Distributed LSTM-based Slice Resource Allocation for Beyond 5G Networks

Ali Ehsanian
EURECOM
Sophia-Antipolis, France
ali.ehsanian@eurecom.fr

Thrasyvoulos Spyropoulos
Technical University of Crete
Chania, Greece
spyropoulos@tuc.gr

***Abstract*—End-to-end network slicing is a new concept for 5G+ networks, dividing the network into slices dedicated to different types of services and customized for their tasks. A key task, in this context, is satisfying service level agreements (SLA) by forecasting how many resources to allocate to each slice. The increasing complexity of the problem setup, due to service, traffic, SLA, and network algorithm diversity, makes resource allocation a daunting task for traditional (model-based) methods. Hence, data-driven methods have recently been explored. Although such methods excel at the application level (e.g., for image classification), applying them to wireless resource allocation is challenging. Not only are the required latencies significantly lower (e.g., for resource block allocation per OFDM frame), but also the cost of transferring raw data across the network to centrally process it with a heavy-duty Deep Neural Network (DNN) can be prohibitive. For this reason, Distributed DNN (DDNN) architectures have been considered, where a subset of DNN layers is executed at the edge (in the 5G network), to improve speed and communication overhead. If it is deemed that a "good enough" allocation has produced locally, the additional latency and communication are avoided; if not, intermediate features produced at the edge are sent through additional DNN layers (in a central cloud). In this paper, we propose a distributed DNN architecture for this task based on LSTM, which excels at forecasting demands with long-term dependencies, aiming to avoid under-provisioning and minimize over-provisioning. We investigate (i) joint training (offline) of the local and remote layers, and (ii) optimizing the (online) decision mechanism for offloading samples either locally or remotely. Using a real dataset, we demonstrate that our architecture resolves nearly *50%* of decisions at the edge with no additional SLA penalty compared to centralized models.**

***Index Terms*—Network Slicing, Resource Allocation, Distributed Deep Neural Network, LSTM Model, 5G Networks**

I. Introduction

5G networks are expected to support a large number of tenants simultaneously with different Quality of Service (QoS) requirements and services with different service level agreements (SLA). Satisfying these expectations makes optimal resource allocation a key to the success of 5G networks. However, traditional optimization approaches for this task are often impractical, paving the way for data-driven approaches proposed in recent AI literature [1]–[4]. Specifically, data-driven algorithms are expected to execute at various network locations, leveraging the computational capabilities at MEC, RAN, and the core in modern cellular networks. Slice resource allocation using deep learning has been a popular recent research direction in the context of 5G networks [5]–[9]. Also, reinforcement learning has recently been applied to the problem of resource orchestration [6], [8], [10].

Nevertheless, optimization objectives in this context are based on the SLAs and are often asymmetric: i.e., the cost of under-provisioning (penalty paid to the tenant) might differ from the cost of over-provisioning (opportunity cost of wasted resources). To this end, the seminal works of [11], [12] have demonstrated how to (i) predict the allocation that properly balances over-/under-provisioning penalties for each slice and (ii) leverage correlations between resource and slice demands using a proposed centralized CNN-based (Convolutional Neural Network) architecture.

However, running a centralized heavy duty DNN faces two key challenges in (5G+) wireless architectures: (i) a number of network optimization tasks, especially those at the RAN, have stringent latencies compared to UE-level applications often offloaded to the cloud. (ii) the overhead of sending raw data over potentially congested edge/wireless links can often be a major hurdle in the application of such solutions. Implementing a decentralized algorithm might offer a potential solution [13], [14], [15]. A promising solution is that of [13], where the authors propose a Distributed Deep Neural Network that attempts to strike a trade-off between "good enough" local image classification for most samples and "stronger" cloud-based classification for only a few. Motivated by this work, we have recently investigated in [16] a distributed version of the architecture proposed in [11], where a 3D-CNN is used for slice resource allocation.

In this paper, we attempt to further show the generality of this methodology by investigating how to distribute a more sophisticated DNN architecture for the same task, based on LSTM (Long Short Term Memory) units. Specifically, the main contributions of this work are the following:

- We propose a distributed LSTM architecture for the problem of balancing under-/over-provisioning of resources to different slices and investigate how the methodology of DDNNs can be applied to such larger, more sophisticated architectures (compared to that of [11]). Such an architecture contains a few LSTM units and a "local exit" (i.e. a prediction layer) at the edge, and a larger number of units and "remote exit" at the central cloud.

