradeoff: (i) $\mathbf{F}_0, \mathbf{F}_1$ must be powerful enough to correctly resolve some decisions locally; and (ii) $\mathbf{F}_0$ must still act as high level feature extractors, so that $\mathbf{F}_2$ can offer added value for "hard" decisions. In DDNN architectures [5], this is commonly achieved by a weighted objective of (7) and (8). We train these blocks (layers) in a centralized offline manner[2]. In Fig. 1(b), the latent variable $\mathbf{z}$ is sent to $\mathbf{F}_1$, which computes $\mathbf{y}_L \in \mathbb{R}^M$, and to $\mathbf{F}_2$, which computes $\mathbf{y}_R \in \mathbb{R}^M$. From the training samples, we pick a batch and perform standard backpropagation to update $\boldsymbol{\theta}_{i\in\{0,1,2\}}$. The joint cost $C_J$ of a sample is:

$$C_J = w \underbrace{f(\mathbf{y}_L, \mathbf{d}; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)}_{\text{local exit loss}} + (1-w) \underbrace{f(\mathbf{y}_R, \mathbf{d}; \boldsymbol{\theta}_0, \boldsymbol{\theta}_2)}_{\text{remote exit loss}}. \tag{9}$$

Observe that $C_J$ depends on $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2$; and the weight $w \in [0,1]$ controls the importance of one exit at the cost of the other. When $w = 0$, only $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_2$ are updated since they affect $C_J$, but $\boldsymbol{\theta}_1$ is not; as a result, only $\mathbf{y}_R$ is trained. In the opposite case, if $w = 1$, only $\mathbf{y}_L$ is trained.

**Remark:** We stress the following two facts regarding training with the joint-exit objective (9):

- Training a *DDNN* with $w = 0$ is equivalent to a standard centralized DNN with a single final exit, i.e., $\mathbf{y}_R$.
- Training a *DDNN* with $w > 0$ yields improved test accuracy for $\mathbf{y}_R$ compared to using $w = 0$, i.e., the centralized case.

The latter, perhaps surprising, fact was first studied in [17], where the authors observed that the joint-exit training acts as a source of regularization to $\mathbf{y}_R$. Intuitively, by optimizing also $\mathbf{y}_L$, we make it harder for $\mathbf{y}_R$ to overfit the data. This is discussed in the results section, where the $\mathbf{y}_R$ of a DDNN (trained with $w > 0$) is shown to perform better than the corresponding final exit of a standard centralized DNN, such as DeepCog [3].

### C. Exploiting the Local Early Exit

The goal of DDNNs is to offer flexibility by splitting the layers across two or more domains, giving rise to the so-called edge-cloud continuum. A first advantage of an early exit is the *adaptive and controllable* speed of inference. This is relevant to slicing, given the stringent latency requirements of 5G+/6G applications. Moreover, early-exit inference

[2]Distributed training is an important topic, but is orthogonal to our work which mostly targets the inference phase of the DDNN.

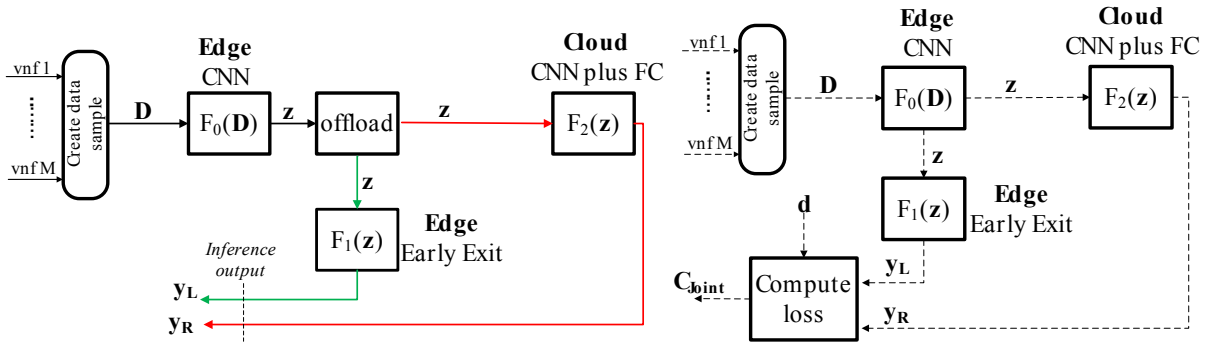



Fig. 1: Signal flow for Distributed-DNN. (a) latent variable $\mathbf{z}$ is used by the offloading mechanism, which then triggers either local (green) or remote (red) response. (b) Dashed lines indicate that training is carried out offline, during which, true demand $\mathbf{d}$ and both exits $\mathbf{y}_L, \mathbf{y}_R$ are needed.

reduces the channel communication costs, which can be quantified by considering the amount of data transmitted through the channel. In our application, since a network can host many slices with many VNFs [18], a centralized approach, transmitting the features $\mathbf{d}$ to the remote cloud, could contribute in congesting the link between the two domains. On the other hand, DDNNs are more scalable since they transmit latent variables, such as $\mathbf{z}$, which only depend on the DNN architecture.

Our focus in this paper is maintaining a low latency, while minimizing the resource allocation cost. We assume that network channel conditions are fixed; hence the Round Trip Time (RTT), defined as the expected total time needed for $\mathbf{z}$ to reach the remote cloud plus the time needed for $\mathbf{y}_R$ to reach back the edge, is considered fixed. This assumption simplifies the problem, as an increase in the use of local exit immediately suggests a decrease in latency[3]. For example, early exiting $50\%$ of the time implies avoiding $50\%$ of RTTs.

**Definition 1** (Offloading $\mathcal{G}$)**.** *An online mechanism, placed at the edge, which irrevocably decides if the inference will be done via the early exit ($\mathbf{y}_{L,t}$) or via the remote one ($\mathbf{y}_{R,t}$). The available information for $\mathcal{G}$ is only edge data, and at time $t$ its output is $l_t \in \{0,1\}$, with $l_t = 1$ indicating to use the local exit, and $l_t = 0$ to use the remote cloud. In a horizon of $T$ samples, we denote the percentage of samples resolved locally $L = \frac{1}{T}\sum_{t=1}^{T} l_t$.*

**Target (informally).** The operator wishes that $\mathcal{G}$ takes offloading decisions which minimize the provisioning cost (objective), and that keep the number of locally resolved samples $L \geq L_0$, with $L_0 \in [0,1]$ (constraint); the later is used as a proxy for latency. In other words, the goal of $\mathcal{G}$ is to filter online the samples for which $C_L - C_R >> 0$ (using one or more variables out of $\{\mathbf{D}, \mathbf{z}, \mathbf{y}_L\}$), and to send them to the remote cloud. The mechanism $\mathcal{G}$ has to solve this online with access to limited information, which renders its target very challenging.

To address the offloading problem, a typical heuristic is to use the entropy of early exit $\mathbf{y}_L$ [5]. The operator controls $L$ implicitly via a tunable entropy threshold. Setting this threshold low means that we accept only very confident answers from the early exit, and hence $L$ will be low. In the next section, we will present two ways of modeling $\mathcal{G}$. We will first explain why the vanilla entropy approach fails in regression tasks and modify it accordingly. Then, we will present a second method to design $\mathcal{G}$, which is data-driven and more grounded to basic optimization principles.

[3]This assumption also allows us to handle samples in a stateless manner. Otherwise, if conditions are changing, actions taken now can affect future rewards. We discuss more on this more in the future work section.

## IV. How to Offload Hard Decisions Remotely

In this section, we are assuming that layers $\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2$ are optimized for the resource allocation task, and we focus on the design of offloading mechanism $\mathcal{G}$ to achieve its targets. First, inspired by the seminal work [5], we present a novel method based on local exit uncertainty. Our main differentiation with respect to [5] is that our DDNN predicts numerical values that correspond to some resource, and not classes, which makes it more challenging. Our preliminary work [6] was the first that attempted to face this problem (Section IV-A). Then, we propose a more principled design of the offloading mechanism by casting its optimization as a supervised learning problem (Section IV-B).

### A. Uncertainty Rule

In this approach, the local information that is used to produce the offloading decision, $l$, is $\mathbf{y}_L$ [6]. Uncertainty $U$ of local layers on input $\mathbf{D}$ should be defined such that low $U$ means $\mathbf{y}_L$ is a confident output; while high $U$ will urge us to forward $\mathbf{z}$ to $\mathbf{F}_2$ and use $\mathbf{y}_R$ as our decision instead. Such a rule came quite naturally in the context of classification tasks [5], using entropy. However, recall that entropy is measured over discrete support set (labels/classes), unlike our case, where the prediction is a numerical value.

