Fast and accurate edge resource scaling for 5G/6G networks with distributed deep neural networks

Theodoros Giannakas, Thrasyvoulos Spyropoulos, Ondrej Smid

Fast and accurate edge resource scaling for 5G/6G networks with distributed deep neural networks

Theodoros Giannakas[1], Thrasyvoulos Spyropoulos[2], and Ondrej Smid[2]
[1] Paris Research Center, Huawei Technologies, France, theodoros.giannakas@huawei.com
[2] EURECOM, Sophia-Antipolis France, first.last@eurecom.fr

***Abstract*—Network slicing has been proposed as a paradigm for 5G+ networks. The operators slice physical resources from the edge, all the way to 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 edge resources (e.g. RAN), a question arises whether operators can: (a) scale edge resources fast enough (often in the order of ms) and (b) afford to transmit huge amounts of data towards a cloud where such a DNN-based algorithm might operate. We propose a Distributed-DNN 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 coupled with a Bayesian mechanism to intelligently offload a subset of (harder) decisions to additional DNN layers running at a remote cloud. Using the publicly available Milano dataset, we investigate how such a DDNN should be jointly trained, as well as operated, to efficiently address (a) and (b), resolving up to 60% of allocation decisions locally with little or no penalty on the allocation cost.**

I. Introduction

The advent of 5G networks has been characterized by a number of radical architectural transformations. Virtualization and slicing of communication, computation, and infrastructure resources allows operators to co-host multiple services and tenants, with a large variety of performance requirements and SLAs (service level agreements). What is more, 5G networks and beyond will be characterized by increased programmability through the use of composable virtual network functions, 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.

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 multi-objective, multi-variable optimization problems that are based on numerous modeling assumptions, like knowledge of key inputs, stationarity, etc. (not often satisfied in practice). As a result, flexible *model-free* methods based on modern machine learning (ML) methods have gained significant attention as an alternative way to tackle wireless network optimization problems arising in beyond 5G networks [1]. Such methods can learn *and* optimize at the same time various networks tasks, operating directly on offline or online 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], [2], [3], [4], the authors use a DNN architecture to predict the base station traffic, based on past samples. More recently, the authors of DeepCog [3] have used an interesting DNN-based approach to directly predict the amount of slice resources needed at a data center/cloud serving assigned BSs to avoid both *underprovision* (which could result in a costly SLA violation) and *overprovision* (which would waste valuable resources that another slice/tenant could put to use). As a result, this control/decision problem can be tackled with popular DNN architectures, by training an objective with appropriately tuned under- and over-provision components. A number of additional novelties are introduced in this work to improve performance. Perhaps the two most important ones, that will also serve as the starting point of our work[1], are:

1) Time-series, corresponding to different BS demands can be correlated (e.g., traffic in BSs near tram, metro, and other "commute" spots); this can be exploited by learning how to *jointly* predict good resource allocations of multiple BSs in parallel, using the same architecture;
2) A 3D Convolutional Neural Network (CNN) architecture could infer such correlations using an appropriately pre-processed image-like representation of past demands.

While this approach is promising, the standard assumption of a *centralized* implementation of the DNN architecture faces 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 phone) that can be "lazily" offloaded to a central computational cloud, the use of DNNs for controlling 5G edge resources (e.g., allocation

The research leading to these results has been supported by the H2020 MonB5G Project (grant agreement no. 871780). This research was conducted while Theodoros Giannakas was with EURECOM, Sophia-Antopolis.

[1]A recent extension of this work includes also reconfiguration costs, to avoid too frequent re-scaling [4], but these features we do not take into account in this work.

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, might violate these requirements. *Second,* constantly sending raw monitored data over possibly already congested links towards a DNN architecture lying deep in the core network (or even outside) has a prohibitive network footprint. Hence, in this work we are interested in how such DNN-based architectures could be appropriately *implemented in a distributed fashion* towards resolving both of the above concerns, yet without compromising the observed performance advantages of the DNN.

An interesting recent work has introduced the concept of Distributed Deep Neural Networks (DDNN) for an image classification application [5]. The key idea behind DDNNs is allowing to distribute the layers of a DNN between different locations, where prediction or control decisions can be taken at each location: e.g., *locally* (near the edge) if the latency requirement for a decision is stringent or the network links to the core are congested, or *remotely/centrally* if additional accuracy is needed. To achieve both tasks well, one needs to *jointly* train both the local and remote layers.

To this 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 can be summarized as follows:

(C.1) We propose an appropriate distribution of the layers of a 3D-CNN architecture, between an edge cloud and a core/remote cloud, and investigate how to jointly train this to provide both accurate local decisions and remote decisions for various objectives.

(C.2) At runtime, the local layers will communicate with the remote layers and delegate the decision there, only if there is limited confidence in the local decision. We propose a novel way to evaluate this local *confidence*, based on Bayesian methods using dropout during the forward pass. This is in contrast with the standard dropout methods for regularization [6], or the entropy-based uncertainty of the original DDNN paper [5].

(C.3) Using real data we demonstrate that, in many different scenarios, the distributed architecture is able to resolve up to 80% of decisions locally, while the uncertainty measure is able to pick out the correct remaining decisions that would benefit from a forward pass through the additional remote layers; the layer distribution and offload mechanisms, in conjunction, can always achieve this large overhead/latency reduction with a minimal objective degradation, and sometimes even improve the objective, compared to a fully centralized architecture with all layers involved in all decisions.

(C.4) We investigate the impact on these achieved distribution tradeoffs of (i) various allocation objectives, (ii) correlation patterns between the parallel traffic demands the architecture resolves, (iii) problem size and architecture size.

To our best knowledge, this is the first work to propose a distributed DNN architecture for slice resource allocation in the context of 5G wireless networks. The paper is structured as follows. Section II setups the problem and presents the problem objectives, while Section III presents our proposed DDNN architecture that solves the problem. In Section IV, we explain how to jointly train all exits of the DDNN, and how to operate it during runtime. Section V presents results and key insights on the DDNN performance. Section VI, and VII discusses related and future work; and Section VIII concludes the paper.

II. Resource Scaling using DNNs

In this section, we revisit how DNNs can be used for accurate and "safe" slice resource allocation (e.g., as initially proposed in [3]).

