Edge/Cloud Slice Resource Allocation for Beyond 5G Networks With Distributed LSTM

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

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

***Abstract*—Efficient resource allocation among slices/users with different Service Level Agreements (SLAs) is a critical task in 5G+ networks, which has prompted recent research into Deep Neural Networks (DNNs). However, challenges arise when dealing with edge resources, including the ability to rapidly scale resources (in the order of milliseconds), and the cost of transmitting large data volumes to a cloud for centralized DNN-based processing. Addressing these issues, we introduce a novel architecture based on Distributed Deep Neural Networks (DDNN). This architecture features a compact set of DNN layers located at the network's edge, designed to function as an autonomous resource allocation unit. Complementing this, there is an intelligent offloading mechanism that delegates a fraction of hard decisions to additional DNN layers situated in a remote cloud (when needed). To implement offloading, we propose a theoretically informed method that learns to mimic an oracle that knows which sample will benefit from additional processing in the cloud. We compare this to a previously proposed heuristic, based on a Bayesian-confidence mechanism. We investigate the interplay of (offline) joint training of the DDNN exits and the ML-based offloading mechanism, and demonstrate that our architecture resolves more than 50% of decisions at the edge with no additional penalty compared to centralized models as well as consistently outperforms previous methods.**

***Index Terms*—Network Slicing, Resource Allocation, Distributed Deep Neural Network, Offloading Mechanism, 5G Networks**

I. Introduction

The advent of 5G and the evolution towards 5G+/6G networks introduces key architectural changes, emphasizing virtualization and resource slicing to support numerous tenants with diverse Quality of Service (QoS) requirements and Service Level Agreements (SLAs). This shift facilitates the use of Virtual Network Functions (VNFs) for enhanced network flexibility and opens up new possibilities for data-driven optimization in wireless network resource allocation. Moving away from traditional models, modern Machine Learning (ML) methods, including deep learning [1], [2], [3], and reinforcement learning [4], [5], [6], are being explored for tasks like slice resource allocation and resource orchestration, marking a significant advancement in the field.

However, within this framework, optimization objectives are driven by the requirements stipulated in SLAs and frequently exhibit an asymmetric nature. The discrepancy between the costs of under-provisioning, which may lead to SLA violations, and over-provisioning costs, representing resource wastage, necessitates a tailored approach in training Deep Neural Networks (DNNs). Additionally, implementing a centralized, heavy-duty DNN for network optimization faces several hurdles. These include *stringent latency requirements*, especially for tasks within the Radio Access Network (RAN) where low latency is crucial. This contrasts with higher latency-tolerant application layer tasks that can be offloaded to central clouds. Another challenge is the *data transmission overhead*; transmitting raw data over potentially congested edge and wireless links to a deep-core network-based DNN architecture can pose a significant obstacle to practical implementation.

Therefore, our focus shifts towards Distributed Deep Neural Networks (DDNN), which aim to overcome these challenges while retaining DNN efficiency as discussed in [7], [8], [9]. DDNNs distribute DNN layers across various locations, allowing for local predictions at the edge when stringent latency requirements or network congestion dictate, or remote (cloud) predictions when heightened accuracy is essential, which necessitates joint training of both local and remote layers. Building upon this research, we recently explored in [10] a distributed adaptation of the architecture presented in [11] and [12], incorporating a 3D Convolutional Neural Network (3D-CNN) for slice resource allocation. The primary objective of this paper is to introduce, train, and analyze a distributed architecture designed to address a data-driven edge resource allocation problem, which demonstrates the generality of this methodology by utilizing a more sophisticated DNN architecture for the same task based on LSTM (Long Short Term Memory) units. Our main contributions are outlined as follows:

- (*Architecture*) We propose and train a distributed LSTM architecture that includes a small number of LSTM units at the network edge, featuring a "local exit" for rapid inferences. Additionally, it encompasses a larger number of units and a "remote exit" located in the central cloud. We selected LSTM for its proven effectiveness in handling time series data and also to generalize our initial findings.
- (*Offline Optimization*) We demonstrate the importance of fine-tuning the joint training hyperparameters for local and remote exits, aiming to strike a balance between empowering the local layers to make a sufficient number of accurate allocation decisions as well as generating valuable features that can be effectively utilized by the remote layers, particularly when there is a requirement

for enhanced decisions.