- (*Offline Optimization*) We demonstrate the impact of properly tuning the joint training hyperparameters of local and remote "exits" (i.e., predicted allocations and related SLA costs) to achieve a good balance between: (i) making the local layers powerful enough to correctly make a large enough number of allocation decisions, while (ii) producing useful features that the remote layers could leverage, when improved allocation decisions are deemed necessary.
- (*Online Optimization*) We propose a mechanism that measures the confidence in the local exit, and predicts whether the remote exit (that requires additional latency and communication) would improve the SLA costs enough to justify the extra overhead (i.e., a type of unsupervised learning).

The rest of this paper is structured as follows: Section II covers the problem setup, while Section III details our LSTM-based DDNN architecture. In Section IV, we delve into offline joint training and online offloading. Section V validates the architecture with real traffic data. Section VI outlines future work and conclusion.

II. Problem Setup

A. Slice Resource Allocation with DNN

Assume we have a set of $\mathcal{K}$ network functions (e.g., VNFs) or network element slices (e.g., BSs). Each network function/element requires some resources which will be allocated according to its traffic demand (e.g., we might need to allocate some resource block at a given BSs). In the remainder, we will assume that a DNN-based architecture is used to *determine* the amount of resources allocated to each of the $\mathcal{K}$ network functions. We stress here that our goal is not to match these resources *exactly* to the future (currently unknown) demand of those VNFs, but to strike a trade-off between the given under- and over-provisioning costs.

We can consider the DNN as a black box and model it with an approximation function with some parameters that takes an input vector and gives the predicted value. Therefore we can write:

$$\hat{y}_t^i = \mathcal{F}(\mathbf{d}_{t,N}^i; \boldsymbol{\theta}), \tag{1}$$

where $\mathbf{d}_{t,N}^i$ is the input vector for the DNN. The input vector $\mathbf{d}_{t,N}^i = \{d_{t-N}^i, ..., d_{t-1}^i\}$ consists of the *N* past traffic samples of BS $i \in \mathcal{K}$ before the time *t*. *N* is the input vector size and is constant during the model training. $\mathcal{F}(;\boldsymbol{\theta})$ is the approximation function with $\boldsymbol{\theta}$ as parameters. The vector $\boldsymbol{\theta}$ represents the model parameters (i.e., weights of the DNN). $\hat{y}_t^i$ will be the allocated resource to the network element *i* at time *t*, which is forecasted by the DNN to balance under-/over-provisioning costs in relation to the (unknown) demand d_t^i at that time.

B. Resource Allocation Objective Functions

In a standard forecasting problem, one wants to predict the traffic value at time *t* using the past *N* traffic samples $\mathbf{d}_{t,N} = \{d_{t-N}, ..., d_{t-1}\}$. The goal is that the predicted value $\hat{y}_t$ to be as close as possible to the real traffic d_t. To achieve this, we can train a DNN with a least squares objective function.

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

An important difference in our work is that the cost of being under or over the true demand is not symmetric, unlike in Eq. (2). If the predicted traffic is less than the needed traffic, not enough resources will be allocated to the slice (under-provisioning, $\hat{y}_t < d_t$), which could violate the SLA with the slice tenants. If the predicted traffic is more than the needed traffic, then more resources than are needed will be allocated to the slice (over-provisioning, $\hat{y}_t > d_t$), which will waste some resources. Depending on the SLA violation cost (under-provision) and the "wasted" resources cost (over-provision), different objectives can be used that the DNN will try to optimize. Without loss of generality, we'll assume the following objective.

$$f(\hat{y}_t, d_t) = \begin{cases} c_1 \cdot (\hat{y}_t - d_t)^2 & \text{if } (\hat{y}_t - d_t) \leq 0 \\ c_2 \cdot (\hat{y}_t - d_t) & \text{if } (\hat{y}_t - d_t) > 0, \end{cases} \tag{3}$$

in other words, higher violations of the SLA lead to significant (quadratic) penalties, while the "opportunity cost" of wasted resources is linear (e.g., the money that another tenant would be willing to pay per unit). Note that other non-symmetric objectives (e.g., the ones used in [11], [12]) readily apply to our architecture. In our experiments, we put the quadratic coefficient $c_1 = 50$ and linear coefficient $c_2 = 1$ (It is important that the quadratic coefficient is high, as all traffic demands time series have been normalized to [0,1]).