To address this issue, our starting point is recent results in [19] and [20], where the main goal is the study of uncertainty, regardless of the task resolved by the ML model (classification or regression). The idea is that forward-passing the same sample, using dropout, multiple times causes a perturbation on $\mathbf{y}_L$, which can be used to estimate the model's uncertainty. In the context of resource allocation

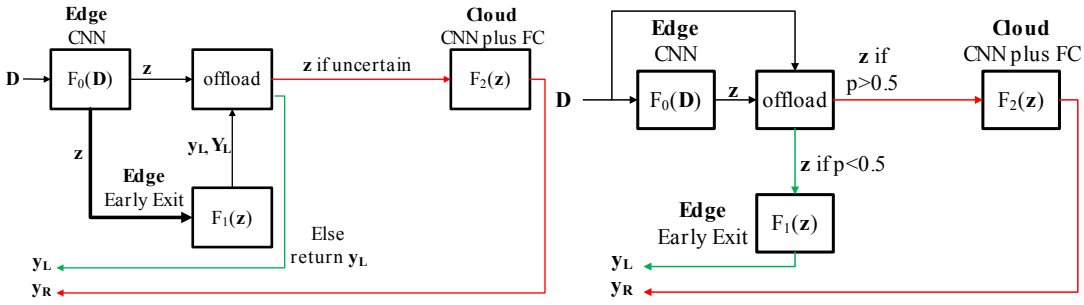



Fig. 2: DDNN with online offloading rule (the "offload" block resides at the edge). (a) The thicker $\mathbf{F}_0$ to $\mathbf{F}_1$ indicates the $B$ forward passes needed in order to generate $\mathbf{Y}_L$, with which we estimate the uncertainty $U$. (b) The data point $\mathbf{D}$ is fed directly to trainable model $\mathbf{G}$ inside the offload mechanism.

with centralized DNNs, this technique was also used in [4]; however, its purpose there was safety. Effectively, when uncertainty was low, the DNN returned the average of the output allocations. Otherwise, it returned one of the conservative output allocations, thereby avoiding underprovision events.

Unlike the above works that focus on DNNs, our main idea is to leverage this technique as a *proxy to discover easy and hard samples*. The idea is to evaluate the uncertainty $U$ of the local exit on $\mathbf{D}$; and if $U$ does *not* exceed some predefined threshold $u$, we accept $\mathbf{y}_L$ as our resource allocation; otherwise we reject the local exit and use $\mathbf{y}_R$. Our approach is carried out online and needs no offline optimization steps. In more detail, a DDNN equipped with *Uncertainty Rule* offloading is depicted in Fig. 2(a). At time $t$, first we forward-pass $\mathbf{D}$ via $\mathbf{F}_0$ to get $\mathbf{z}$, which we send to "offload"; moreover, the latent variable $\mathbf{z}$ is sent to $\mathbf{F}_1$ so that $\mathbf{y}_L$ is computed and also sent to "offload". Second step is to compute the offloading decision, which is done as follows. The input $\mathbf{D}$ is forward-passed $B$ times through $\mathbf{F}_0$ and $\mathbf{F}_1$, by applying random *dropout* of the neurons at those blocks. This creates $B$ random realizations of local exit $\mathbf{y}_L$; we denote the $b$-th one as $\mathbf{y}_L^b$. Further, we denote with $\mathbf{Y}_L \in R^{B\times M}$ the matrix that stacks those $B$ realizations. (We remind that every row of that matrix is of size $M$, which is the number of VNFs we allocate resources for.) Then, uncertainty, $U$, is measured using the maximum variance across the $B$ random realizations. In particular, we compute the variance of every column of $\mathbf{Y}_L$, which results in $M$ variances, one for each VNF. Formally, $U$ is given below:

$$U = \max_{m\in\mathcal{M}}\{\mathrm{Var}(y_L^m)\}. \tag{10}$$

Intuitively, this can be understood as a proxy for the worst-case VNF prediction. Finally, we compare $U$ to a predefined threshold $u$, which is selected by the physical network operator. If $U > u$, we offload to the remote cloud, otherwise we use the unperturbed version of the local exit $\mathbf{y}_L$ which has been stored at the "offload" block. Notice that the threshold $u$ acts as a knob: $u = 0$ means that even very small uncertainty $U$ is not tolerated and the operator will trust the remote DNN to make the resource allocation; while a high value of $u$ implies that the operator might accept local exit decisions that have uncertainty, because the system could, at the moment, have, e.g., very stringent latency requirements.

Our intuition for using this approach is the existence of some correlation between the variance of $\mathbf{y}_L$ and the difference $C_L - C_R$; however, *there is no theoretical guarantee* that $\mathbf{y}_R$ will be "better" (i.e. have lower resource provision cost) if the $\mathbf{y}_L$ confidence is low. In fact, this is not true, in general, even for the original entropy-based metric, for classification problems like the one in [5]. We remind, therefore, that the target of $\mathcal{G}$ is to discover the $1-L_0$ percentage of samples for which the cloud benefit is high and delegate them remotely, and to this end, we propose an alternative, optimization-based methodology in the following subsection.

### B. Optimized Rule

We propose a second offloading methodology that leverages the training data we have access to, which are typically ignored by uncertainty-based methods [5], [6]. The key idea stems from taking the target defined in Section III-C and translating it into an optimization problem.

**Formulation.** The offloading task needs to be addressed by an online algorithm. In the literature of online algorithms (see e.g., Ski Rental [21]), a typical practice is to formalize the offline full horizon optimization problem, in order to define the optimal benchmark. The key step is to observe that, since $\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2$ have already been optimized, we know the local and remote exit allocations, and therefore their associated costs, $C_L$ and $C_R$, for all samples in the *training data*. Below, we use subscript $k = 1, \ldots, K$ to indicate training samples we have access to. The offline optimization problem addressed by $\mathcal{G}$ then becomes as follows.

**Optimization Problem (Offline).**

$$\underset{l_k}{\textit{minimize}} \quad \frac{1}{K}\sum_{k=1}^{K}\Big(C_{L,k}\cdot l_k + C_{R,k}\cdot(1-l_k)\Big), \tag{11a}$$

$$\textit{subject to} \quad \frac{1}{K}\sum_{k=1}^{K} l_k \geq L_0. \tag{11b}$$

Problem (11) has a tradeoff: the higher $L_0$ (i.e., the more the early exit samples), the less we use the remote exit, which will typically have better (lower) costs. We solve it with Algorithm 1, to which we provide as inputs $\{C_{L,k}\}_{k=1,..,K}$, $\{C_{R,k}\}_{k=1,..,K}$ and the constraint requirement $L_0$.

**Algorithm 1**

Input: $L_0, \{C_{L,k}\}_{k=1,..,K}, \{C_{R,k}\}_{k=1,..,K}$
Compute $\mathbf{b}$: $b_k \leftarrow C_{L,k} - C_{R,k}$ ▷ $\mathbf{b} \in \mathbb{R}^K$
$\mathbf{b}' \leftarrow \text{Sort}(\mathbf{b})$ ▷ Increasing order
$b^\star \leftarrow \text{percentile}(L_0, \mathbf{b}')$
return $b^\star$

**Oracle-based labeling.** The critical benefit $b^\star$ is the value for which $L_0$ percent of the benefit values of the training data are lower than $b^\star$. The idea is to use $b^\star$ to create *new labels*, in order to cast the optimization of $\mathcal{G}$ as supervised learning. In particular, samples $\mathbf{D}$ for which $b_k < b^\star$ are labeled as "easy" (i.e., their benefit when using the remote cloud is not too big, and thus we can resolve locally) and the rest as "hard". Below, we explain first how to train an ML (supervised) model that mimics the behavior of the offline optimal algorithm, and then how the said model operates during inference.

**Training.** For our training data, we compute the critical benefit $b^*$ using Algorithm 1. Then we label as "easy" the $\mathbf{D}$ for which $C_L - C_R < b^*$, and the rest as "hard". We now have access to a new *meta*-dataset, whose input/output pairs are well defined. Using this new data, we can train a classification model that will serve as our offloading mechanism. Notice that if the latency requirement $L_0$ changes, the labels also change, and we thus need to train a new *Optimized Rule*.

**Inference time.** In Fig. 2(b), we show the execution steps of a DDNN, equipped with the *Optimized Rule*, at inference time. First, $\mathbf{D}$ is used as input in $\mathbf{F}_0$ and in the offload mechanism, where the already trained ML model lies. Block $\mathbf{F}_0$ produces $\mathbf{z}$ which is also sent to offloading mechanism. In parallel, the *Optimized Rule* computes the offloading decision, $p$. If $p < 1/2$, latent variable $\mathbf{z}$ is forwarded to $\mathbf{F}_1$, otherwise to $\mathbf{F}_2$, and the corresponding resource allocation will be used (green or red path).