A. Data-Driven Resource Scaling

We consider a set of $\mathcal{M}$ network elements/functions, each of which requires some resources allocated, adapted to its traffic demand. These functions could belong to the same slice (e.g., different VNFs) or to different slices. We have past traffic samples d_t^i (scalar value) for a given network element $i \in \mathcal{M}$ at time interval t.[2] More concretely, we have:

- *Input*: The vector of N past samples for BS i, indicated as $\mathbf{d}_{t,N}^i = \{d_{t-N}^i, \ldots, d_{t-1}^i\}$. Notice that although we use N in the subscript, N does not vary; it simply expresses the fact that our input is of size N (with N integer).
- *Output*: A function approximator $\mathcal{DNN}(\mathbf{d}_{t,N}^i; \boldsymbol{\theta})$, which is parameterized by $\boldsymbol{\theta}$. The goal is to train this function to "predict" an allocation of resources $\hat{y}_t$ that "matches" the real demand d_t^i that will arrive, where the goodness of this match depends on the chosen optimization objective.

B. Resource Scaling Objectives

Traffic Forecasting. Let us first assume the standard regression goal, namely, given past traffic samples $\mathbf{d}_{t,N} = \{d_{t-N}, \ldots, d_{t-1}\}$ for some network element or slice (we drop here the superscript i for simplicity), to predict the next traffic sample d_t as accurately as possible. A DNN can be trained for this goal [2], [7], using a standard least squares objective:

$$f(\hat{y}_t, d_t) = (\hat{y}_t - d_t)^2 \tag{1}$$

Resource Scaling. We assume here that the inputs $\mathbf{d_{t,N}}$ of the DNN are the same as above. A key difference, is that we do not just want to predict the true traffic level d_t^i, but rather to provide an amount of resources $\hat{y}_t$ that can "satisfy" this demand (maybe by safely exceeding it with some margin). As a result, the objective might be asymmetric, with the cost of *underprovisioning* depending on the Service Level Agreement (SLA) with the slice tenant, and the cost of *overprovisioning*

[2]In this work we focus on Base Station (BS) traffic intensity, as we use such real datasets in our validation, but the methodology is generally applicable to any type of traffic demands over time (e.g., CPU, memory, queue sizes).

TABLE I: Main Notation

$\mathcal{M}$	Set of Base Stations, of cardinality M
H, W	Image height and width, $M = H \times W$
d_i^t	Demand of BS i at time t, $\in \mathbb{R}$
$\mathbf{d}_{t,N}^i$	Demand of BS i from $t-N$ to $t-1$, $\in \mathbb{R}^N$
$\mathbf{d}_t$	Demand of M BSs at time t, $\in \mathbb{R}^M$
$\mathbf{d}_{t,N}$	Demand of M BSs at $t-1$ to $t-N$, $\in \mathbb{R}^{M \times N}$
$\mathbf{D}_t$	$\mathbf{d}_t$ placed as an image, $\in \mathbb{R}^{W \times H}$
$\mathbf{D}_{t,N}$	$\mathbf{d}_{t,N}$ placed as an image, $\in \mathbb{R}^{W \times H \times N}$
$\boldsymbol{\theta}_i$	DNN layer parameters of layer i
$\hat{y}$	DNN output for $\mathbf{D}_t$, given $\boldsymbol{\theta}$
f	loss function

dependent on the opportunity cost of not allocating these extra resources to another slice. While this problem is a resource allocation problem (a *control* problem, in essence) it can be treated with the same DNN architecture, by simply picking the objective differently. We denote the following events:

- with "u": $\hat{y}_t < d_t$, (*underprovisioning*),
- with "o": $\hat{y}_t \geq d_t$ (*overprovisioning*).

Then, our allocation of resources $\hat{y}_t$ incurs some cost of the following form:

$$f(\hat{y}_t, d_t) = \mathcal{I}_u \cdot g_u(\hat{y}_t - d_t) + \mathcal{I}_o \cdot g_o(\hat{y}_t - d_t), \quad (2)$$

where $\mathcal{I}_{u/o}$ is the indicator function for event u or o.

For example, the authors in [3] minimize the following objective:

$$f(\hat{y}_t, d_t) = \mathcal{I}_u c_u + \mathcal{I}_o c_o (\hat{y}_t - d_t). \quad (3)$$

That is, a constant c_u penalty is assumed for *any* SLA violation, while the overprovisioning cost increases linearly (c_o could denote, e.g., the constant revenue loss per unused resource). Other objectives could capture different SLAs, e.g.,

$$f(\hat{y}_t, d_t) = \mathcal{I}_u \left(c_u + c_u^{'} (\hat{y}_t - d_t)^2 \right) + \mathcal{I}_o c_o (\hat{y}_t - d_t), \quad (4)$$

suggests an SLA that quadratically penalizes underprovision (e.g., because it leads to non-linear congestion and related performance degradation), and $c_u^{'}$ its importance. In the validation section, we will consider both these two objectives.

III. Distributing the DNN

In this section, we apply this generic resource scaling task to a beyond 5G network setup. Using a recent centralized DNN architecture for this task [3], as our starting point, we propose a distributed architecture that runs a (thinner) subset of this DNN at an edge cloud (e.g., MEC), to improve decision latency, and communicates with additional layers in a remote cloud (that can potentially provide increased performance) [8].

A. 5G Network Setup

Without loss of generality, in the remainder of the paper we will assume the problem setup depicted in Fig. 1. Specifically, we assume a set of BSs $\mathcal{M}$; each one of these M BSs serves a subset of users generating a demand (per BS), that is captured by some monitored quantity d_t^i, which is random and possibly non-stationary. A data-driven approach is used to allocate some matching resources for each BS, as described in Section II (e.g., CPU resources for baseband processing in a CRAN BBU cloud [9], or MEC CPU resources for computation offloading for associated users).

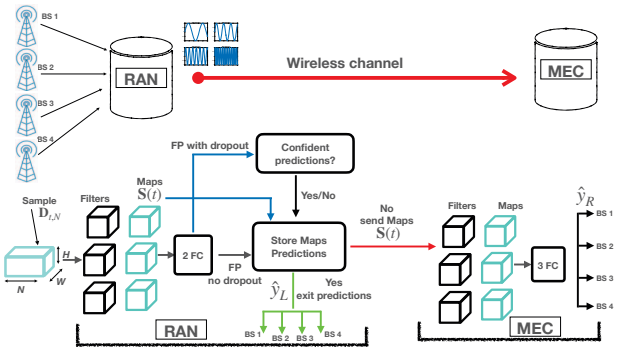


Fig. 1: Above: RAN (local cloud) collects the BS signals, and also wants to perform resource scaling with a local DNN; it might need to send some data to MEC (remote cloud), where there are additional NN layers. Below: a 3D sample arrives, is then forward-passed (FP) from the local NN *without* a dropout, and the maps along with the predictions for the 4 BSs are stored; then the same sample is forward-passed B times *with* dropout at the local NN; a confidence mechanism decides if the sample will be inferred locally or sent to the MEC

B. Edge Layer and Local Exit

We base our edge architecture on 3D-CNNs, as they have been found convenient to exploit correlations between BSs (this will require additional and appropriate preprocessing; we describe this in Section IV-A).