III. Proposed Distributed Deep Neural Network

A. System Model

We assume a 5G network in which a set of BSs requires some resources. Each BS at time *t* demands an amount of resources (e.g., BW), d_t^i, to fulfill its corresponding user's SLAs. We have the past *N* demand values $\mathbf{d}_{t,N}^i = \{d_{t-N}^i, ..., d_{t-1}^i\}$. Traffic demand samples are random and possibly non-stationary. The vector $\mathbf{d}_{t,N}^i$ is given to the DDNN to determine the allocated resources for each BS at time *t*, $\hat{y}_t^i$. We can describe the DDNN with the following equation:

$$(\hat{y}_{L,t}^i, \hat{y}_{R,t}^i) = \mathcal{F}(\mathbf{d}_{t,N}^i; \boldsymbol{\theta}_{\mathcal{DDNN}}), \tag{4}$$

where the $\mathcal{F}(;\boldsymbol{\theta}_{\mathcal{DDNN}})$ is the approximation function that models the DDNN, and $\boldsymbol{\theta}_{\mathcal{DDNN}}$ represent the model parameters. Observe that, compared to the standard DNN of Eq. (1), the DDNN function here has two outputs: $\hat{y}_{L,t}^i$ and $\hat{y}_{R,t}^i$, which are the output of the local exit and the remote exit, respectively.

Our DDNN architecture is based on LSTM components that can naturally capture long-term dependencies between samples. The detailed architecture can be found in Fig. 1. This architecture can be seen as the first (very small) DNN module residing at the edge and making a local allocation decision:

$$\hat{y}_{L,t}^i = \mathcal{F}_L(\mathbf{d}_{t,N}^i; \boldsymbol{\theta}_L), \tag{5}$$

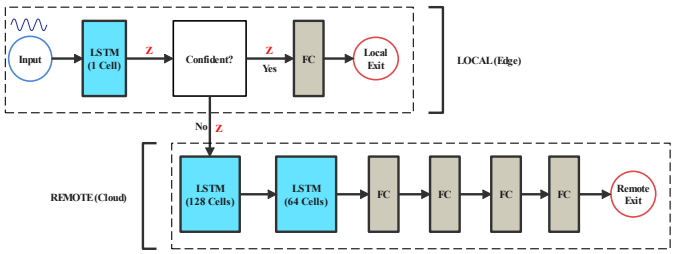


Fig. 1. LSTM network is distributed over Edge and Cloud. In the Online mode, the time series is given to the local DNN then the output is passed to the confidence mechanism which determines whether the sample will be inferred locally or remotely.

where $\mathcal{F}_L(;\boldsymbol{\theta}_L)$ corresponds to the local DNN layers, with $\boldsymbol{\theta}_L$ (the respective parameters). For some samples, this local decision might not be deemed appropriate (e.g., it is not a confident one). In such cases, the output of the local LSTM (e.g., z in Fig. 1) is sent to a second, much larger DNN module assumed to reside far from the location (e.g., BS) for which the decision is needed. This second module takes the output of the local LSTM block as input and provides its own allocation decision as follows:

$$\hat{y}^i_{R,t} = \mathcal{F}_R(\mathbf{z}^i_{t,N}; \boldsymbol{\theta}_R), \tag{6}$$

where $z^i_{t,N}$ is the output of the LSTM block in the local component that is given as the input to the remote part. $\mathcal{F}_R(;\boldsymbol{\theta}_R)$ corresponds to the additional DNN layers at the remote cloud, with $\boldsymbol{\theta}_R$ as its parameters.

B. Local Exit

At the local part, there is one LSTM block. This LSTM block has just **1** hidden unit. Compared to the remote part, the local LSTM is very simple. In the Training phase (offline mode), the output of this block (z^i_t) will be sent to the Fully Connected (FC) block in local and also to the remote layers (see Fig. 1 assume there is no Confidence). In the Inference time (online mode), the output of the local LSTM, i.e., z^i_t, will be given to the Confidence block, which decides whether this intermediate signal goes to the local FC or to the remote layers. The FC block in the local component is linear, the output of this block is called the *local prediction* or *local exit inference*, i.e., $\hat{y}^i_{L,t}$.

C. Remote Exit

The intermediate signal z^i_t is the input data of the remote DNN. At the remote DNN, first there is a LSTM block with **256** hidden units followed by a dropout block. Then, there is the last LSTM block with **128** hidden units which is also followed by a dropout block. Then, there are four FC blocks. The first three FCs have 128, 64 and 32 hidden neurons and each with ReLu activation function. The last FC block is linear to make the predictions. The output of this block is called the *remote prediction* or *remote exit inference*, i.e., $\hat{y}^i_{R,t}$.