- (*Online Optimization*) We propose an ML-based offloading mechanism that learns to "hand-pick" the few samples that could benefit from additional processing in the cloud, enough to amortize the additional communication and latency costs this implies. This decision is made blindly at the edge, *before actually sending anything to the cloud*. Our mechanism is trained using past data samples, trying to mimic an oracle for this task. We implement and compare our method to a previously proposed heuristic based on dropouts, and show that the proposed mechanism consistently outperforms the heuristic, better matching the oracle performance.

The remainder of this paper is organized as follows: Section II discusses the problem setup, and Section III elaborates our LSTM-based DDNN architecture. Section IV explores both the offline joint training and the online offloading. Section V validates the architecture with real traffic data. Section VI presents future work and the conclusion.

II. Data-driven Resource Allocation

Slice Resource Allocation with DNN: We assume a network infrastructure scenario where a provider hosts multiple network slices, each comprising various Virtual Network Functions (VNFs). We conceptualize each VNF as a discrete-time signal or time series, with values reflecting the computational resources (such as CPU and memory) required by these VNFs. We denote the set of VNFs as $\mathcal{K} = \{1, ..., K\}$ and represent the resource demand of VNF k at time t as d_t^k. We maintain a record of the previous N traffic samples for all VNFs. Our objective is to leverage this archive of past demands to efficiently allocate resources for all $\mathcal{K}$ VNFs using a DNN-based architecture. We formulate the DNN-based resource allocation problem as follows:

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

where $\mathbf{d}_{t,N}^k$ is the DNN input. $\mathbf{d}_{t,N}^k = \{d_{t-N}^k, ..., d_{t-1}^k\}$ includes the N traffic samples of VNF $k \in \mathcal{K}$ before the time t. N is the input vector size and is fixed during the model training. $\mathcal{F}(\cdot; \boldsymbol{\theta})$ represents the DNN as an approximation function, with $\boldsymbol{\theta}$ being the vector of model parameters (i.e., weights of the DNN). $\hat{y}_t^k$, the DNN's forecast for network element k at time t, aims to balance the costs of under- and over-provisioning against the expected (unknown) demand d_t^k at that time.

Objective Function for Slice Resource Allocation: In the realm of standard traffic forecasting, the objective is to predict traffic at a given time t based on a series of past N traffic samples, with the aim for the predicted value $\hat{y}_t$ to closely approximate the actual traffic d_t. This is traditionally achieved through training a DNN with a least squares objective function.

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

A distinction of our work from conventional models lies in the asymmetric nature of the costs of underestimating and overestimating the demand (unlike in Eq. (2)). Under-provisioning (when the predicted traffic $\hat{y}_t$ is less than the actual demand d_t) can lead to insufficient resource allocation for a network slice, potentially violating the tenants' SLAs. Over-provisioning (when the predicted traffic $\hat{y}_t$ is more than the actual demand d_t) results in an excess allocation of resources, leading to inefficiencies and resource wastage. To address this, a DNN is tailored to optimize objectives that vary based on the cost implications of SLA violations (under-provisioning) and resource wastage (over-provisioning). Without loss of generality, we will adopt the following objective function:

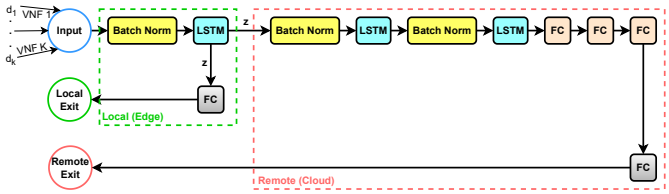


Fig. 1: LSTM network is distributed over Edge and Cloud.

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

where higher penalties are applied for SLA violations through quadratic terms and the "opportunity cost" of wasted resources is linear (e.g., the money that another tenant would be willing to pay per unit)[1].

III. Proposed Distributed Deep Neural Network

To address the challenges and drawbacks of centralized DNNs, we adopt a distributed DNN. We assume that edge computing capabilities can accommodate a portion of the DNN's computational workload, although not its entirety, enabling it to make an immediate and valid resource allocation decision via its early exit feature. Consequently, this necessitates the partitioning of the DNN across two geographically distant locations: the edge and the remote cloud.