The *Optimized Rule* has the following advantages at inference time compared to the *Uncertainty Rule*:

1) Since it does not need an estimate of uncertainty (hence no need for artificial randomness), it requires a single forward-pass of the edge DNN blocks.
2) When it offloads to the cloud, it bypasses the execution of $\mathbf{F}_1$, potentially achieving important energy savings.

## V. Performance Evaluation

### A. Preliminaries

**Dataset and preprocessing.** We validated the performance of our architecture with the public Milan dataset [22], widely used in similar studies [23]. Specifically, we use traffic (time-series signals) seen by BSs (measured in MBs) to simulate traffic patterns of VNFs.

To exploit the full potential of 3D-CNNs on time-series signals, we apply a preprocessing procedure that was first used in [3]; we restate it here for completeness. At a high level, this procedure is responsible for taking an input $\mathbf{D} \in R^{M \times N}$, i.e., a 2D matrix, and transforming it to a new $\mathbf{D}' \in R^{M_D \times M_H \times N}$, i.e., a 3D tensor that will be used as input to the 3D CNN. Notice that the transformed $\mathbf{D}'$ is now a rectangular prism of size $M_D \times M_H$ and length $N$. Essentially, this procedure aims at placing the BSs in matrix coordinates, such that highly correlated BSs end up neighbors, as is the case with pixels in images. Note that this procedure applies to both centralized and distributed inference, as its role is to transform the input $\mathbf{D}$.

We begin by normalizing each time series with respect to their min and max values. The placement is then found in two steps. First, we compute the correlation pairs, $q_{ij}$, between BSs $i$ and $j$ using shape based distance [24]. Then, let $\mathbf{p} \in \mathbb{R}^{2 \times M}$ be our optimization variable, with $p_m \in \mathbb{R}^2$ indicating the location of BS $m$ in 2D. The locations $p_m$, that maximally respect the BSs correlations $q_{ij}$, are found by solving:

$$\underset{\mathbf{p}=[p_1,\ldots,p_M]}{\text{minimize}} \sum_{i \neq j} (||p_i - p_j|| - q_{ij})^2. \tag{12}$$

Second, we map every BS to a matrix coordinate. To do this, we define costs $c_{ij} = ||p_i - y_j||_2^2$, where $y_j$ is the $j$-th point of the *regular* 2D grid. We now need to find the assignment of BSs to points in the 2D regular grid that minimize the total cost. Let our optimization variable be $X \in \{0,1\}^{M \times M}$ with $x_{ij} = 1$ meaning that BS $i$ is assigned to point $j$ of the 2D regular grid. Using [25] we solve:

$$\underset{X}{\text{minimize}} \sum_{i=1}^{M} \sum_{j=1}^{M} c_{ij} x_{ij}$$

under constraints $\sum_{i=1}^{M} x_{ij} = 1$, and $\sum_{j=1}^{M} x_{ij} = 1,\ \forall\ (i,j)$, which gives the desired assignment.

For our experiments, we randomly pick $M = 49$ BSs and do the above preprocessing step. The input sample we feed to the edge CNN is a "traffic box" of size $7 \times 7$ and length $N$ (window of past samples)[4]. We will use $N = 6$ and $N = 25$ (these values correspond to 1h, and to 4h10' respectively).

**DDNN layers.** For a fair comparison against [3], we use an architecture based on 3D-CNNs and FC layers.

- **Edge**: $\mathbf{F}_0$ is a CNN with 32 filters of kernel size 3; and $\mathbf{F}_1$ a FC layer from $64 \rightarrow 49$.
- **Cloud**: $\mathbf{F}_2$ is a CNN with 16 filters of kernel size 5, and two FC layers, one $128 \rightarrow 64$ and a $64 \rightarrow 49$.

where in both cases, $M = 49$ is the number of VNFs for which we give predictions.

Finally, the *Optimized Rule's* ML model is represented by a 3D-CNN (32 filters) and a FC layer, $64 \rightarrow 1$. It is important to stress that other classifiers (DNN-based or not) could be trained to perform the offloading task. Nevertheless, the above design seems like a reasonable choice since $\mathbf{D}$ is already arranged as a 3D traffic box.

**Objective function.** We remind the reader that our goal is to provision resources for VNFs, with asymmetric costs for under- and over-provisioning. To this end, and without any loss of generality, we use the objective, proposed in [3], see (3), which captures well the nature of our target.

[4]Although we picked as traffic image a square grid, the methodology can be applied to more general structures.

**Performance metrics.** In this simulation, we will focus on two metrics, where both are measured over the $K$ *test data samples*. The first is the average cost $C$, expressed as:

$$C = \frac{1}{K}\sum_{k=1}^{K}\Big(C_{L,k}\cdot l_k + C_{R,k}\cdot(1-l_k)\Big), \qquad (13)$$

where $C_{R,k}$ is the allocation cost of the $k$-th sample when using the remote exit $\mathbf{y}_{R,k}$; $l_k$ (defined in Section III-C) indicates an early (local) exit decision for sample $k$. The second is the percentage of test samples resolved locally at the edge, defined as:

$$L = \frac{1}{K}\sum_{k=1}^{K} l_k. \qquad (14)$$

**Baselines.** We discuss five methods that will be used for comparison against our proposed solutions. The first four are DDNN-based, with layers $\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2$ optimized using joint-exit training, see (9). Later when we show a plot of results for a DDNN, the same $w > 0$ was used to train all said baselines. The last baseline is centralized.

***Cloud:*** A *DDNN* which resolves all samples at the remote cloud; that is, it uses always $\mathbf{y}_R$ for the resource allocation. The cost achieved by this baseline is denoted as $C_{\text{cloud}}$ and should be expected to be low (good) as it makes use of the cloud layers for every sample; by design it has $L = 0\%$.

***Edge:*** A *DDNN* resolving all samples at the early exit, i.e. $\mathbf{y}_L$. We call its cost as $C_{\text{edge}}$, and we expect that it is rather high (bad), as it makes use of the few edge layers for every sample; by design it has $L = 100\%$.

***Random:*** A *DDNN* whose decision to resolve a sample locally, $l$, is an i.i.d. Bernoulli random variable with success probability $L_0$. As a result, the percentage of samples resolved locally is (on average) simply $L = L_0$. In our simulations, we show results of the DDNN with *Random* offloading using a $L_0 \in [0, 1]$; this will return a range of costs $C$, which we should expect to be between $C_{\text{cloud}}$ (for $L_0 = 0$ offloads all the samples to the remote cloud) and $C_{\text{edge}}$ (when $L_0 = 1$, it resolves all samples locally).

***Oracle:*** A *DDNN* with access to the test set *beforehand*, which solves the problem optimally offline. For a given $L_0$, it uses Algorithm 1, and sends to the cloud only the actually $1-L_0$ "hard" samples — said differently, the ones for which $C_L - C_R > b^*$. In our simulations, we plot the resource allocation cost for increasing $L_0$ (or increasing $b^*$). We stress that this an unrealistic, ideal and non-causal benchmark which helps for comparison purposes mostly.

**Remark on *Oracle*:** The *Oracle* can outperform the *Cloud* in terms of allocation cost. This can be explained by the fact that there are samples for which $C_L < C_R$, i.e., the local exit has lower provision cost. The *Oracle* will correctly use the early exit for these samples, achieving even lower cost compared to resolving everything at the remote cloud.

***DeepCog:*** A DNN *entirely* placed at the cloud, with a single final exit, which is not using joint-exit training. This baseline uses the core principles of [3]: (i) traffic image inputs, (ii) 3D-CNNs, and (iii) centralized training. Since it is based on the remote cloud, it has $L = 0$ by default. We denote its cost as $C_{\text{DeepCog}}$. Notice that the key difference between *Cloud* and *DeepCog* is that the former uses joint-exit training, while the latter does not.

TABLE II: List of baselines for performance comparison

| | Joint-exit training | Cloud offloading |
|---|---|---|
| *Cloud* | ✓ | always |
| *Edge* | ✓ | never |
| *Random* | ✓ | random w.p. $1 - L_0$ |
| *Oracle* | ✓ | if $C_L - C_R > b^*$ |
| *DeepCog* [3] | ✗ | is cloud based |

The training methods and the offloading policies for the baselines are summarized in Table II.

### *B. Deep Learning Resources*

Two important considerations we need to make when using deep learning are the resources that are needed, both in latency and energy, during the inference phase of the model.