More specifically, at the edge (local DNN) we have a 3D-CNN ($\boldsymbol{\theta}_{CNN_L}$) with $F_L = 32$ filters (stands for "filters at local", each with kernel size ($3 \times 3 \times 3$), and two additional Fully Connected (FC) layers:

- FC1 (with ReLU nonlinearity): $N \cdot M \cdot F_L \rightarrow 2 \cdot M$,
- FC2 (linear): $2 \cdot M \rightarrow M$

(Thus, having M outputs, as many as the number of BSs for which we wish to perform resource scaling for.) FC1 increases the expressiveness of the local layers, and FC2 layer serves as the *local exit point* of the architecture - see Fig. 1.

Definition 1 (Local Exit Prediction)**.** *Given model parameters $\boldsymbol{\theta}_{CNN_L}$ for the 3D-CNN, and $\boldsymbol{\theta}_{1,2}$ for the FC1 and FC2, there is a function $\mathcal{DNN}^L(\cdot)$, which receives as an input $\mathbf{D}_{t,N}$ and returns a signal $\in \mathbb{R}^M$*

$$\hat{y}_t^L = \mathcal{DNN}^L(\mathbf{D}_{t,N}) \quad (5)$$

we call the left handside of (5) local exit prediction.

This local exit point attempts to make a decision immediately, and will be *trained jointly with the regular decision point at the core cloud* [5], [10].

Definition 2 (Local NN Maps)**.** *Given model parameters $\boldsymbol{\theta}_{CNN_L}$ for the 3D-CNN at the local NN, there is a function $\mathcal{G}(\cdot)$, which receives as an input $\mathbf{D}_{t,N}$ and returns a signal*

$$\mathbf{S}_t = \mathcal{H}(\mathbf{D}_{t,N}) \quad (6)$$

where $\mathbf{S}_t \in \mathbb{R}^{N \times M \times F_L}$; we call this tensor "local maps".

For a visual exposition of the above definitions, we refer the reader to Fig. 1.

Confidence Mechanism and Local Exit. Since our goal is to strike balance between high accuracy and high number of locally resolved samples, we design a simple decision mechanism that quantifies the "quality" of the resource forecast at the edge. The role of this mechanism is twofold: (a) detect the "hard cases", i.e., samples for which further processing and extra layers (at the remote cloud) would be of additional value, and (b) given a $\{$Yes, No$\}$ confidence signal to either exit the predictions of the local layers, or transmit the exported maps $\mathbf{S}_t$ of the local 3D-CNN to the remote layers over the network. The technical details of this mechanism are presented in the next section, where we will be particularly interested in the runtime operation of the DDNN.

C. Cloud Layers and Remote Exit

The maps $\mathbf{S_t}$ first pass through another 3D-CNN layer (denoted as $\boldsymbol{\theta}_{\boldsymbol{CNN}_R}$) that consists of $F_R = 16$ filters with kernel sizes ($5 \times 5 \times 5$), and then three FC ones; these are:

- FC3 (with ReLU nonlinearity): $N \cdot M \cdot F_R \rightarrow 8 \cdot M$,
- FC4 (with ReLU nonlinearity): $8 \cdot M \rightarrow 4 \cdot M$,
- FC5 (linear): $4 \cdot M \rightarrow M$,

The FC layers bring additional learning capabilities to the remote NN; it is important to stress, that once the sample arrives at the remote layers, the remote exit has no further actions to decide and must always return a set of predictions.

Definition 3 (Remote Exit Prediction)**.** *Given model parameters for the 3D-CNN at the remote NN $\boldsymbol{\theta}_{CNN_R}$, and $\boldsymbol{\theta}_{\mathbf{3,4,5}}$ for FC3, FC4, FC5; there is a function $\mathcal{DNN}^R(\cdot)$, which receives as input the local maps $\mathbf{S}_t$ and returns a signal $\in \mathbb{R}^M$*

$$\hat{y}_t^R = \mathcal{DNN}^R(\mathbf{S}_t) \tag{7}$$

we call the left handside of (7) remote exit prediction.

Note that the edge and remote cloud architectures are a function of the traffic image size (number of BSs M) we will construct later.

IV. DDNN: Training and Runtime

While training and operating a centralized DNN is relatively straightforward, doing the same for a distributed DNN has some key differences.

Training: the local and the remote layers must be *jointly* trained to achieve an intricate tradeoff: (i) the local subnet/exit must become powerful enough to be able to correctly resolve some decisions locally; (ii) the local layers must still act as high level feature extractors, so that remote additional layers can offer true added value for decisions not resolved locally; (iii) the overall performance (that also hinges on the local confidence mechanism) should be close to the centralized DNN one.

Runtime (or Inference time): Unlike a centralized DNN that inputs a past sample $\mathbf{D}_{t,N}$, does a forward pass through all layers, and outputs a proposed allocation for each of the M elements, a forward pass for a DDNN is more complicated. The local subnet produces both a local decision (through the FC layers branch) and a set of features (directly through the CNN branch) to potentially be further processed (by the remote layers). A confidence mechanism will decide which of the two branches to keep per sample.

In this section, we describe all these required steps to operate the proposed DDNN of Fig. 1 in training and runtime.

A. Data preparation

The resource demand for the resources, at the M BSs, is a multi-variate time-series. It has been observed that subsets of these time series may exhibit correlations, e.g., common diurnal patterns or weekday-weekend differences. More elaborate correlations might also exist between base stations that cover areas of similar "type". E.g., base stations covering train station or metro station users will typical exhibit strong peaks around commute times, while residential areas will have stronger peaks during early morning and in the evening, for example [3]. Nevertheless, these correlations are not known a priori; neither which time series (BS demand) i is correlated with another time series (BS demand) j, nor how strong this corellation is. While the 3D-CNN architecture is designed to exploit such correlation, an important data preparation step is needed to facilitate this. We note here that this step is generic to both a centralized and a distributed setup.

Milano Dataset. We will use a popular, publicly available dataset [11] that has been used in many related works [12], [13]. It consists of measurements for 10000 BS, and has the following entries per BS: "Internet Traffic", "Calls" and "SMS" for 21 days, which translates to 3024 values per entry; we will be interested only in the "Internet Traffic". For every BS m, we thus have a time series $\{d^m\}$ with 3024 demand values each: we remove the few BSs that have fewer samples than this. We also normalize each time series according to their min-max value, as is common to facilitate training.