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

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

where the $\mathcal{F}(\cdot; \boldsymbol{\theta}_{\text{DDNN}})$ is the approximation function that models the DDNN, and $\boldsymbol{\theta}_{\text{DDNN}}$ represents the model parameters. Observe that, compared to the standard DNN of Eq. (1), the DDNN function has two outputs: $\hat{y}_{L,t}^k$ and $\hat{y}_{R,t}^k$, which are the outputs 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

[1]Note that our architecture is adaptable to various non-symmetric objective functions, similar to those in [11], [12].

samples. The detailed architecture can be found in Fig. 1. This architecture includes a very small initial DNN module located at the edge, responsible for making a local allocation decision:

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

where $\mathcal{F}_L(\cdot; \boldsymbol{\theta}_L)$ represents the local DNN, with its parameters $\boldsymbol{\theta}_L$. For some samples, this local decision might not be deemed appropriate (i.e., have a relatively high cost). In such cases, the output of the local LSTM (e.g., $\mathbf{z}^k_t$ as shown in Fig. 1) is sent to a much larger DNN module, assumed to be located remotely from the decision-required site (e.g., BS). This remote module provides its own allocation decision as follows:

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

where $\mathbf{z}^k_t$ is the output of the LSTM block in the local component that is given as the input to the remote part. $\mathcal{F}_R(\cdot; \boldsymbol{\theta}_R)$ represents the remote DNN, with $\boldsymbol{\theta}_R$ as its parameters.

Local Exit: In a DDNN, the local exit is part of the initial layers of the neural network situated at the network edge. Without loss of generality, the local component of our DDNN includes a single LSTM block with just one hidden unit. The output from this block, $\mathbf{z}^k_t$, is routed to both the local Fully Connected (FC) block and the remote layers (see Fig. 1). The output of the local FC is termed the *local prediction* or *local exit inference*, i.e., $\hat{y}^k_{L,t}$.

Remote Exit: In a DDNN, the remote exit is part of the network architecture that is located in a central cloud. Without loss of generality, the remote component of our DDNN comprises two LSTM blocks with 256 and 128 hidden units. Following these, there are four Fully Connected (FC) blocks. The first three FC blocks are configured with 128, 64, and 32 hidden neurons, all utilizing the Rectified Linear Unit (ReLU) activation function. The final FC block is linear. The output of the remote linear FC block, termed the *remote prediction* or the *remote exit inference*, i.e., $\hat{y}^k_{R,t}$ (see Fig. 1).

IV. DDNN Training and Inference

A. Offline DDNN Joint Training

Training a DDNN is more complex than a centralized DNN, as it involves the concurrent training of both local DNN and remote DNN modules with a unified objective. This joint training process requires carefully balancing the contributions of both local and remote exits in the objective function and enabling the backpropagation of loss through the respective layers of each DNN module. The DDNN loss is calculated as:

$$\begin{aligned} \text{DDNN Loss} &= \sum_{k=1}^{K} w_L \cdot f(\mathcal{F}_L(\mathbf{d}^k_{t,N}; \boldsymbol{\theta}_L), d^k_t) \\ &\quad + w_R \cdot f(\mathcal{F}_R(\mathbf{z}^k_t; \boldsymbol{\theta}_R), d^k_t) \\ &= \sum_{k=1}^{K} w_L \cdot f(\hat{y}^k_{L,t}, d^k_t) + w_R \cdot f(\hat{y}^k_{R,t}, d^k_t). \end{aligned} \tag{7}$$

Joint training of local exits alongside standard remote/final exits was initially introduced in [13], [14] (GoogleNet), and [15] (BranchyNet), primarily as a regularization technique, not intended for use during inference. In contrast, our architecture integrates the local exit as a key component in the inference process. Unlike those works, our joint training approach is designed to achieve a specific performance trade-off between:

- Backpropagating the performance of the local exit to its layers (i.e., $\boldsymbol{\theta}_L$) to ensure the reliability of local decisions $\hat{y}^k_{L,t}$, even though they are made by a smaller/simpler DNN module.
- Backpropagating the remote exit's performance to *both* remote (i.e., $\boldsymbol{\theta}_R$) and local layers (i.e., $\boldsymbol{\theta}_L$), enhancing the remote layers inferences ($\hat{y}^k_{R,t}$), and ensuring the local layers produce valuable intermediate features (i.e., $\mathbf{z}^k_t$) for further remote processing.

In Eq. (7), the "local weight" (w_L) and "remote weight" (w_R) are critical in determining the influence of the local and remote exits on the overall loss during the DDNN's joint training. These weights, constrained within [0, 1] where $w_R = 1 - w_L$, are pivotal in the optimization process. Setting $w_L = 0$ (and hence $w_R = 1$) makes the DDNN function similarly to a centralized DNN, focusing on optimizing remote exit performance. Conversely, setting $w_L = 1$ (and $w_R = 0$) shifts the focus to optimizing local exit performance. The selection of optimal w_L and w_R values is crucial for achieving desired outcomes, such as reliable local predictions, effective local resource allocation, and accurate remote predictions.