For our experiments, we used google colab [26], a free platform where users can develop deep learning projects on GPU-enabled servers. Each user is allocated an NVIDIA Tesla T4, which can execute up to 8.1 TFLOPs, with a RAM of 16 GB [27].

**Inference latency.** First, we provide some data related to latency. Notice that while the setups where such distributed architecture could be applied may vary, below we try to devise a scenario with realistic values, as a reference, to better illustrate the potential latency savings. The two sources of inference latency are the following. (A) Communication: In a recent systems-oriented study, to emulate an edge-cloud connection in a wide area network (e.g., via 4G or beyond), the authors set the RTT to 42.46 ms [28]. (B) Computation: In our case, it is the execution times for layers $\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2$. We use those layers for inference on 714 samples (our test data) 10 times; the average times per sample are as follows: (i) $T_{\mathbf{F}_0} = 0.1$ ms, (ii) $T_{\mathbf{F}_1} = 0.12$ ms, and (iii) $T_{\mathbf{F}_2} = 0.23$ ms. According to those values, the more samples we resolve locally, the lower the inference latency. As an example, if $L = 30\%$, that would translate to an expected inference time $T = 0.3 \times (T_{\mathbf{F}_0} + T_{\mathbf{F}_1}) + 0.7 \times (T_{\mathbf{F}_0} + \text{RTT} + T_{\mathbf{F}_2})$.

**Energy and size.** Second, we report some figures regarding the consumed energy and the size of our DDNN. The former is measured, as is common in the deep learning community, in floating point operations (FLOPs) and multiply-accumulate operations (MACs); while for the latter we report the number of parameters (also called weights)[5]. Our DDNN architecture consists of two models, one at the edge and one at the remote cloud; below we report these three numbers for both. Note that the differentiation between the two is important since the edge is in practice more resource-constrained than the remote cloud. To make these measurements more meaningful, and for the sake of comparison, we provide the same data for the well-established AlexNet model, which can be deployed onto mobile devices [29]. From Table III, our cloud model

[5]In that table, "M" stands for mega.

TABLE III: Deep learning resources

| | MFLOPs | MMACs | MParameters |
|---|---|---|---|
| *DDNN-Edge* | 15.3 | 7.6 | 6.3 |
| *DDNN-Cloud* | 222.1 | 111.0 | 12.7 |
| *AlexNet [30]* | 2859.5 | 1428.4 | 61.1 |

(i.e., $\mathbf{F}_2$), needed 222.1 MFLOPs, indicating that it is a rather light architecture—an order of magnitude less than AlexNet. What is more, the edge model (i.e., $\mathbf{F}_0, \mathbf{F}_1$), which is actually the resource-constrained model needs only 15.3 MFLOPs. In fact, looking at the rapidly improving edge AI hardware market, we observed that our architecture is much lighter than the current hardware capabilities for edge ML. An impressive edge AI hardware example is a recent product of NVIDIA, called Jetson TX2, which has capabilities of 1.2 TFLOPs and a memory of 8 GB [31].

### C. Worst- and Best-Case Tradeoffs

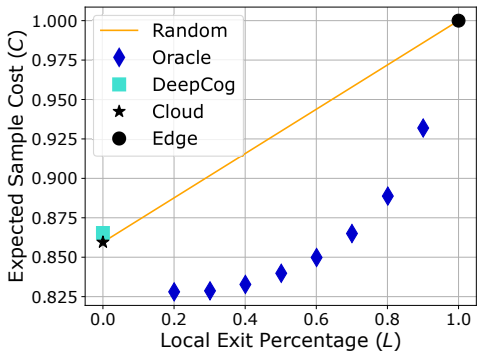
The goal of this subsection is twofold. *First*, we want to make sanity checks on the training of DDNN. In particular, we would like to draw some conclusions on the difference between costs of local and remote exits, ($C_L$) and ($C_R$). If $C_R$ is better than $C_L$ for most samples, this implies that the overall cost performance can benefit by offloading samples to the remote cloud, i.e., do resource allocation using also $\mathbf{F}_2$. For this, we will use the naive methods *Cloud* and *Edge*. *Second*, we would like to understand the performance limits of a DDNN. To this end, we leverage the *Oracle* to answer the following question: for given $L_0$ requirement (or critical benefit $b^*$), what is the best possible combined cost (obviously using both $\mathbf{y}_L$ and $\mathbf{y}_R$ depending on which one is better) that a DDNN can achieve.

In each plot of Fig 3, we have chosen a different under-provision penalty; in Fig. 3(a), we have $c_u = 0.5$ and in Fig. 3(b), we have selected $c_u = 10$. The overprovision on both cases was penalized linearly with $c_o = 1$; see objective function in (3). For each objective, we train a *DeepCog* and a DDNN with joint-exit training (see (9)) with $w > 0$. The joint training of the DDNN corresponding to $c_u = 0.5$ and Fig. 3(a) was done using $w = 0.2$, whereas the one of $c_u = 10$ Fig. 3(a) was with $w = 0.4$. In all cases, we used a window of past samples $N = 6$. The results for different values of $w > 0$ do not vary much; however, we chose to show the ones that empirically demonstrated the most interesting performance.

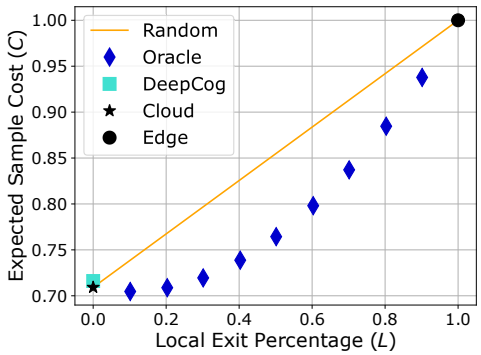
**Cost difference of naive methods.** We compare the cost of *Cloud* and *Edge*, using the same trained DDNN. First, in Fig. 3(a), we can see a 14% difference between the local and the remote exit $C_{\text{edge}}$ and $C_{\text{cloud}}$, while in Fig. 3(b), the respective difference is around 30%. This gap indicates that for this specific task, the remote layers offer added benefit as they achieve lower cost. Had we observed $C_{\text{edge}} = C_{\text{cloud}}$, this would imply that the local layers suffice for the task.

**Performance limits of DDNN.** The second aim is to compare the non-causal *Oracle* with the rest of the baselines.

*Oracle:* For increasing constraint requirement $L_0$, every blue diamond represents the optimal resource allocation cost




(a) $c_u = 0.5$




(b) $c_u = 10$

Fig. 3: Baselines performance curves. $x$ axis is $L$ and $y$ axis is cost $C$, for all test data, when using the baselines of the legend.

$C$ (in $y$-axis), see (13), and its corresponding percentage of samples that was resolved at the edge $L$ (in $x$-axis), see (14). On the left, we start by demanding $L \geq L_0 = 0$, and almost all samples are naturally resolved at the cloud ($L$ is very low); and $L_0$ increases, more samples are resolved at the edge. We observe the following two points. (a) Cost $C$ is increasing as more samples are resolved locally; and it does so in a "convex" fashion (with upwards curvature), suggesting that a *perfect* offloading rule is able to classify as "easy" only the samples whose benefit $b$ (of using the remote cloud) is indeed very low, and thus the average cost $C$ is not severely increased. (b) With respect to *DeepCog*, in Fig. 3(b) we can see a cost increase of only 6% when a huge 50% portion of the samples are resolved locally. In addition, in the adjacent plot, Fig. 3(a), when $L = 70\%$ of samples are resolved locally (thus speeding up the expected inference time by a lot), the *Oracle* loses nothing in cost performance; that is the $y$-axis value of *Oracle* at $L = 70\%$ is almost the same as the $C$ of *DeepCog* which resolves all samples at the cloud ($L = 0$).

*Random:* We vary $L_0$ from $0 \to 1$ and observe the cost $C$ increasing *linearly*, starting from $C_{\text{cloud}} \to C_{\text{edge}}$. The linear behavior indicates that this policy serves as a worst-case performance bound; any DDNN with offloading whose cost value lies above this line is essentially not useful[6].

**Implication.** Comparing the *Oracle* with the *Random*, we

[6]It is easy to construct an even worse offloading policy, i.e., pick the exit with the highest cost, however, this does not give a meaningful bound.

conclude that at least the non-causal *Oracle* can do much better than *Random*. On the contrary, if the two curves of these two coincided, there would no hope for any DDNN with an online offloading policy to do better. Our desire is that our online offloading policies to have performance curves that lie close to the *Oracle*.

**Take-away #1.** A DDNN with *Oracle* offloading can have the same resource provisioning cost as *DeepCog*, while resolving $L \approx 70\%$) of the samples quickly at the edge.