Multi-variate Time Series to Image Conversion. The multivariate time series should now be organized in a 3D "box" of size $H \times W \times 3024$, so that there is correlation to be exploited later by 3D-CNN. The key idea is to ensure that nearby "pixels" correspond to traffic intensities of BSs that are correlated. For this task, we will use all our t data samples $\mathbf{d}_t$, each one of size $M \times 1$ (number of BSs). We use Shape Based Distance (SBD) to derive correlation values q_{ij} between BS i and j, $\mathbf{Q} \in \mathbb{R}^{M \times M}$. Each BS must be placed in one of the $H \times W$ locations/pixels of the $\mathbf{D}_t$ image. Let x_i denote the location of BS i in the 2D grid; we solve the problem and place the *whole* time series of BS i on the coordinate x_i.

Each BS m must be assigned a coordinate in the image $x_m = (a, b)$, with $a \in \{1, \ldots, H\}$, $b \in \{1, \ldots, W\}$ and $m \in \mathcal{M}$. Using a similar approach to [3], we first find the placement onto the continuous 2D grid by solving the following problem over the variable $\mathbf{p}$:

$$\underset{p_1,\ldots,p_M}{\text{minimize}} \sum_{i<j} (||p_i - p_j|| - q_{ij})^2 \tag{8}$$

therefore, if coordinates of BSs i, j are far with respect to their q_{ij}, then $||p_i - p_j||$ should also be large, thus minimizing

the objective function. Having the 2D continuous coordinates $\mathbf{p}$, we can formulate and solve the linear sum assignment problem -which decides which BS goes to which of our M available coordinates- using the Hungarian algorithm in polynomial time [14], [15], [16], [17].

B. Joint Training of Local and Remote Exits

Since our DNN has two exit points, we will formulate our objective function to be the weighted sum of two losses.

Definition 4 (Local and Remote Exit Losses)**.** *Given an input sample $\mathbf{D}_{t,N}$, a fixed set of parameters $\boldsymbol{\theta}$ (describing the whole NN), the local and remote exit predictions (see Defs. 1, 3)) $\hat{y}^L$, and $\hat{y}^R$, and a true demand signal $\mathbf{D}_t$, the losses incurred at both exits will be denoted as:*

$$\textit{Local Exit Loss:} \quad f(\hat{y}_t^L, \mathbf{D}_t) \tag{9}$$

$$\textit{Remote Exit Loss:} \quad f(\hat{y}_t^R, \mathbf{D}_t) \tag{10}$$

where $f(\cdot)$ expresses a notion of distance between the true demand image and the predicted one, as in (3) and (4).

The proposed DNN architecture then attempts to minimize the following objective over the model parameters $\boldsymbol{\theta}$ of all layers as we described in Sections III-B and III-C:

$$\sum_{k=1}^{K} \underbrace{w_L \cdot f(\hat{y}_k^L, \mathbf{D}_k)}_{\text{local exit loss}} + \underbrace{w_R \cdot f(\hat{y}_k^R, \mathbf{D}_k)}_{\text{remote exit loss}} \tag{11}$$

where k iterates through the samples of the training dataset. In other words, we are interested in minimizing *both* a local under/over-provisioning cost as in (2), related to the output at the local exit point $\hat{y}_t^L$ (which trains only $\boldsymbol{\theta}_L$, the parameters of the local DNN) at the edge, and a similar cost at the final exit point of the DDNN $\hat{y}_t^R$, i.e., the "normal" output of a DNN (which trains both $\boldsymbol{\theta}_L$ and $\boldsymbol{\theta}_R$).

Importantly, the weights $w_L \in [0, 1]$ and $w_R = 1 - w_L$ decide the impact of the local and remote exits on the overall (joint) performance loss of the DDNN. For example, choosing $w_L = 0$ on (11), would train the DDNN as a regular DNN with two 3D-CNN layers three FC layers, and a single output, as the centralized architecture in [3]. On the other hand, a w_L closer to 1 would try to optimize the performance at the local exit point (e.g., to maximize the amount of decisions that can be confidently taken at the edge), while putting less importance in the *additional* performance benefits offered by the extra layers at the remote cloud. As we will see later in Section V, the choice of (w_L, w_R) is crucial to strike a good tradeoff between the three (potentially conflicting) goals set in the beginning of this section: local performance, local feature extraction, remote/global performance. In the majority of cases we tested, minimum $w_L \geq 0.5$, (or $w_L \geq w_R$) seems necessary to achieve any such tradeoff, which might be reasonable since the local subnet is more shallow.

Remark: It is important to stress here that this joint training is necessary, due to the coupling of the local and remote layers via the top branch of Fig. 1: the local layers provide some feature extraction for the remote ones, in addition to producing a local exit (bottom branch). Hence, the distributed architecture is not "simply" a local DNN and a remote DNN that could be separately trained[3]. What is more, as this training is performed offline, the actual training could be performed centrally (in fact, this is what we do in this work, for simplicity), while the architecture at runtime operates in distributed mode. Distributed training is an interesting topic that is orthogonal to this work (see e.g. [19], [20]).

C. Local Confidence for DDNN Runtime

A crucial component of the distributed architecture, is the confidence mechanism, briefly discussed in Section III-B, to decide whether the local decision would be "good enough" or the forward pass should continue at the remote layers. Ideally, this mechanism should:

1) resolve locally samples whose remote exit would be similar to the local exit (no added value offered by the extra layers);
2) continue forward pass to remote layers, if the remote exit cost would be reasonably lower than the local (could improve the overall performance).

A key obstacle though is that *the remote exit/decision cannot be known beforehand, at the edge, unless the entire forward pass is finished.* Hence, we essentially require an "unsupervised" mechanism to establish how *confident* we are in the local exit decision.

In [5], the entropy of the local decision is used as a confidence metric. This is a natural metric for a classification problem (as the one in [5]), since high entropy implies that the DNN is not "sure" about which class is the correct, and additional layers could help refine its label decision. Unfortunately, an entropy metric is not applicable to a problem like ours, which is, in essence, a regression problem. To this end, we propose a different, Bayesian confidence metric, based on random dropouts *applied to the local forward pass* [21], [22] (we stress that this is a different dropout mechanism than the one we use during training which serves as a regularization [6]).

Definition 5 (Uncertainty)**.** *We force the local exit to infer a single input sample, say B times. The output we receive is M arrays (one for each BS) $\in \mathbb{R}^B$ each. For every BSs $m \in \mathcal{M}$, we compute the standard deviation of BS m, and then take the its maximum value among the M BSs.*

$$\mathcal{U} = max_m\{std_m\} \tag{12}$$

This metric serves as a worst case estimate of how much perturbations have affected the local decisions. In practice, about 10 iterations with dropout suffice to provide reasonable uncertainty estimates; in our experiments we iterated $B = 10$ and used a drop-out probability $p = 0.4$ on the last linear layer of the local NN, i.e., FC2.