Our approach differs from that in [16] (distributed learning), as we concentrate on distributing the actual architecture between the edge and the core/cloud, rather than focusing on distributed training, although it remains a feasible option.

B. DDNN Online Inference

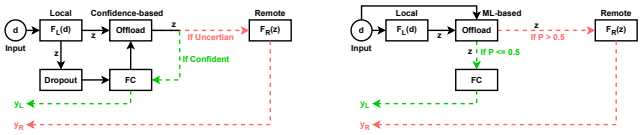
After training the local and remote DNN modules for effective collaboration, a critical decision remains during the forward pass at the local exit. *Whether the allocation decision made at this stage is sufficient or if the process should extend to the remote DNN layers for further refinement*[2].

This decision-making process, in principle, is an unsupervised learning task. It involves choosing between relying on the local decision or escalating to the remote layers for additional processing, *without prior knowledge of the potential benefits or extent of this extra processing*.

Oracle-based Offloading: Initially, we assume an "oracle" with the knowledge of the potential added value of remote processing for each sample, which we will utilize as a reference in our analysis. We can calculate the loss difference:

$$\mathcal{L} = \sum_{k=1}^{K} (f(\hat{y}^k_{L,t}, d^k_t) - f(\hat{y}^k_{R,t}, d^k_t)) = C_L - C_R, \tag{8}$$

[2]Note that we are not predicting the traffic load, we are predicting the allocation of resources that will minimize the total cost, ideally balancing under-provision costs and SLA penalties with over-provision costs due to wasted resources.



(a) Confidence-based (b) ML-based

Fig. 2: DDNN Inference Scheme (Online Optimization): In both configurations, only the remote block is located in the cloud, while all other blocks are deployed at the edge.

we then compare the loss difference, $\mathcal{L}$, to the round trip transmission cost (from edge to cloud and back), C_T, which is known because of oracle assumption. We offload the samples for which $\mathcal{L} < C_T$, indicating that the overall cost of local resolution is lower than that of remote resolution, and conversely, $\mathcal{L} > C_T$ suggests that decisions made remotely are more beneficial. Hence, if such an oracle existed, it would enable us to consistently make correct decisions about whether a sample should be offloaded locally or should be forwarded to the remote layers.

Bayesian Confidence-based Offloading: In practice, we lack such an oracle, meaning the remote decision $\hat{y}^k_{R,t}$ and its associated costs (remote cost and transmission cost) remain unknown at the edge. This information is ascertained only by transmitting the sample to the remote cloud. To this end, the authors in [10] propose a Bayesian confidence metric based on random dropouts, applied to the local forward pass (this is different from the dropout regularizer block). The confidence block, consisting of a dropout block with dropout probability $p = 0.4$ and a linear FC block, processes the intermediate signal $\mathbf{z}^k_t$ to perform inferences on each input sample ten ($J = 10$) times. The inferences of the confidence block vary each time due to the random nature of dropout. For each VNF $k \in \mathcal{K}$, the standard deviation (σ_k) is calculated, and the average of these values across all K VNFs is defined as the **Uncertainty** ($U = \frac{1}{K}\sum_{k=1}^{K}\sigma_k$). The confidence mechanism assesses the measured uncertainty (U) against a predefined confidence threshold (η). If $U < \eta$, the local decision is deemed confident (green path in Fig. 2(a)). If not, the intermediate signal $\mathbf{z}^k_t$ is forwarded to remote layers, where it is presumed to make accurate decisions (red path in Fig. 2(a))[3].

ML-based offloading: We propose an offloading mechanism leveraging the power of neural networks. We train a "binary classifier" that decides whether the samples should be resolved locally or remotely, subsequent to the DDNN training.[4]. During the (offline) DDNN training all samples go through both the local and remote layers, therefore after the training, for the training set samples, the local cost $C_L = \sum_{k=1}^{K} f(\hat{y}^k_{L,t}, d^k_t)$, the remote cost $C_R = \sum_{k=1}^{K} f(\hat{y}^k_{R,t}, d^k_t)$, and the transmission cost C_T are known.

The classifier uses the training samples and categorizes samples into two classes: **class 0:** $C_T < C_L - C_R$, the remote cost is small enough to amortize the cost C_T, therefore it is better to resolve the sample in the remote cloud. **Class 1:** $C_T \geq C_L - C_R$, indicates the remote cost is very high and it is beneficial to resolve the sample locally. Without loss of generality, we choose a neural network with three FC layers for the binary classifier. Its output, denoted by p, is a probability indicating the likelihood of offloading a sample either locally or remotely (this employs a supervised learning method).