**Remark on Fig. 3(a).** When $b^* = 0$, the *Oracle* resolves around $20\%$ of the samples locally. Hence, for $20\%$ of the test samples, the local exit has better (lower) cost than the remote cloud. Notably, the phenomenon that *more* layers return a *worse* result (higher cost) is not new, and has been coined as "DNN overthinking"; for more details see [32]. This is a follow-up to the remark we made on the *Oracle* in the previous subsection.

### *D. DDNNs with our Proposed Online Offloading Policies*

This subsection comprises the main body of our results, where we present the findings for DDNNs, coupled with our proposed offloading policies. We first explain how our results were generated, and then investigate how our proposed methods fare against the baselines.

We train four DDNNs (layers $\mathbf{F}_0$, $\mathbf{F}_1$, $\mathbf{F}_2$) using the following parameters. First, Fig. 4(a): $c_u = 0.5$ (underprovision penalty of objective function), $N = 6$ (window of past samples) with $w = 0.2$; second, Fig. 4(b): $c_u = 1$, $N = 6$ with $w = 0.2$; third, Fig. 4(c): $c_u = 10$, $N = 25$ with $w = 0.4$; and finally fourth, Fig. 4(d): $c_u = 20$, $N = 25$ with $w = 0.7$. In all of them, overprovision is penalized linearly with the difference $y - d$. To generate our results, we pick a trained DDNN and use an offloading policy that decides where the sample will be resolved. We describe below how a point $(L, C)$ in the plots of Fig. 4 is generated.

***Uncertainty Rule*.** First, the input $\mathbf{D}$ is forward-passed $B$ times. Then the uncertainty $U$ is computed using (10), and is compared to threshold $u$ (set by the operator). This is done for all data points of the test set. To generate multiple performance regimes $(L, C)$ (see Fig. 4), we started with $u = 0$. For this value, all samples are exited remotely (i.e., $L = 0\%$), implying no uncertainty tolerance. We gradually increased $u$ to $0.5$; for $u = 0.5$, the rule exited all samples locally, i.e., $L = 100\%$. It should be noted that $u = 0.5$ was found empirically, by means of multiple simulations. In between those extreme values, we chose a small step size to generate multiple $(L, C)$ points of the plots in Fig. 4. Finally, tuning $B$ is also not straightforward: large $B$ might be required to get reliable estimates of $U$, but in practice each extra forward-pass incurs a cost. We show two versions of the *Uncertainty Rule*, with $B = 2$ and $B = 10$. As it can be already understood, this mechanism demands significant tuning effort from the operator [6]; and in fact, this was our main driving force for designing the next offloading rule which is more principled and easier to tune.

***Optimized Rule*.** For a given $L_0$, we train (offline) a single DNN model which learns to offload (to the cloud) samples that are hard to predict. We forward-pass all the test samples, and the DNN (i.e., the *Optimized Rule*) classifies every sample as "local" or "remote" with one forward-pass. To generate more $(L, C)$ points, we increase $L_0$ which leads to an increase in $L$. Compared to the *Uncertainty Rule*, this rule does not demand a lot of tuning. The operator only needs to train an ML model by inserting the percentage of sample $L_0$ that they wish to resolve locally.

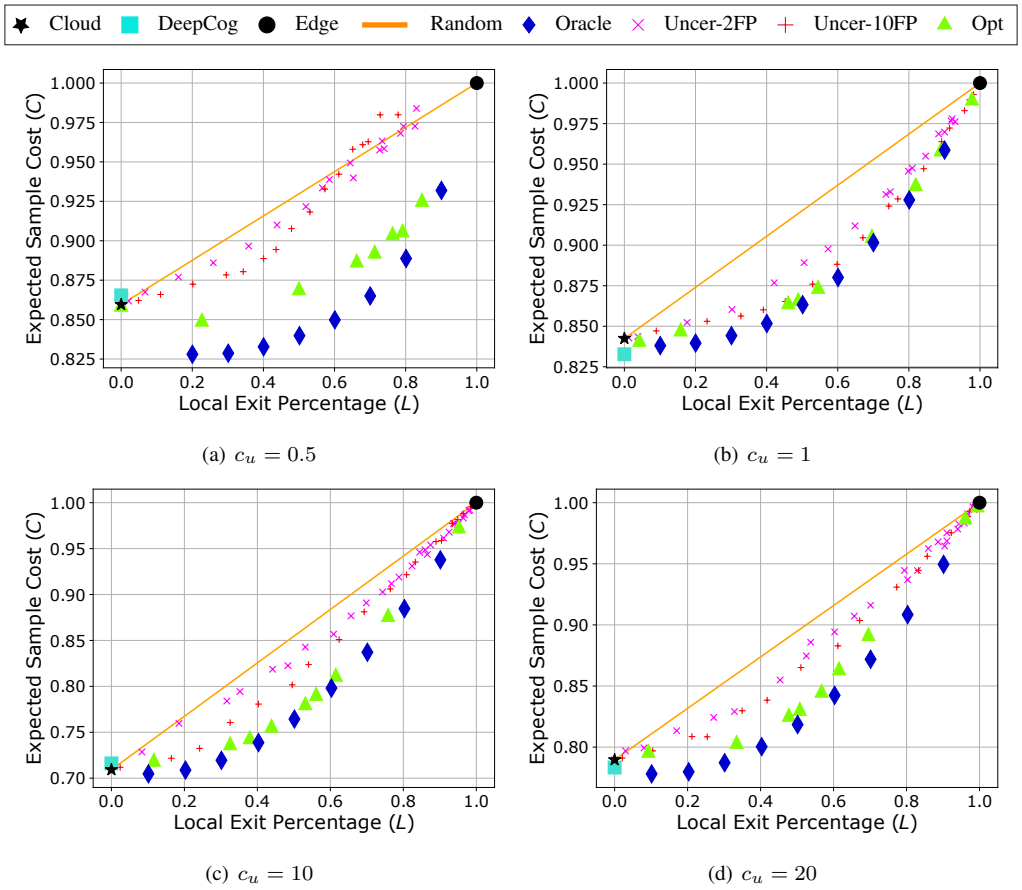
**Main result: Performance curves.** We investigate how our two offloading policies, together with a trained DDNN, fare against (a) the *DeepCog*, and (b) the upper (*Oracle*) and lower (*Random*) bound baselines. In general, as one would expect, the cost $C$ increases as $L$ goes from $0 \rightarrow 1$; in other words, the more we allocate resources based on $\mathbf{y}_L$, the higher the average cost $C$ becomes (fewer layers).

With respect to the centralized *DeepCog*, as shown in Fig. 4(a) (cyan square at $L = 0\%$), we see the following: (a) When the *Uncertainty Rule* resolves $L = 40\%$ at the edge and its $C$ is only $2\%$ higher. (b) Even more strikingly, the *Optimized Rule* resolves 50% of the samples at the edge, while losing nothing in provisioning cost. In more detail, in Fig. 4(b), an only 5% performance degradation allows the *Optimized Rule* to resolve 50% of the samples locally; a big improvement in latency. Similar merits are observed in Fig. 4(c), where a $5\%$ cost increase allows us to raise the utilization of local exit to $L = 40\%$. Finally, in Fig. 4(d), the *Uncertainty Rule* resolves about 42% of samples locally while increasing 5% of cost; and the *Optimized Rule* increases the cost by 7% while resolving 58% at the edge.

We conclude by comparing our two proposed solutions, and observe three points. First, in all of our plots, the *Optimized Rule* outperforms the *Uncertainty Rule*. Second, the *Optimized Rule* is in three out of four cases very close to the *Oracle*, and in no case close to the *Random*. Third, the *Uncertainty Rule* has more unpredictable behavior, as in Fig. 4(a) it is close to *Random* and in Fig. 4(b) close to *Oracle*. Representatively, in Fig. 4(b), we can observe that the *Uncertainty Rule* performs close to *Oracle* when it makes 10 forward-passes; however, it is close to *Random* when $B = 2$ (2 samples are not sufficient to estimate $U$). On the other hand, the *Optimized Rule* outperforms both and is close to the *Oracle* upper performance bound. Finally, in Fig. 4(c), the *Uncertainty Rule* is better than *Random* (and actually better as $B$ increase), but again the *Optimized Rule* outperforms it and lies close to the *Oracle*.

While our focus is on the cost paid by the operator, as computed by (13); another important cost we account for during inference is the energy in order to take the offloading decision. Although the *Uncertainty Rule* performs well in some occasions, it needs many ($B = 10$) forward-passes at the local exit to estimate $U$, and these operations are costly. On the other hand, the *Optimized Rule* does only one forward-pass, consuming, as a result, much less energy. This is just a hint on the issue of energy; we believe that more detailed energy-oriented modeling should be done to study this appropriately, and defer this for future work.