Definition 6 (Confidence Mechanism)**.** *Given (a) the measured uncertainty $\mathcal{U}$, (b) a confidence threshold T_{conf} which is*

[3]This would require, at runtime, to send raw input data across the network, rather than potentially compressed features (additional/adaptive compression could further be applied using techniques like [18]).

a design parameter of the DDNN, the confidence mechanism compares the two, and if $\mathcal{U} < T_{conf}$ all M predictions are carried out locally; otherwise, the exported maps $\mathbf{S}_t$ are sent to the remote layers, so that the M predictions take place there.

Given we have the full DNN model $\boldsymbol{\theta}$ parameters, during inference time the actions that need to be taken are:

- do inference at the local exit using the full model once, and store the prediction $\hat{y}^L, \mathbf{S}$;
- forward-pass the input sample B times using dropout p, and compute $\mathcal{U}$;
- compare $\mathcal{U}$ with T_{conf}, and act according to Def. 6.

Discussion. The algorithm considers the metric of uncertainty, in order to characterize the "quality" of the local prediction $\hat{y}_L$. However, when we say "quality", we refer to something specific: the objective function value of local NN $f(\mathbf{D}_t, \hat{y}_t^L)$ with respect to $f(\mathbf{D}_t, \hat{y}_t^R)$. Ideally, we would want to have access on an oracle that tells us whether it is worth it to offload on the remote NN or not; specifically, whether

$$E_t^L = f(\mathbf{D}_t, \hat{y}_t^L) - f(\mathbf{D}_t, \hat{y}_t^R) > 0 \tag{13}$$

is positive or negative. Essentially, given the DNN parameters $\boldsymbol{\theta}$, one could even consider the case of having a hard constraint on the percentage or samples that are served locally, say $> 70\%$. In that case, the optimal solution would be to pick the 30% samples with the highest $\{E_t^L\}$, and for those samples perform inference at the remote cloud. An interesting avenue of research would be the framework of Constrained Online Convex Optimization [13], [23] where in some sense, a data-driven algorithm would try to track the characteristics of the samples for which $\{E_t^L\}$ is high. In a sense, here, instead of considering such approaches, we argue that the uncertainty is indeed a good *proxy metric* that detects high $\{E_t^L\}$ samples as will see in Section V. Interestingly, uncertainty behaves the best in the cases of models whose joint training objective had $w_R < w_L$.

V. Performance

Having presented the proposed DDNN architecture, as well as its training and runtime operation, in this section we will go ahead and investigate the achieved trade-offs and discussed benefits of such an architecture, considering a number of problem dimensions such as: (i) problem objective, (ii) traffic demand correlation, (iii) problem size (number of BSs handled jointly). We will also offer a number of accompanying plots that attempt to shed some light as to when/why these better trade-offs (compared to a centralized architecture) are achieved.

A. Validation Setup

First of all, for training, we are using for every case that we will present a variety of weight pairs (w_L, w_R); during training, we use the Adam Optimizer with a learning rate of $5 \cdot 10^{-4}$. Regarding our scenarios we pick a number of M BSs time series out of the 10K available ones. Unless otherwise stated, $M = 16$. We will elaborate in Section V-E two different ways of picking these, and their impact on various performance tradeoffs. (For more details on the data preprocessing, we refer the reader to Section IV-A.)

As mentioned earlier, the main goal of the proposed distributed DNN is to perform almost as well (in terms of the incurred allocation cost) as a fully centralized one (of similar size) that runs all layers at the remote cloud. As explained in Section IV, such an architecture can be emulated by choosing training weights $(w_L, w_R) = (0, 1)$ and the confidence threshold to $T_{\text{conf}} = 0$. Essentially, it is an architecture that uses two 3D-CNNs, and FC3, FC4, FC5 -bypassing *always* FC1, FC2. The resulting trained DNN, which we will refer to as "Centralized DNN", then resembles the architecture of DeepCog [3]. This will serve as our baseline for comparison.[4].

B. Performance Metrics

Local and Remote Exit Cost. Following the Def. 4, and more specifically: (9) and (10), during service, *for every test sample*, regardless of whether prediction was carried out locally or remotely, we will keep track of *both* exit costs, as if all samples were exited from both. The exit costs will help us draw conclusions on the confidence mechanism and the overall DDNN performance. We will use two objective functions, namely: (3) and (4). The first one, is essentially the objective of [3], which penalizes underprovisioning with a constant: we will explore two cases, namely $c_u = 0.5$, and $c_u = 1.0$; the second objective has stricter SLA penalties for underprovision, namely an additional quadratic term; for the latter, the constant penalty term will be $c_u = 0.5$, and the quadratic weight $c'_u = 30$. In both case, the weight for overprovision is $c_o = 1.0$. We refer the reader to Section II-B for more details.

Error at Local Exit. Following the previous performance metric, we will be particularly interested in the quantity $E^L(t)$, (13), defined as the cost difference between the local and the remote exit.

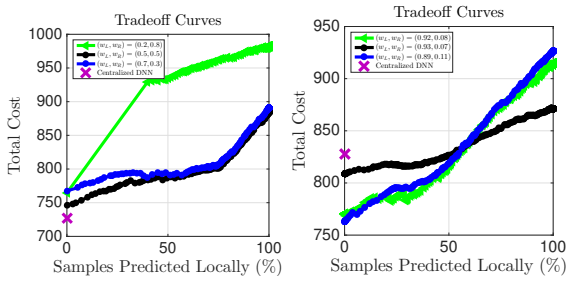
Distributed DNN Total Cost. During inference (runtime), given a specific T_{conf} the mechanism decides whether inference takes place, and accordingly, in the total cost we add terms:

$$C_{\text{DDNN}} = \sum_{k=1}^{K} \mathcal{I}_L(k) \cdot f(\mathbf{D_k}, \hat{y}_k^L) + (1 - \mathcal{I}_L(k)) \cdot f(\mathbf{D_k}, \hat{y}_t^R) \tag{14}$$

where $\mathcal{I}_L(k)$ is the event where the confidence mechanism decided to do the forecast locally.

Centralized DNN Total Cost. During runtime, all decisions of the centralized DNN are carried out remotely, thus its total

[4]We remind the reader that, while the architecture of the "Centralized DNN" baseline and the one in [3] are very similar, the actual objective here is to "predict" an allocation for each individual BS, rather than an aggregate demand of multiple BSs - potentially smoothing out some uncertainty due to the LLN law - as is the case in [3].