As shown in Fig. 2(b), during inference (online mode) the input signal ($\mathbf{d}^k_{t,N}$) is given to the local DNN and also to the ML-based offloading block. The offloading mechanism computes the p and makes the decision for offloading. If the decision is to avoid the edge-cloud round trip, the intermediate signal $\mathbf{z}^k_t$ will be forwarded to the local FC and the resources are allocated based on $\hat{y}^k_{L,t}$ (green path in Fig. 2(b)). Conversely, if the classifier deems that the sample necessitates additional processing layers, the intermediate signal $\mathbf{z}^k_t$ will be forwarded to the remote DNN and the remote inference $\hat{y}^k_{R,t}$ is the allocation decision (red path in Fig. 2(b)).

Although this approach requires an offline training step and some additional hardware to implement the classifier, unlike the Bayesian Confidence-based Offloading, it does not require J costly forward passes of the local FC. Therefore, at inference time, it is faster and adds minimal latency.

The online use of the offloading block at the edge raises a critical question: *does the latency saved by bypassing the central cloud outweigh the additional latency introduced by this block?* In a centralized system, decision-making time is the sum of the round-trip transmission time (RTT) to the cloud and the processing time for all samples through the full model. In contrast, the DDNN's total time includes processing time for the offloading mechanism, processing time for locally resolved samples, RTT to the cloud for remotely resolved samples, and processing time for remotely resolved samples. We delve into this latency trade-off in the next section.

V. Performance Evaluation

A. Preliminaries

Data Preparation: To train and test the architecture, we utilize the publicly available Milano dataset [17], frequently used in related studies [10], [18]. LSTMs yield good results with short time series, typically between 100 and 300 samples in a sequence. We use the past 144 samples (the daily measured samples) to predict the next sample for 16 base stations together, i.e., we set N to 144 and K to 16. Therefore, 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 set the quadratic coefficient $c_1 = 50$ and linear coefficient $c_2 = 1$ in Eq. (3) (It is important to set the quadratic coefficient high because all traffic demand time series are normalized to [0,1]). We use Python and PyTorch to implement our models. We run the models on Google Colab

[3]It is important to note that in allocating resources for multiple *correlated* elements, the decision is binary: either all K decisions are made locally or all are made remotely. In future work, we plan to investigate more complex hierarchies of layers, with potentially partial views.

[4]This binary classifier is trained subsequent to the DDNN, to ensure the stability of the entire data path, considering the DDNN's training sensitivity to the choice of local/remote exit weights, as we shall see.

server using an Nvidia V100 GPU, 16 GB HBM2, and 32 GB RAM.

Performance Metrics: Our two metrics are the percentage of samples resolved locally and the overall cost of the DDNN:

$$C_{\text{DDNN}} = \sum_{m=1}^{M} \mathcal{I}_m \cdot C_L^m + (1 - \mathcal{I}_m) \cdot C_R^m, \tag{9}$$

where, for sample m, C_L^m and C_R^m are the local and remote exit costs, respectively, and $\mathcal{I}_m$ indicates whether the sample exited locally or not.

$$\mathcal{I}_m = \begin{cases} 1 & \text{if the sample m exited locally} \\ 0 & \text{else.} \end{cases} \tag{10}$$

We implement and compare the following models:

DeepCog: A fully centralized DNN based on 3D-CNN that was proposed in [11] for a similar resource allocation task. It also uses a pre-processing methodology and arranges the time series data into an appropriate "image"-like frames for input. We implement both as well. The model is entirely cloud-based; it lacks edge prediction capabilities, resulting in a local resolution percentage of zero. For quick reference, the architectures are summarized in Table I.

Centralized LSTM: A fully centralized version of our LSTM architecture without local exits, indicating that all data samples are processed and resolved within the cloud. Notably, this model does not incorporate joint training. Being exclusively cloud-based, it cannot make edge predictions, and consequently, the local resolution percentage is zero. It is important to note that this model is distinct from resolving all samples remotely in a DDNN, due to differences in their architecture and training methodologies.

Oracle-based DDNN: A DDNN with the offline optimal offloading policy that operates under the premise of having complete knowledge of both exits. This model, referred to as the "Oracle", is idealistic as it assumes perfect information, which is unrealistic in real-world scenarios. The offloading decision within this model hinges on comparing the transmission cost (C_T) with the differential cost between local (C_L) and remote (C_R) exits. As C_T increases, the model tends to favor local resolutions.