**Take-away #2.** The *Optimized Rule* exhibits an almost ideal




(a) $c_u = 0.5$ (b) $c_u = 1$

(c) $c_u = 10$ (d) $c_u = 20$

Fig. 4: Proposed methods plus baselines performance curves. $x$ axis is $L$ and $y$ axis is cost $C$, for all test data, when using the methods of the legend.

behavior; it is consistently better than *Random*, and in many cases it is close to the optimal *Oracle*.

### *E. Further Studies on the Optimized Rule*

Having established the benefits of the proposed *Optimized Rule*, we study it further. The results that follow correspond to the case when the underprovision penalty $c_u = 20$, and the overprovision is linear.

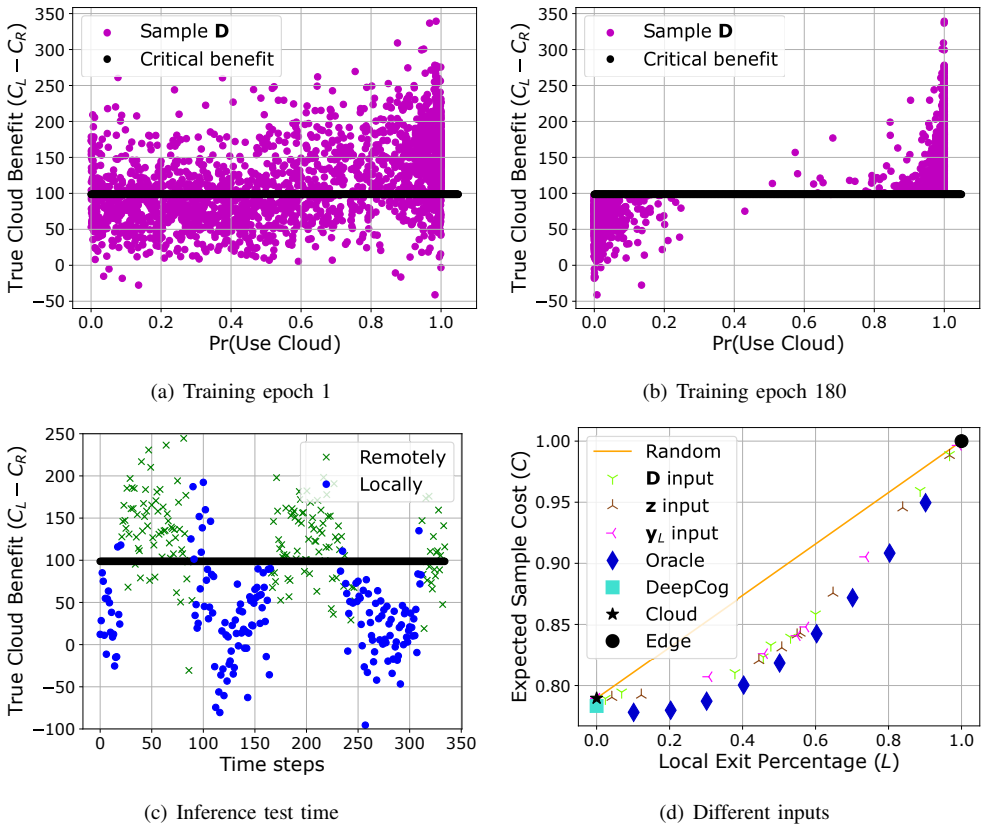
***-Training.*** In Figs. 5(a), 5(b), we see two scatter plots, where a dot corresponds to a training sample $\mathbf{D}$. In particular, $y$-axis is the benefit of using the remote cloud, $C_L - C_R$; while $x$-axis is the probability with which the *Optimized Rule* offloads remotely, allocating resources based on $\mathbf{y}_R$.

In these plots, the *Optimized Rule* has been trained using a single $L_0$, which resulted to a critical benefit $b^* \approx 100$ (noted with a black horizontal line); see Section IV-B. In epoch 1, the samples are randomly placed as expected; this suggests that the untrained *Optimized Rule* returns random decisions. However, after 180 epochs, the desired behavior is observed: the samples with benefit *lower* than the critical one (black line) start getting probability less than 1/2 to be offloaded. On the contrary samples with benefit *higher* than $b^*$ are sent to the cloud, which is exactly the behavior we we wanted to observe. Further, observe that samples with benefit *much less* than $b^*$ are given a probability close to 0.0; whereas the ones with a benefit much higher than $b^*$ are confidently sent to the cloud, with their probability being close to 1.0. Notice that while Fig. 5(b) implies overfitting, this is not the model we extracted from the training; we show it in order to exhibit that the ML model of the *Optimized Rule* was able to classify accurately the training samples.

***-Test.*** In Fig 5(c), on the $y$-axis we plot the cloud benefit values, $C_L - C_R$ of the test dataset plotted on time ($x$-axis); the black line corresponds to the critical benefit. The experiment we see in this plot was carried out in two stages: (a) we compute the benefit values of the test samples offline, which gives us the true labels ("easy" or "hard"); then we simulate the real-time inference, allowing the *Optimized Rule* to make its decisions. With green we mark the samples for which the *Optimized Rule* returned $l = 0$, whereas with blue the ones for which $l = 1$.

**Take-away #3.** The *Optimized Rule*, using only the features of sample $\mathbf{D}$, can successfully infer for the majority of samples if the cloud benefit exceeds $b^*$.

**Different input features in *Optimized Rule* DNN.** It is clearly possible to use other local signals to train the *Optimized Rule*, with the tradeoff in that case being potentially better accuracy vs higher training time, or more expensive *Optimized Rule* ML model. Although this is an interesting topic, it is orthogonal topic to our work; nonetheless, to better fortify our methodology, we tried other candidate signals in order to see if significant changes will occur. Specifically




(a) Training epoch 1

(b) Training epoch 180

(c) Inference test time

(d) Different inputs

Fig. 5: (a,b,c): Detecting samples with benefit higher than $b^* \approx 100.0$, $c_u = 20$. Plots (a,b) depict the evolution of the training, while (c) shows the testing. (d): Performance curves (testing) when feeding the *Optimized Rule* DNN different local signals as input.

TABLE IV: Latency for *Optimized Rule* (DNNs are on GPU)

| $L$ (%) | 4.2 | 15.8 | 46.2 | 54.4 | 69.6 | 81.9 | 97.7 |
|---|---|---|---|---|---|---|---|
| Latency (ms) | 41.0 | 36.1 | 23.1 | 19.6 | 13.2 | 7.9 | 1.2 |

these were: $\mathbf{D}$, $\mathbf{z}$, and $\mathbf{y}_L$. From the three, $\mathbf{z}$ is the richer one, as it is an encoded (latent) version of $\mathbf{D}$; recall that $\mathbf{z} = \mathbf{F}_0(\mathbf{D})$. To this end, we train three DNNs based on MLPs, each receiving one of these inputs. Interestingly, as we can see in Fig. 5(d), no major advantages were achieved by different features, at least in this scenario; nonetheless, a detailed investigation is of course open for future work.

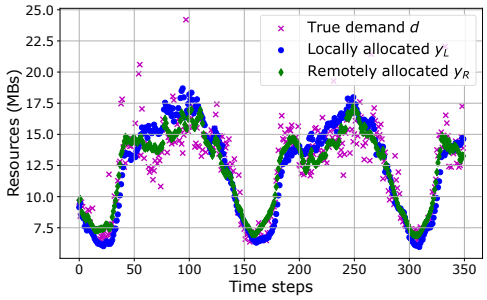
**Latency improvement.** In Table IV we show some results that correspond to Fig. 4(b). In particular, from that plot we list the values of $L$, achieved by the *Optimized Rule* and of the corresponding latency we measured. When a sample is resolved locally, it experiences the computation latency of $\mathbf{F}_0, \mathbf{F}_1$ – while when it is resolved remotely it experiences $\mathbf{F}_0, \mathbf{F}_2$ and the RTT [28]. We see clearly that as $L$ grows, the inference latency drops; in fact, for example when the *Optimized Rule* resolves $L = 81\%$ of samples locally, it has 12% higher cost than the centralized baselines (*DeepCog* and *Cloud*), while having 5 times lower inference latency.

**Local and remote exits.** Finally, we consider the effect of increasing the penalty paid $c_u$ when underprovision occurs. In Fig. 6, we see the resource provision outputs of the edge and the remote cloud along with the true demand. On the left, the penalty is low, $c_u = 1$, and hence neither exit "worries" too much about underprovisioning. As a result, the output looks more like the DNN exits are trying to track the demand, rather than be safe and avoid underprovisioning. On the right, increasing $c_u$ to 20 leads to a drastically different resource provision, as we can see that both exits are trying to heavily overprovision in order to avoid the penalty $c_u$. Notice, however, that in the latter plot it is more evident that the remote allocation $\mathbf{y}_R$ *overprovisions less* than $\mathbf{y}_L$.