(a) Constant $c_u = 0.5$ penalty (b) Constant $c_u = 1.0$ penalty

Fig. 2: Operating points: (total cost, percentage of samples predicted at local exit) for increasing T_{conf}. Constant underprovision penalty

cost is:

$$C_{\text{centralized}} = \sum_{k=1}^{K} f(\mathbf{D_k}, \hat{y}_k^R) \tag{15}$$

Percentage of Local Decisions. Given a specific T_{conf}, it is computed as:

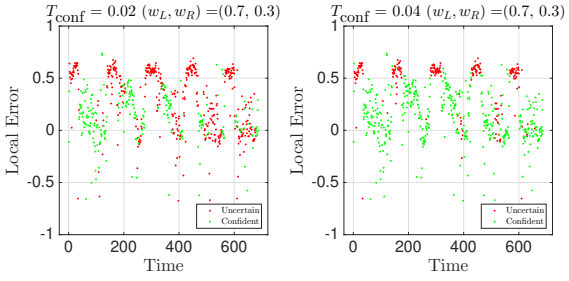
$$\sum_{k=1}^{K} \mathcal{I}_L(k)/K. \tag{16}$$

C. Resource Scaling Cost vs Communication Tradeoff

Constant $c_u = 0.5$ underprovision penalty. We use as cost function the constant underprovisioning (3) with $c_u = 0.5$ and linear overprovisioning with $c_o = 1$. We choose a range of confidence thresholds $T_{\text{conf}} \in [0, 0.5]$, and for every T_{conf} we iterate the test set; we measure the following tuple (total cost, locally resolved), as in (14) and (16). The set of operating points returns the "tradeoff curve" of some model (w_L, w_R). In Fig. 2(a), we plot the tradeoff curves for three models: (0.7, 0.3), (0.5, 0.5) and (0.2, 0.8). Finally, with magenta we plot a single operating point "X", which corresponds to the cost achieved by the centralized DNN, whose hardware is placed at the cloud; by construction, this resolves *all* samples remotely.

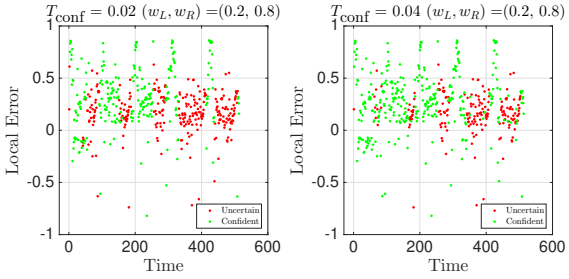
Observation 1: The two models with $w_L > w_R$ have found a "sweet spot"; as T_{conf} increases, the cost increases only by a 10% (compared to the Centralized case -magenta "X"), while resolving about 75% of samples locally. Thus, when the joint training is done properly, the uncertainty and confidence mechanism is clearly able to correlate the Bayesian uncertainty metric with true added value of the additional remote layers. The (0.2, 0.8) model does not work as harmonically with the confidence mechanism, as its cost performance exhibits a sudden jump, suggesting bad training of the exit as we will see shortly.

Moreover, in Fig. 3, we present the $E^L(t)$ as a function of time for two values of T_{conf}, namely 0.02, and 0.04 for the $(0.7, 0.3)$; we do the same in Fig. 4 for $(0.2, 0.8)$. The locally resolved (confident) samples are indicated with green, and the remote with red (uncertain). In Fig. 3, the more we increase T_{conf} the more samples will be designated as "confident" and exited locally. An interesting take-away from Fig. 3(b) is that the confidence mechanism sends to the remote layers *the*



(a) $T_{\text{conf}} = 0.02$ (b) $T_{\text{conf}} = 0.04$

Fig. 3: Error at local exit vs Time: Samples resolved: (a) green-local, (b) red-remote. Constant underprovision penalty $c_u = 0.5$; $(w_L, w_R) = (0.7, 0.3)$



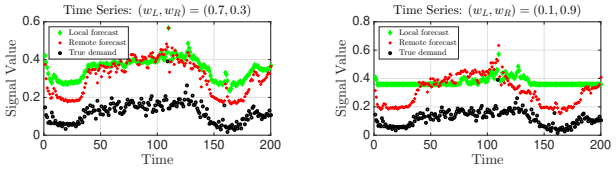
(a) $T_{\text{conf}} = 0.02$ (b) $T_{\text{conf}} = 0.04$

Fig. 4: Error at local exit vs Time: Samples resolved: (a) green-local, (b) red-remote. Constant underprovision penalty $c_u = 0.5$; $(w_L, w_R) = (0.2, 0.8)$

correct samples, i.e., the ones with really high E^L, which when we forecast remotely will improve the overall cost performance. In contrast however, Fig. 4, says a different story: in both T_{conf}, there are many values which are very confident but of *very high* E^L which are not detected by the confidence mechanism, a phenomenon which again hints poor choice of (w_L, w_R).

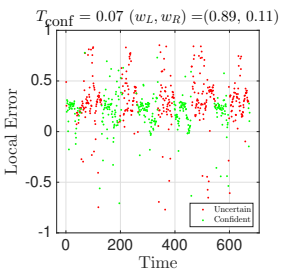
Observation 2: In Fig. 5, we see the true effect of the good and bad training; in both cases the remote predictions $\hat{y}^R$ are reasonable, however the local ones $\hat{y}^L$ differ significantly. In the (0.7, 0.3) case, $\hat{y}^L$ overprovisions more than the $\hat{y}^R$, however in the (0.1, 0.9), we see that the $\hat{y}^L$ has not really learned properly, as it stays constant for great periods of time. Essentially, this strikes the significance of (w_L, w_R); in good training cases, there is correlation between E^L and uncertainty $\mathcal{U}$.

Constant $c_u = 1.0$ underprovision penalty. We increase $c_u = 1$ by keeping the same objective function, i.e., (3); we will discuss results of similar nature as the previous ones, but

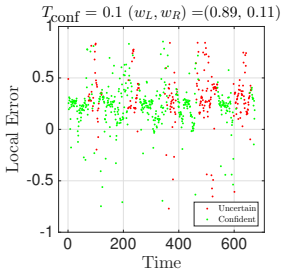


(a) $(w_L, w_R) = (0.7, 0.3)$ (b) $(w_L, w_R) = (0.1, 0.9)$

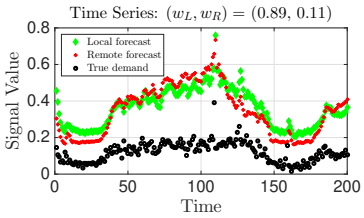
Fig. 5: Offline forward pass of all samples; green: local exit forecast, red: remote exit forecast, black: true demand. Constant underprovision penalty $c_u = 0.5$



(a) E^L vs Time: $T_{\text{conf}} = 0.07$



(b) E^L vs Time: $T_{\text{conf}} = 0.1$



(c) Resource Scaling for local (green) and remote (red) exits, along with true demand (black)

Fig. 6: $(w_L, w_R) = (0.89, 0.11)$ Constant underprovision penalty $c_u = 1$

we will stress qualitative differences between the two cases.