IV. Joint Training and online Inference

A. Offline DDNN Training

While training a centralized DNN is quite straightforward, DDNN training requires *jointly* training the local Eq. (5) and remote Eq. (6) DNN modules towards achieving a common goal. Joint training means weighing both the local and remote exits in the objectives as follows and allowing the error of both to backpropagate through their respective DNN layers (we will elaborate shortly on this).

$$\begin{aligned} Loss_{DDNN} &= \sum_{m=1}^{M} w_L \cdot f(\mathcal{F}_L(\mathbf{d}; \boldsymbol{\theta}_L), d_m) \\ &\quad + w_R \cdot f(\mathcal{F}_R(\mathbf{z}; \boldsymbol{\theta}_R), d_m) \\ &= \sum_{m=1}^{M} w_L \cdot f(\hat{y}_{L,m}, d_m) + w_R \cdot f(\hat{y}_{R,m}, d_m). \end{aligned} \tag{7}$$

The idea of jointly training the local exits, together with the (standard) remote/final exit, has been first proposed in GoogleNet [17] and BranchyNet [18]. However, the use of local exits there was meant as an additional regularization method and not to be used at inference time[1]. Unlike that work, in our architecture the local exit is a crucial inference component, hence joint training aims at a desired performance trade-off between:

- backpropagating the local exit performance to the local layers (i.e., $\boldsymbol{\theta}_L$) to ensure that the majority of local decisions $\hat{y}_L$ can be relied on, despite being based on only a small/simple DNN module.
- backpropagating the remote exit performance to *both* the remote layers (i.e., $\boldsymbol{\theta}_R$) to ensure they can provide improved inferences $\hat{y}_R$, when needed, and also to the local layers (i.e., $\boldsymbol{\theta}_L$) to ensure that they produce sufficiently useful intermediate features (i.e., z) for the additional remote layers.

In Eq. (7), w_L is the "local weight" and w_R is the "remote weight", which regulate the impact of the local and remote exit on the overall loss of the DDNN during the joint training. These weights, $w_L, w_R \in [0,1]$ and $w_R = 1 - w_L$, play an important role in the optimization process. If we set $w_L = 0$ (which also means $w_R = 1$) then the DDNN resembles a centralized DNN and tries to optimize the performance in the remote exit. Similarly, if we set $w_L = 1$ (which also means $w_R = 0$) then the DDNN tries to optimize the performance in the local exit. Selecting the best (w_L, w_R) is crucial in order to achieve the desired goals (i.e., good local predictions, allocating resources locally for many slices, and good remote predictions).

It is important to note here that this type of distributed training is different from the one in [22]. Our goal is to distribute the actual architecture (between the edge and the core/cloud) appropriately, and not necessarily to also make

[1]The perspective of utilizing distributed DNNs with early exits [19], [20], [21] is a new and exciting research direction that has many unexplored possibilities.

the training process distributed (although this is certainly possible).

B. Online Inference and Offloading

Having trained the local and remote DNN modules to collaborate smoothly, there is one task remaining. As the sample goes through a forward pass reaching the local exit, the key decision is as follows: *Is this allocation decision predicted to be good enough, or should we continue the forward pass through the remote DNN layers as well?*.

It is important to note that this, in principle, is an unsupervised learning decision. We have to decide whether to stick with the local decision or transmit to the remote layers for additional processing *without knowing a priori whether this extra processing will actually help, and how much*.

Oracle-based Offloading: First, let us assume an "oracle" that *does* know the potential added value of remote processing for any sample. We will use this as reference. We can calculate the loss difference:

$$\mathcal{L}_m = f(\hat{y}_{L,m}, d_m) - f(\hat{y}_{R,m}, d_m), \tag{8}$$

and if, for example, we want to offload for example *40%* of the samples locally, then we pick *40%* of the samples with the lowest $\mathcal{L}_m$ for offloading in the local exit and the remaining samples, which have higher $\mathcal{L}_m$, will be exited remotely. Hence, if we *did* have such an oracle, we could certainly always keep the local decisions that will be better than the remote ones. While among samples for which the remote exit is better, we would choose to keep the local ones that are closest to the remote ones. Finally, when Eq. (8) is large, this suggests that the remote exit could offer significant cost benefits (hence justifying the additional overhead).

Bayesian Confidence-based Offloading: Unfortunately, in practice we do *not* have such an oracle, as the right term of Eq. (8), i.e., the remote decision $\hat{y}_{R,m}$ and the related cost, cannot be known at the edge, without actually sending the sample to the remote cloud. To this end, the authors of [13] propose to use the entropy of the local image classification as a confidence "proxy" for Eq. (8): significant uncertainty in the classification suggests potential for improvement by sending the sample through additional (remote) layers. Nevertheless, this cross-entropy metric is not applicable to non-classification problems like ours. In fact, very little can be found in related DDNN literature ([18], [23]) for offloading decisions in such regression-type problems. We propose a methodology based on random dropouts, applied to the local forward pass, motivated by the Bayesian confidence metric in [16] (this dropout is different from the dropout block used as a regularizer during the training).