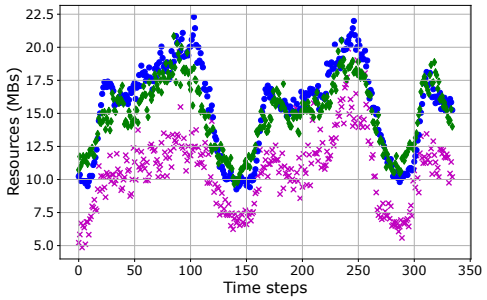
## VI. Related Work

Our work applies early-exit DDNNs on a resource allocation application in slicing. Below, we discuss related art with respect to the application and the DDNN methodology.

**Resource allocation for slicing.** The problem of resource allocation has been examined by a variety of angles; it has been addressed using DNN-based approaches [3], [4] and stochastic control methods [33]. Additionally, slicing has been a fruitful field of application for other data-driven methods such as Online Convex Optimization (OCO) [23], and Reinforcement Learning (RL), see e.g., [34], [35]. Moreover, in slicing, a more general problem is the service function chain embedding, which deals with the placement and reconfiguration of VNFs and virtual links on top of the physical network infrastructure. Some works resolving this problem typically use techniques based on algorithmic heuristics for scalability reasons, such as [7], [10]–[15]. What is more, fairly recently, with the resurgence of Continuous




(a) $c_u = 1$




(b) $c_u = 20$

Fig. 6: Example VNFs true demand, edge/local exit allocation and cloud/remote allocation; as $c_u$ increases the allocation becomes "safer".

Optimization, an ADMM-based algorithm framework was proposed in [9], which exhibited fast convergence properties.

**Early exit DDNNs.** The viewpoint of DDNNs [5], [32], [36]–[38] with early exit(s) is a recent and promising research avenue with many under-explored applications. In this line of research, a key quantity is the amount of information transmitted to the remote layers. Recent works have proposed systematic ways of compressing that information (see $\mathbf{z}_t$ in Fig. 1), e.g., by dynamically adjusting the sent features depending on network conditions [38]–[40]. Furthermore, on a different note, a more theoretical work was presented in [41]. The authors, assuming that more layers imply a better accuracy, formulated the offloading decision-making problem as an online learning problem and managed to achieve non trivial regret guarantees.

**Progressive inference.** In this approach the DNN is split in two locations, but all samples are forward-passed through all layers. As a result, the expected accuracy of the model remains fixed. The first work using this approach was the seminal measurement-oriented Neurosurgeon [29], where the authors experimented on known architectures, such as AlexNet [30], and explored non-trivial energy/latency tradeoffs, depending on different wireless conditions (3G, LTE, WiFi), as well as different DNN execution units (CPU, GPU). Another work in the same spirit is the systems-oriented Couper [28]. Under the assumption that the whole DNN is containerized in the device, in the edge, and in the cloud, the authors explore practical scheduling algorithms to identify the split layer under dynamic conditions. The application studied in Couper is visual analytics, and there, the tradeoff is between lost frames and latency.

**Positioning.** Finally we stress *how we differentiate* with closely related work. Compared to [3], the novelties are: (i) distributed inference, (ii) per VNF, rather than aggregate, resource allocation, (iii) significant performance benefits, both in terms of overhead/latency and often in terms of allocation cost. With respect to [5], we differentiate at the ML task, namely multivariate time series prediction instead of image classification. This essentially leads to drastically different (and less obvious) offloading policies; i.e., (i) an *Uncertainty Rule* that uses dropout of the local exit during inference, and (ii) an *Optimized Rule* based on supervised learning that acts directly on the sample. Finally, and more generally (iii), we define two baselines (i.e., *Random*, and more importantly *Oracle*) which give a better understanding on whether the offloading rule performs well or not.

## VII. Discussion and Future Work

Our DDNN architecture is a promising solution for fast resource provisioning, and a first step towards DDNNs for time-series related tasks. Below, reflecting on the limitations of our work, we discuss some avenues for future research.

**Hierarchical DDNNs.** The distributed inference model we use here is fairly simple, having one early exit at the edge and a second one at the remote cloud. Further, we have assumed that VNF demands are collected at the edge and are processed together in the $\mathbf{F}_0$ block. However, there exist hierarchical DDNN architectures [5] where, for instance, one could have a *small* DNN per VNF, to instantly do the resource allocation. If the small DNN is uncertain, then an output of that DNN could be sent to the collective edge $\mathbf{F}_0$ (as we do now) for further processing. In some sense, this creates new local exits, which are of different nature than the collective local we have now; in this hierarchical model, the that $\mathbf{F}_0$ might receive an incomplete view of the traffic image.

**Joint optimization of DDNN layers and offloading.** We decomposed the optimization of the system in two steps: (a) joint-exit training for the DDNN layers and the exits, and (b) given the trained layers, the design of the offloading mechanism. A perhaps more challenging approach is to design an algorithm that will optimize the *end-to-end* system jointly, for some specific inference latency requirement.

**Distributed Recurrent NNs.** We leveraged CNNs to exploit the spatiotemporal features of the data, for more details, interesting reads are [1], [2], [42]. Nevertheless, a more natural alternative for such regression-like tasks involving time-series signals is RNNs (e.g. LSTMs). A major characteristic of RNNs, however, is that they use an internal state to process a sequence of inputs, suggesting strong correlations between consecutive samples. That recursive nature of RNNs creates further, but interesting, complications when it comes to i) distributing their layers, ii) training their layers, and iii) designing the offloading algorithm. On the other hand, the upside of using CNNs is that they treat each sample independently (no memory or hidden state), and therefore, their layers can be naturally split and use multiple exits.

**Reinforcement Learning-based offloading.** An assumption we made in Section III-C is that channel conditions, through-

out the period we examine, are fixed. This allows us to claim that average RTTs are fixed. However, if this assumption is removed, we could have more complicated situations. For example, we can offload a hard decision to the cloud with good channel conditions, but when the actuation signal $\mathbf{y}_{R,t}$ is sent back to the edge, the conditions have changed (e.g., congested core-cloud link), rendering the said offloading decision extremely harmful in terms of latency. This more general setup requires different modeling, and to our opinion the correct tool is Reinforcement Learning (RL), since the underlying problem could be essentially cast as a Markov Decision Process (MDP). A good starting point, along this direction, could be to model the MDP, using intuition from the measurements and the findings of [29], where the authors showed different latency and energy consumption for different wireless connectivity.

**Budgeted resource allocation.** Finally, an assumption we made is that resources $\mathbf{y}$ we can provision are not constrained; the same is assumed also in [3]. In fact, resource allocation with DNNs is challenging, due to the inherent difficulty of imposing constraints on DNN outputs [43]. To our opinion, OCO approaches [44] where one can readily impose and treat (instantaneous or long-term) constraints with projections, or primal-dual approaches, are more promising alternatives.

## VIII. Conclusions

We have developed a promising DDNN framework for the task of resource provisioning for 5G+ networks. The proposed architecture consists of two exits: a local and a remote, which have to be trained jointly. The joint training forces the local exit to improve its performance, but also acts as a regularizer for the remote one. More importantly, we pair the DDNN with an offloading mechanism whose role is to decide which samples are worth resolving at the remote cloud and which not. To serve as an offloading mechanism we proposed two methods. First, a Bayesian-inspired *Uncertainty Rule*, which aims to detect the samples of high uncertainty, and send them to the remote cloud for further processing. Second, we formulate the offloading as an offline constrained optimization problem, and define an algorithm that performs a novel sample labeling; we then train an ML model (via Supervised Learning) that tries to mimic the offline optimal (*Optimized Rule*). Comparing our DDNN model with its equivalent centralized one, we observed that it is possible to resolve $\approx 60\%$ of the incoming samples locally, with less than 5% in cost performance loss.

## References


[1] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications surveys & tutorials*, vol. 21, no. 3, pp. 2224–2287, 2019.

[2] J. Wang, J. Tang, Z. Xu, Y. Wang, G. Xue, X. Zhang, and D. Yang, "Spatiotemporal modeling and prediction in cellular networks: A big data enabled deep learning approach," in *IEEE INFOCOM 2017*.

[3] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Deepcog: Cognitive network management in sliced 5g networks with deep learning," in *IEEE INFOCOM 2019*.