First of all, in Fig. 2(b), we see an interesting difference compared to $c_u = 0.5$: our distributed DNNs, that were trained with $w_L > w_R$ have achieved a cost that is lower than the centralized DNN (for which we have: $w_L, w_R = 0, 1$). Although this might seem surprising, there is rich literature surrounding this phenomenon [5], [10], [24]; it has been claimed that early exits serve as *additional strong source of regularization* for a DNN, and by using them. We can see that in Fig. 2(b), and more specifically for the case where $T_{\text{conf}} = 0$, i.e., when our distributed DNN resolves *all* samples remotely; there, the model $(w_L, w_R) = (0.89, 0.11)$ can achieve a cost which is 7% than the one of the centralized DNN which is trained using $(w_L, w_R) = (0.0, 1.0)$ We can increase confidence threshold up to a point where our distributed DNN has more or less the same cost as the centralized architecture. We observe that we are able to resolve 60% locally, and have the same cost as the centralized architecture: a clear benefit!

Observation 3: There are scenarios where not only can we resolve a significant percentage of decision locally, but we can even improve the overall cost as well, compared to an equally powerful but centralized DNN.

Moreover, the harmonic coexistence of the confidence mechanism alongside the DNN is evident from Figs 6(a) and 6(b). We can see that the mechanism captures samples that are of very high E^L, thus using the remote layers in order to improve in most cases, and resolves locally many samples which are of $E^L \approx 0$, or even lower (these samples are better forecast locally!)

Key take-away: To achieve such trade-offs, we have to carefully choose the local/remote weights at joint training—the performance of the DDNN is sensitive in those crucial parameters, and a more systematic study of the importance of these parameters is deferred for future work.

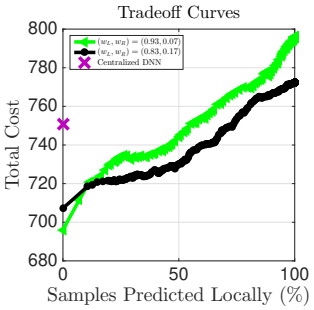
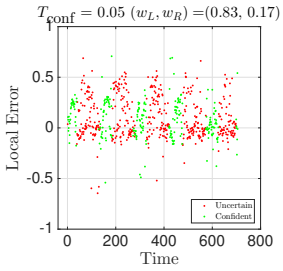
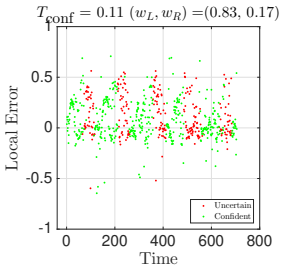


Fig. 7: Operating points: (total cost, percentage of samples predicted at local exit) for increasing T_{conf}. Underprovision penalty: quadratic $c'_u = 30$ plus constant $c_u = 0.5$



(a) $T_{\text{conf=0.05}}$



(b) $T_{\text{conf}} = 0.11$

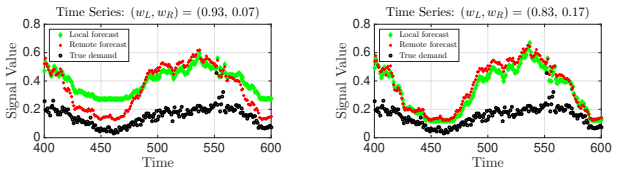
Fig. 8: (0.83, 0.17). Error at local exit vs Uncertainty: green (below threshold) - forecast local, red (above threshold) - forecast remotely. Underprovision penalty: quadratic $c'_u = 30$ plus constant $c_u = 0.5$

D. Quadratic plus Constant Underprovisioning

We study the same problem, but use a stricter (in the underprovision penalty sense) objective (4). Looking at Fig. 7, with $c_u = 0.5$ and $c'_u = 30$, we plot the tradeoff curves for two models (w_L, w_R) where the "avoid overthinking" phenomenon is again present. Both presented models, i.e., (0.93, 0.07) and (0.83, 0.17), heavily outperform the centralized DNN. The (0.83, 0.17) model has the *same cost with the centralized*, when it is able to resolve 75% of its samples locally. In Fig. 8, we see the following: regardless of T_{conf}, the remote layers take responsibility for the "hard" cases (many red samples with high E^L), and to make things better, the locally resolved samples are of relatively low E^L, showing that the confidence mechanism serves its purpose of classifying correctly hard and easy samples. Finally, from Fig. 9, we can see that the local exit, although will less layers, it is able to perform more or the less the same resource allocation -and sometime overprovision less than the remote.

Key take-away: The DDNN architecture paired with the confidence mechanism is robust to other objectives as well.

Finally, we present Table II which has the following information: it quantifies for the models we presented up to this point, the mean (over all BSs and over all timestamps) percentage of underprovisioning. In the table we only show w_L, (which also defines $w_R = 1 - w_L$), and with dash we refer to the centralized DNN (which resolves all samples



(a) (0.93, 0.07) (b) (0.83, 017)

Fig. 9: Resource Scaling for local (green) and remote (red) exits, along with true demand (black). Underprovision penalty: quadratic $c'_u = 30$ plus constant $c_u = 0.5$

TABLE II: Under-provisioning: all objectives

Objective	w_L	Local [%]	Remote [%]
$c_u = 1$	0	–	0.61
	0.89	0.83	0.61
	0.92	0.82	0.62
	0.93	0.67	0.68
$c'_u = 30$ and $c_u = 0.5$	0	–	1.51
	0.83	2.47	1.65
	0.93	2.04	1.88
$c_u = 0.5$	0	–	1.89
	0.7	2.21	2.62
	0.5	2.03	2.03

remotely). Let us focus on the first and third row (the constant underprovisioning penalty cases); both the remote and local exit underprovision less as we go from $c_u = 0.5 \rightarrow c_u = 1$; which is reasonable as the objective is stricter. Then a second comparison is to observe second and third rows, we clearly see that the quadratic plus constant $c_u = 0.5$ underprovisions much less in the remote exits, however the local ones, perform similarly, a fact which hints that the remote exit was better trained.

E. Larger Instances and Correlation

As explained earlier, the BS time series might exhibit some degree of correlation, that the architecture that jointly handles them tries to leverage. To this end, in this section we explicitly focus on the impact of such correlation both in the centralized and in the distributed operation. However, this step is performed *given* the set $\mathcal{M}$ of the BSs. Along this line of thought, we compare two scenarios: (a) M BSs chosen randomly out of the set of the 10K available ones; (b) M BSs that exhibit a higher correlation than random ones (according the the SBD metric -see Section IV-A). In Fig. 10, we plot the tradeoff curves for $M = 16, 25, 36$, thus 6 curves in total.