The confidence block includes a dropout layer with dropout probability $p = 0.4$, followed by a linear FC block. The intermediate signal z is given to the confidence block and it is forced to infer for each input sample, say $J = 10$ times. The randomness of the dropout makes the inference of the confidence block different at each time. Therefore, for each base station, we have an array $\in \mathcal{R}^J$. For each base station $k \in \mathcal{K}$, we calculate the standard deviation (σ_k) and then take the average among the K base stations. We refer to this value as **Uncertainty**:

$$U = \frac{1}{K} \sum_{k=1}^{K} \sigma_k. \tag{9}$$

This metric serves as a worst case estimate of how much perturbations have affected the local decisions. The confidence mechanism compares the measured uncertainty (U) value with a given confidence threshold (η), which is a design parameter of the DDNN. If $U < \eta$, then the model is confident about the local decision, and it is considered "good enough". Otherwise, the intermediate signal z will be sent to the remote layers, where the remote decision is assumed to be the correct decision, and then we proceed with the next set of samples[2].

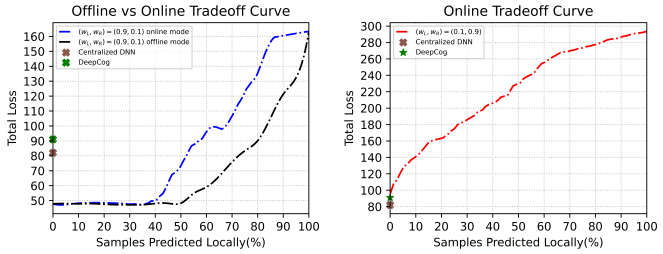
As the confidence block is used for inference in the online mode, a question that comes is: *Is the latency savings from avoiding the central cloud, enough to justify the extra latency added by this new "confidence block" at the edge?* If we send everything to the central cloud, then the total time to make a decision is the sum of two components: the round-trip transmission time (RTT) from the edge to the cloud and the processing time through all the layers of the full model for all the samples. In DDNN, the total time comprises the processing time of the confidence mechanism for all samples, the processing time of the local FC for confident samples, the RTT from the edge to the cloud for non-confident samples, and the processing time of remote layers for non-confident samples. We will investigate this latency trade-off in the next section.

V. Performance Evaluation

A. Data Preparation

To train and test our proposed architecture, we will use the Milano dataset [24], which is publicly available and has been used in many related works [16], [25], [26]. The LSTM network takes sequences (time series) as the input. LSTMs yield good results with short sequences, up to 100-300 samples in a sequence. On longer sequences, LSTMs still work but can gradually forget information from the oldest samples. We use the past 144 samples (the number of samples measured in one day) to predict the next sample for all 16 base stations together. In other words, in the sequence $\mathbf{d}_{t,N}^i = \{d_{t-N}^i, ..., d_{t-1}^i\}$ we set *N = 144* and $K = 16$. As a result, the input of the DDNN is an array with dimensions (Number of samples, 16, 144), and the output is an array with dimensions (Number of samples, 16, 1). We use Python and PyTorch for implementing our models. We run the models on Google Colab server using an Nvidia V100 GPU, 16 GB HBM2, and 32GB RAM.

[2]Note that, when deciding the allocation for multiple *correlated* elements, we either make all K decisions locally or all K decisions remotely. In future work, we plan to investigate more complex hierarchies of layers, with potentially partial views.



(a) Offline vs Online trade-off curve for $(w_L, w_R) = (0.9, 0.1)$

(b) Online trade-off curve for $(w_L, w_R) = (0.1, 0.9)$

Fig. 2. Total loss vs percentage of samples predicted locally

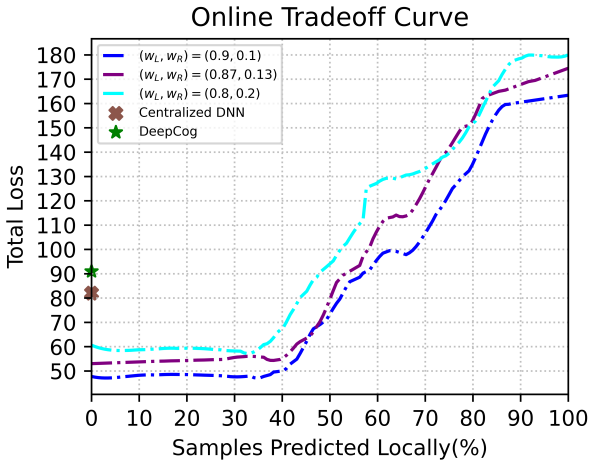


Fig. 3. Total loss vs percentage of samples predicted locally as η increases in online mode.