[4] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Aztec: Anticipatory capacity allocation for zero-touch network slicing," in *IEEE INFOCOM 2020*.

[5] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *IEEE ICDCS 2017*.

[6] T. Giannakas, T. Spyropoulos, and O. Smid, "Fast and accurate edge resource scaling for 5g/6g networks with distributed deep neural network," *IEEE WoWMoM*, 2022.

[7] S. Vassilaras, L. Gkatzikis, N. Liakopoulos, I. N. Stiakogiannakis, M. Qi, L. Shi, L. Liu, M. Debbah, and G. S. Paschos, "The algorithmic aspects of network slicing," *IEEE Comms Magazine 2017*.

[8] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[9] M. Leconte, G. S. Paschos, P. Mertikopoulos, and U. C. Kozat, "A resource allocation framework for network slicing," in *IEEE INFOCOM 2018*.

[10] Y. Wang, C.-K. Huang, S.-H. Shen, and G.-M. Chiu, "Adaptive placement and routing for service function chains with service deadlines," *IEEE Trans. on Network and Service Management*, 2021.

[11] B. Farkiani, B. Bakhshi, S. A. MirHassani, T. Wauters, B. Volckaert, and F. De Turck, "Prioritized deployment of dynamic service function chains," *IEEE/ACM Trans. on Networking*, 2021.

[12] Y. Bi, C. C. Meixner, M. Bunyakitanon, X. Vasilakos, R. Nejabati, and D. Simeonidou, "Multi-objective deep reinforcement learning assisted service function chains placement," *IEEE Trans. on Network and Service Management*, 2021.

[13] D. Zheng, G. Shen, X. Cao, and B. Mukherjee, "Towards optimal parallelism-aware service chaining and embedding," *IEEE Trans. on Network and Service Management*, 2022.

[14] J. Li, W. Liang, W. Xu, Z. Xu, X. Jia, A. Y. Zomaya, and S. Guo, "Budget-aware user satisfaction maximization on service provisioning in mobile edge computing," *IEEE Trans. on Mobile Computing*, 2022.

[15] Y. Yue, B. Cheng, M. Wang, B. Li, X. Liu, and J. Chen, "Throughput optimization and delay guarantee vnf placement for mapping sfc requests in nfv-enabled networks," *IEEE Trans. on Network and Service Management*, vol. 18, no. 4, pp. 4247–4262, 2021.

[16] F. Debbabi, R. Jmal, L. C. Fourati, and R. L. Aguiar, "An overview of interslice and intraslice resource allocation in b5g telecommunication networks," *IEEE Trans. on Network and Service Management 2022*.

[17] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets," in *Artificial intelligence and statistics*, pp. 562–570, PMLR, 2015.

[18] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE Comms Magazine 2017*.

[19] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *ICML 2016*.

[20] L. Zhu and N. Laptev, "Deep and confident prediction for time series at uber," in *IEEE ICDMW 2017*.

[21] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "Competitive randomized algorithms for nonuniform problems," *Algorithmica*, vol. 11, no. 6, pp. 542–571, 1994.

[22] T. Italia, "Milano Grid," 2015.

[23] N. Liakopoulos, G. Paschos, and T. Spyropoulos, "Robust user association for ultra dense networks," in *IEEE INFOCOM 2018*.

[24] J. Paparrizos and L. Gravano, "K-shape: Efficient and accurate clustering of time series," in *ACM SIGMOD 2015*.

[25] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[26] "Google colab." https://colab.google/. Last accessed: December 23, 2024.

[27] "NVIDIA Tesla T4." https://www.techpowerup.com/gpu-specs/tesla-t4.c3316. Last accessed: December 23, 2024.

[28] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska, "Couper: Dnn model slicing for visual analytics containers at the edge," in *ACM/IEEE Symposium on Edge Computing 2019*.

[29] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

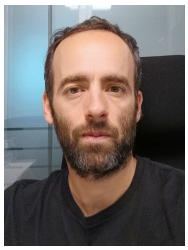
[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *NeurIPS 2012*.

[31] "NVIDIA Jetson TX2." https://www.nvidia.com/en-eu/autonomous-machines/embedded-systems/jetson-tx2/. Last accessed: December 23, 2024.

[32] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," in *ICML 2019*.

[33] A. T. Z. Kasgari and W. Saad, "Stochastic optimization and control framework for 5g network slicing with effective isolation," in *2018 CISS 2018*.
[34] Q. Liu, T. Han, and E. Moges, "Edgeslice: Slicing wireless edge computing network with decentralized deep reinforcement learning," in *IEEE ICDCS 2020*.
[35] A. Okic, L. Zanzi, V. Sciancalepore, A. Redondi, and X. Costa-Pérez, "$\pi$-road: a learn-as-you-go framework for on-demand emergency slices in v2x scenarios," *arXiv preprint arXiv:2012.06208*, 2020.
[36] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *ICPR 2016*.
[37] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, "Why should we add early exits to neural networks?," *Cognitive Computation*, vol. 12, no. 5, pp. 954–966, 2020.
[38] S. P. Chinchali, E. Cidon, E. Pergament, T. Chu, and S. Katti, "Neural networks meet physical networks: Distributed inference between edge devices and the cloud," in *ACM HotNets 2018*.
[39] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *ACM MobiCom 2020*.
[40] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019*.
[41] M. K. Hanawal, A. Bhardwaj, *et al.*, "Unsupervised early exit in dnns with multiple exits," *arXiv preprint arXiv:2209.09480*, 2022.
[42] A. Furno, M. Fiore, R. Stanica, C. Ziemlicki, and Z. Smoreda, "A tale of ten cities: Characterizing signatures of mobile traffic in urban areas," *IEEE Trans. on Mobile Computing*, 2016.
[43] H. Kervadec, J. Dolz, J. Yuan, C. Desrosiers, E. Granger, and I. B. Ayed, "Constrained deep networks: Lagrangian optimization via log-barrier extensions," in *EUSIPCO 2022*.
[44] M. Zinkevich, "Online convex programming and generalized infinitesimal gradient ascent," in *ICML 2003*.

**Theodoros Giannakas** received the Diploma in Electrical and Computer Engineering from the University of Patras, Greece, his MSc in Wireless Communications from the University of Southampton, UK, and his PhD in Computer Science and Networks from EURECOM, Sophia Antipolis, France. He has worked as a postdoctoral researcher in EURECOM from 2020 until 2021; and as a research engineer in Huawei Technologies, Paris Research Center, France, from 2021 to 2024. He currently works as a data scientist in Optasia, Athens, Greece. His main research interests include machine learning and optimization, with applications on networking and finance. He was a co-recipient of the Best Paper Runner-Up Award in IEEE WoWMoM 2021.



**Dimitrios Tsilimantos** received the Diploma and Ph.D. degree in Electrical and Computer Engineering from the National Technical University of Athens (NTUA), Greece, in 2004 and 2009 respectively. From 2011 to 2014 he was with the National Research Institute in Informatics and Control (INRIA) in Lyon, France, as a postdoctoral researcher. Since 2014 he has been with Huawei Technologies, Paris Research Center, France, where he is currently a principal researcher in the Advanced Wireless Technology Lab. His research interests include machine learning, multi-agent settings, network planning, resource management, energy efficiency, stochastic geometry, decision theory and video streaming, with main focus on cellular and wireless networks.

**Apostolos Destounis** was born in Athens, Greece. He obtained the Diploma in Electrical and Computer Engineering from National Technical University of Athens in 2009, the M.Sc. in Communications and Signal Processing from Imperial College London in 2010 and the PhD in Telecommunications from Supélec (now CentraleSupélec) in 2014. During his PhD, between 2011-2014, he was also with Alcatel-Lucent Bell Labs (now Nokia Bell Labs) France. From May 2014 to September 2022 he was with Huawei Technologies, Paris Research Center, France. Since October 2022, he is an AI Researcher at Ericsson Research France. His research interests include machine learning and optimization, with applications in the physical layer and resource allocation problems in wireless networks.

**Thrasyvoulos Spyropoulos** received the Diploma in Electrical and Computer Engineering from the National Technical University of Athens, Greece, and the PhD degree in Computer Engineering from the University of Southern California in 2006. He was a post-doctoral researcher with INRIA in 2007 and then, a senior researcher with the Swiss Federal Institute of Technology (ETH) Zurich, until 2010. From 2010 to 2022, he was a professor with EURECOM, France. He is currently a professor with the Technical University of Crete, Greece. He was a co-recipient of the Best Paper Award in IEEE SECON 2008, and IEEE WoWMoM 2012, as well as the Runner-Up for ACM MobiHoc 2011, IEEE WoWMoM 2015, and IEEE WoWMoM 2021.