Observation 5: For M fixed, and comparing correlated to randomly picked BSs: the huge performance gap, indicates that there is a lot of benefit in using BSs that are correlated -before even placing them in the 2D grid- for the distributed DNN architecture that we propose.

Regarding the cost performance between the same method of choosing BSs: we can clearly see that the three curves up top, are the ones where the BSs were chosen randomly (worst performance). In addition, the performance differences *among them* are not that significant. The trend is slightly different for the correlated ones: we see the performance differences larger; this hints that although the correlation

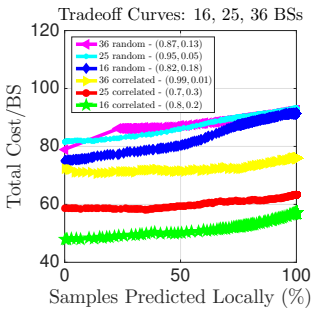


Fig. 10: Comparison of tradeoff curves for 16 and 25 BSs.

helped improve the cost (compared to having random BSs), maybe even larger architectures are needed. Since we try to predict per BSs, the task becomes increasingly challenging.

VI. Related Work

Data-driven methods and DNNs have been used recently for both traffic prediction as well as for slice resource allocation. The problem has been viewed by a variety of angles; it has been faced through pure DNN based approaches [3], [4], by stochastic control methods (Lyapunov) methods [25]. Additionally, slicing has been a fruitful field of application for online convex optimization [12], [26], [13]. Moreover, reinforcement learning [27], [28] for resource orchestration has been recently applied on the problem On the other hand, DNNs [10], [5], [24], [8], [18] with early exit(s), is a recent and promising research avenue with many under-exploited applications. Moreover, a key quantity is the amount of information transmitted to the remote layers. Recent works that have proposed systematic ways of compressing the information of the transmitted (towards the cloud) maps, e.g., by dynamically adjusting the sent features depending on network conditions [18], [29], [30].

Here, we also stress the points we differentiate with closely related works. Compared to [3], the novelties are: (i) distributed inference, (ii) different objectives investigated, as well as a per BS (rather than aggregate) prediction, (iii) significant performance benefits, both in terms of overhead/latency and often in terms of allocation cost, and (iv) a confidence mechanism that estimates the quality of the local exits. With respect to [5], we differentiate at the following: (i) a different DNN architecture, (ii) different ML task task, namely multivariate time series prediction rather than image classification, (iii) an in depth-investigation of the interplay between joint training weights, uncertainty mechanism, and time-series correlation and its impact on objective cost, actual time-series prediction and percentage of samples resolved locally.

VII. Future Work

Local or Remote Decision. The uncertainty is just *a* way of distinguishing what samples should be forecast locally; however, other data-driven optimization methods whose goal is to learn the characteristics of the E_L signal *by receiving feedback from the environment* are interesting alternatives.

Such an algorithm could for example dynamically tune the T_{conf} instead of using it as a tuning parameter, as we did.

Objective weights. A crucial step of the algorithm that we presented is the selection of (w_L, w_R); note that however, one may wish to split an NN in *more than two* locations, a setup which is even more challenging than ours. This motivates as an interesting avenue of research: the design of a scheduler which can automatically tune these weights *during* training.

VIII. Conclusions

We have developed a very promising Distributed DNN framework for the task of resource scaling for 5G and beyond networks. The proposed architecture consists of two exits: a local and a remote, which have to be trained jointly by assigning a weight to each. 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 DNN with a Bayesian based confidence mechanism which aims to detect the samples of high uncertainty and send them to the remote cloud for further processing. Comparing our algorithm with its equivalent centralized one, we observed that it is possible to resolve $\approx 60\%$ of the incoming samples locally, with almost zero 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-IEEE Conference on Computer Communications*, pp. 1–9, IEEE, 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-IEEE Conference on Computer Communications*, pp. 280–288, IEEE, 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-IEEE Conference on Computer Communications*, pp. 794–803, IEEE, 2020.

[5] 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)*, pp. 328–339, IEEE, 2017.

[6] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[7] C. Zhang and P. Patras, "Long-term mobile traffic forecasting using deep spatio-temporal neural networks," in *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pp. 231–240, 2018.

[8] 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.

[9] R. Schmidt and N. Nikaein, "Ran engine: Service-oriented ran through containerized micro-services," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 469–481, 2021.

[10] 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)*, pp. 2464–2469, IEEE, 2016.

[11] Telecom Italia, "Milano Grid." https://doi.org/10.7910/DVN/QJWLFU, 2015.

[12] N. Liakopoulos, G. Paschos, and T. Spyropoulos, "Robust user association for ultra dense networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 2690–2698, IEEE, 2018.

[13] N. Liakopoulos, A. Destounis, G. Paschos, T. Spyropoulos, and P. Mertikopoulos, "Cautious regret minimization: Online optimization with long-term budget constraints," in *International Conference on Machine Learning*, pp. 3944–3952, PMLR, 2019.

[14] P. John and L. Gravano, "K-shape: Efficient and accurate clustering of time series," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), p. 1855–1870, Association for Computing Machinery, 2015.

[15] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pp. 14–27, 2017.

[16] I. Borg and P. J. Groenen, *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

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

[18] 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 *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pp. 50–56, 2018.

[19] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 2018.

[20] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[21] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, pp. 1050–1059, PMLR, 2016.

[22] L. Zhu and N. Laptev, "Deep and confident prediction for time series at uber," in *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pp. 103–110, IEEE, 2017.

[23] H. Yu, M. J. Neely, and X. Wei, "Online convex optimization with stochastic constraints," *arXiv preprint arXiv:1708.03741*, 2017.

[24] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," in *International Conference on Machine Learning*, pp. 3301–3310, PMLR, 2019.

[25] A. T. Z. Kasgari and W. Saad, "Stochastic optimization and control framework for 5g network slicing with effective isolation," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, IEEE, 2018.

[26] N. Liakopoulos, G. Paschos, and T. Spyropoulos, "No regret in cloud resources reservation with violation guarantees," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 1747–1755, IEEE, 2019.

[27] Q. Liu, T. Han, and E. Moges, "Edgeslice: Slicing wireless edge computing network with decentralized deep reinforcement learning," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 234–244, IEEE, 2020.

[28] A. Okic, L. Zanzi, V. Sciancalepore, A. Redondi, and X. Costa-Pérez, "π-road: a learn-as-you-go framework for on-demand emergency slices in v2x scenarios," *arXiv preprint arXiv:2012.06208*, 2020.

[29] 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 *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pp. 1–15, 2020.

[30] 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*, pp. 1423–1431, IEEE, 2019.