B. Performance Metrics

The DDNN overall cost is the summation of the local and the remote loss when some percentage of samples at each exit point in the hierarchy is offloaded:

$$C_{DDNN} = \sum_{m=1}^{M} \mathcal{I}_m \cdot f(\hat{y}_{L,m}, d_m) + (1 - \mathcal{I}_m) \cdot f(\hat{y}_{R,m}, d_m), \tag{10}$$

where $\sum_{m=1}^{M} \mathcal{I}_m \cdot f(\hat{y}_{L,m}, d_m)$ represents the local exit loss, $\sum_{m=1}^{M} (1 - \mathcal{I}_m) \cdot f(\hat{y}_{R,m}, d_m)$ represents the remote exit loss, and $\mathcal{I}_m$ indicates whether the sample exited locally or not.

$$\mathcal{I}_m = \begin{cases} 1 & \textit{if the sample } m \textit{ exited locally} \\ 0 & \textit{else.} \end{cases} \tag{11}$$

We train a centralized DNN with the same size as the proposed distributed model. Additionally, we implement the model in [11] known as DeepCog and train the model with our data. The centralized DNN loss, the DeepCog loss, and the DDNN total loss in offline mode (Oracle) serve the baselines for comparison with the DDNN total loss in the online mode.

C. Resource Allocation and Communication Trade-off

Experiment 1: After jointly training the model, for each η (confidence threshold) in $[0, 1]$ we measure the samples in the test set that can be exited locally (according to the explanations given in IV-B) and then calculate the total loss, as in Eq. (10). We plot the trade-off curve, which represents the total loss versus the percentage of samples resolved in the local exit. When we use *Oracle-based Offloading*, we obtain the offline trade-off curve. By using *Bayesian Confidence-based Offloading*, we produce the online trade-off curve.

In Fig. 2(a) we plot both offline and online trade-off curves for the model with (local, remote) training weights $(w_L, w_R) = (0.9, 0.1)$. We can see that the Bayesian offloading mechanism (blue curve) has a good performance compared to the oracle offloading (green curve), which represents the ideal offloading policy. We can see that when 40% (or less) of the samples are exited locally, the oracle and Bayesian mechanisms have similar performance.

The online trade-off curve for three models is shown in Fig. 3. We repeat the process for three DDNN models each using the following (local, remote) training weights, respectively: $(0.8, 0.2)$, $(0.87, 0.13)$ and $(0.9, 0.1)$. In this figure, we also mark the centralized DNN loss and the DeepCog loss. The centralized LSTM architecture indeed slightly outperforms the centralized CNN-based architecture of DeepCog, as expected. (We remind you that our main goal here is not to improve the DNN architecture itself, but rather to investigate how to distribute more complex, memory-based DNNs for such tasks).

As we can see in Fig. 3 the $(w_L, w_R) = (0.9, 0.1)$ model achieves the best trade-off in the considered scenario. It is possible to resolve more than *50%* of the samples locally while the overall loss equals that of the centralized DNN. The $(w_L, w_R) = (0.8, 0.2)$ model, can resolve all the samples remotely, i.e., $\eta = 0$, while the total loss is nearly *25%* less than that of the centralized DNN loss. Also, with this model, we can offload more than *40%* of the samples locally while the cost is the same as that of the centralized DNN[3].

It is evident that the overall trade-off curve is affected by the choice of the (offline) training weights. In Fig. 2(b), the online trade-off curve for the model with $(w_L, w_R) = (0.1, 0.9)$ is shown (by increasing the confidence threshold η, we can plot it). We observe that with this pair of weights, the model doesn't perform well and in fact, its loss is always more than that of the centralized models. We have also analyzed other weight combinations, yet it is clear that even for "non-optimal" weight choices, there is still an interesting trade-off achieved. Another important observation is that, in our model a local weight higher than the remote weight ($w_L > w_R$ or $w_L > 0.5$) is needed to counterbalance the fact that the local module is much simpler/shallower, and be able to surpass the centralized DNN performance.

[3]One might wonder why we manage to achieve better performance than a fully centralized DNN, with lower overhead (win-win), rather than "almost as good performance but with lower overhead". This is due to the effect of the local exit on the gradient flow, a phenomenon observed in other research studies like [13], [18], [19], [27] and the additional regularization effect of the local exit is declared to be responsible for this.

TABLE I
LATENCY COMPARISON

L (%)	5	20	40	50	60	80	95	CDNN	DeepCog
T (ms)	41.01	34.62	26.11	21.86	17.61	9.10	2.72	42.67	42.63

Experiment 2: In Fig. 4, the real traffic demand (BandWidth) (**d**) in the test set, the local allocations ($\hat{\mathbf{y}}_L$), and the remote allocations ($\hat{\mathbf{y}}_R$) for one of the base stations with two different (local, remote) training weights $(w_L, w_R) = (0.9, 0.1)$ and $(w_L, w_R) = (0.1, 0.9)$ have been illustrated. To obtain local and remote predictions for all samples, we don't use the confidence mechanism. In Fig. 4(a), we can see that with $(w_L, w_R) = (0.9, 0.1)$ the local exit has good performance and some samples can be predicted locally, while Fig. 4(b) shows that with $(w_L, w_R) = (0.1, 0.9)$, the local exit doesn't work well and using local predictions increases the cost which was expected according to the trade-off curve for this weight pair in Fig. 2(b). Also, we observe that both models avoid SLA violations, rather than trying to "match" the demand (as an MSE objective would), due to the higher cost of under-provisioning in our objective function.

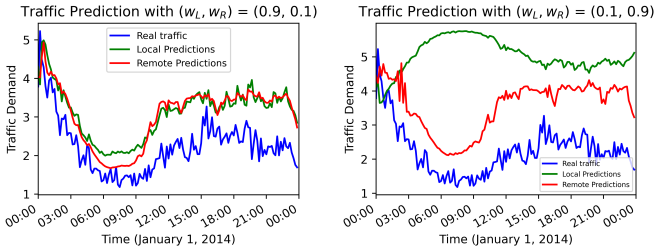
Experiment 3: As we explained in section IV-B we need to calculate the "Communication" and "Computation" latency in order to see if our model is adding latency or if we have latency gains. **Communication time**: We refer to the recent systems-oriented study in [28] where the authors set the round transmission time (RTT) from edge to cloud, on average, to 42.46 ms for each sample. **Computation time**: We run each model multiple times on the same server and calculate the average processing time for each sample using different models[4].

The results are shown in Table I. L represents the percentage of samples that exit locally and T represents the average time for resolving one sample in each scenario. For example, when 40% of the samples resolve locally, then T = average processing time of the confidence block for a sample + 0.4 (average processing time of the local inference for a sample) + 0.6 (RTT + average processing time of the remote inference for a sample). We can conclude that as more samples are resolved locally, the inference latency decreases. For example, when the DDNN resolves *50%* of the samples locally, it has nearly the same cost as the centralized baselines, while experiencing *49%* lower inference latency.

Key Observation 1: It is possible to distribute a sophisticated LSTM-based DNN architecture and achieve significant latency/communication reduction (due to local resolution of allocation decisions), with almost no penalty on the total cost.

Key Observation 2: The Bayesian offloading confidence mechanism has a good performance in detecting which samples can be offloaded locally in the online mode.

[4]Please note that the setup for implementing such a distributed architecture can differ. In this experiment, we aim to create a scenario with practical values as an example. This is done to effectively demonstrate the potential reduction in latency. We run the models on a server using an Nvidia V100 GPU, 16 GB HBM2, and 32GB RAM.



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

Fig. 4. Traffic demand predictions for two base stations, data forwarded pass (no confidence mechanism)

Key Observation 3: The training weights (w_L, w_R) significantly impact the model performance. Choosing the right weights to jointly train the local and remote modules is crucial for the overall performance.

Key Observation 4: Having a local weight higher than the remote weight $(w_L > w_R)$ is necessary to counterbalance the fact that the local module is simpler/shallower and to surpass the performance of the centralized DNN.

Key Observation 5: (SLA Violations Avoidance) The models prioritize avoiding service level agreement violations over matching demand, aligning with the higher cost associated with under-provisioning in the objective function.

Key Observation 6: (Latency Reduction with Local Resolution) As more samples are resolved locally, the inference latency decreases.

VI. CONCLUSIONS AND FUTURE WORK

We designed and implemented a Distributed Deep Neural Network (DDNN) for forecasting future traffic demand and allocating resources accordingly in 5G+ networks. The proposed DDNN is a DNN with multiple exit points: one local exit (e.g., Edge) and one remote exit (e.g., Cloud). The DDNN needs to be trained jointly to achieve the desired goals. During joint training, a weight is assigned to the local exit, and another weight is assigned to the remote exit, which encourages good performance at the local exit and also affects the performance of the remote exit. Additionally, the objective function plays an important role in avoiding under-provisioning. We use a Bayesian confidence mechanism to determine either the samples should offload locally or remotely. In comparison with centralized models, our algorithm can achieve lower or equal cost performance while resolving more than *50%* of the samples locally and also reducing latency.

We are currently investigating alternative confidence mechanisms to reduce computational usage and accelerate the allocation process in online mode. Additionally, we are considering implementing the model with multiple local exits. As the local and remote weights play an important role, adaptive tuning of local and remote weights (w_L, w_R) during training could be an interesting topic for future work.

VII. Acknowledgment

This project has been supported by the European Union's Horizon 2020 research and innovation program under Grant Agreement No. 861165 (SEMANTIC Project).

References

[1] C. Zhang, P. Patras and H. Haddadi, "Deep Learning in Mobile and Wireless Networking: A Survey," in IEEE Communications Surveys and Tutorials, vol. 21, no. 3, pp. 2224-2287, 2019.

[2] C. -X. Wang, M. D. Renzo, S. Stanczak, S. Wang and E. G. Larsson, "Artificial Intelligence Enabled Wireless Networking for 5G and Beyond: Recent Advances and Future Challenges," in IEEE Wireless Communications, vol. 27, no. 1, pp. 16-23, February 2020.

[3] D. P. Kumar, T. Amgoth, C. S. R. Annavarapu, "Machine learning algorithms for wireless sensor networks: A survey," Information Fusion, vol. 49, pp. 1-25, ISSN 1566-2535, 2019.

[4] M. Chen, U. Challita, W. Saad, C. Yin and M. Debbah, "Artificial Neural Networks-Based Machine Learning for Wireless Networks: A Tutorial," in IEEE Communications Surveys and Tutorials, vol. 21, no. 4, pp. 3039-3071, 2019.

[5] H. Halabian, "Distributed Resource Allocation Optimization in 5G Virtualized Networks," in IEEE Journal on Selected Areas in Communications, vol. 37, no. 3, pp. 627-642, March 2019.

[6] Y. Liu, J. Ding and X. Liu, "A Constrained Reinforcement Learning Based Approach for Network Slicing," 2020 IEEE 28th International Conference on Network Protocols (ICNP), Madrid, Spain, pp. 1-6, 2020.

[7] C. Zhang, M. Fiore, C. Ziemlicki, P. Patras, "Microscope: Mobile Service Traffic Decomposition for Network Slicing as a Service," MobiCom '20: Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, no. 38, pp. 1–14, April 2020.

[8] V. Sciancalepore, X. Costa-Perez and A. Banchs, "RL-NSB: Reinforcement Learning-Based 5G Network Slice Broker," in IEEE/ACM Transactions on Networking, vol. 27, no. 4, pp. 1543-1557, August 2019.

[9] J. Salvat, L. Zanzi, A. Garcia-Saavedra, V. Sciancalepore and X. Costa-Perez, "Overbooking Network Slices through Yield-Driven End-to-End Orchestration," CoNEXT '18: Proceedings of the 14th International Conference on emerging Networking Experiments and Technologies, pp. 353–365, December 2018.

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

[11] D. Bega, M. Gramaglia, M. Fiore, A. Banchs and X. Costa-Perez, "DeepCog: Cognitive Network Management in Sliced 5G Networks with Deep Learning," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, pp. 280-288, 2019.

[12] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Aztec: Anticipatory capacity allocation for zero-touch network slicing," IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, Toronto, ON, Canada, pp. 794-803, 2020.

[13] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, pp. 328-339, 2017.

[14] O. Nassef, W. Sun, H. Purmehdi, M. Tatipamula, and T. Mahmoodi, "A survey: Distributed Machine Learning for 5G and beyond," Computer Networks, vol. 207, ISSN 1389-1286, 2022.

[15] X. Chen, C. Wu, Z. Liu, N. Zhang and Y. Ji, "Computation Offloading in Beyond 5G Networks: A Distributed Learning Framework and Applications," IEEE Wireless Communications, vol. 28, no. 2, pp. 56-62, April 2021.

[16] T. Giannakas, T. Spyropoulos and O. Smid, "Fast and accurate edge resource scaling for 5G/6G networks with distributed deep neural networks," 2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), Belfast, United Kingdom, pp. 100-109, 2020.

[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, pp. 1-9, 2015.

[18] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," 23rd International Conference on Pattern Recognition (ICPR), pp. 2464-2469, 2016.

[19] 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.

[20] 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," HotNets '18: Proceedings of the 17th ACM Workshop on Hot Topics in Networks, pp. 50–56, November 2018.

[21] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, pp. 1423-1431, 2019.

[22] 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, ISSN 1084-8045, 2018.

[23] Y. Matsubara, M. Levorato, and F. Restuccia, "Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges," ACM Computing Surveys, vol. 55, issue 5, No. 90, pp. 1-30, 2022.

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

[25] N. Liakopoulos, G. Paschos, and T. Spyropoulos, "Robust user association for ultra dense networks," IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, Honolulu, HI, USA, pp. 2690-2698, 2018.

[26] N. Liakopoulos, A. Destounis, G. Paschos, T. Spyropoulos, and P. Mertikopoulos, "Cautious regret minimization: Online optimization with long-term budget constraints," Proceedings of the 36th International Conference on Machine Learning (PMLR), vol. 97, pp. 3944–3952, 2019.

[27] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," Proceedings of the 36th International Conference on Machine Learning (PMLR), vol. 97, pp. 3301–3310, 2019.

[28] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska, "Couper: DNN Model Slicing for Visual Analytics Containers at the Edge," SEC '19: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, pp. 179–194, November 2019.