Random-based DDNN: A DDNN with random offloading policy where the offloading decision for each sample is modeled as an independent and identically distributed (i.i.d.) Bernoulli random variable with a success probability p. Consequently, with a constant p, the proportion of samples processed locally is effectively $\mathcal{L} = p$. Our simulations explore the implications of this Random policy across a range of p values within $[0, 1]$, delineating a spectrum of costs.

Confidence-based DDNN: A DDNN with the confidence-based offloading policy, which was explained in the previous section. The offloading policy is based on uncertainty. The uncertainty value U is compared to a confidence threshold (η). By increasing η, more samples are resolved locally.

ML-based DDNN: A DDNN with the ML-based offloading policy that was explained in the previous section. The offloading mechanism operates based on the output of the binary classifier. When C_T is modified, the classifier's behavior alters, consequently influencing the output decisions. By increasing C_T, more samples are resolved locally.

TABLE I: Models for Performance Comparison

Model	Joint Training	Edge Offloading	Cloud Offloading	Realizability
Centralized LSTM	×	×	✓	✓
Centralized DeepCog	×	×	✓	✓
Oracle-based DDNN	✓	✓	✓	×
Random-based DDNN	✓	✓	✓	✓
Confidence-based DDNN	✓	✓	✓	✓
ML-based DDNN	✓	✓	✓	✓

B. Experiments

Experiment 1 (Resource Allocation Trade-off): After training the model, we plot trade-off curves to illustrate total loss versus local sample resolution percentage. The model processes the test set, yielding output samples that are handled either locally or remotely based on the offloading mechanism, and calculates the total loss using Eq. (9). We obtain the offline oracle-based trade-off curve by using *Oracle-based Offloading*, where changing the transmission cost (C_T) alters local sample resolution rate. This curve is the lower bound baseline. The trade-off curve based on a random policy is derived by adjusting the probability parameter p from 0 to 1, as previously discussed. This approach establishes an upper bound baseline for performance evaluation. Any offloading policy that results in costs exceeding this curve is deemed ineffective[5]. By applying Bayesian *Confidence-based Offloading* and adjusting η (the confidence threshold) within the range of $[0, 1]$, we generate the online confidence-based trade-off curve. The online ML-based trade-off curve is generated by employing the *ML-based Offloading*, where increasing the transmission cost (C_T) modifies the curve.

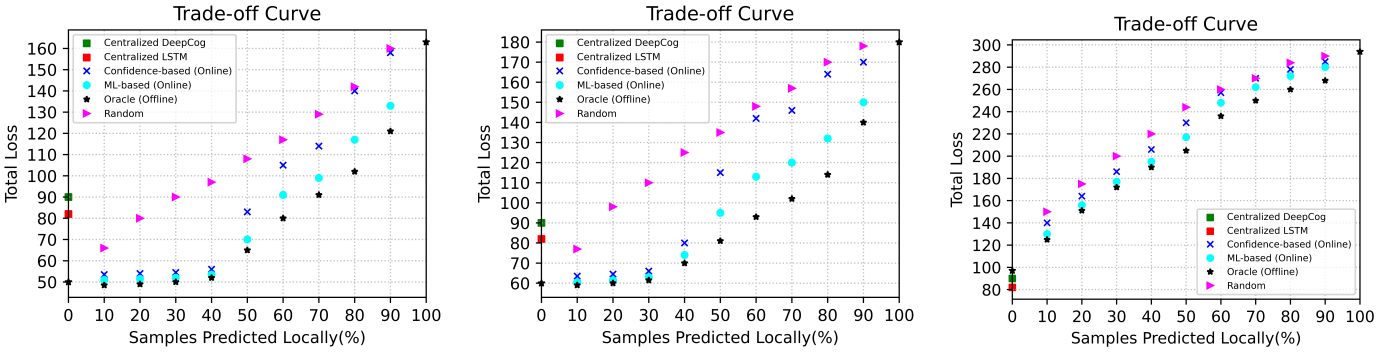
Figs. 3(a) and 3(b) show trade-off curves for models with training weights (w_L, w_R) set to $(0.9, 0.1)$ and $(0.8, 0.2)$ respectively. Notably, at the operational point where no samples are predicted locally on the x-axis, the LSTM architecture consistently outperforms the 3D-CNN-based model. More importantly, simply introducing a local exit during training improves the baseline performance by 20-50%, even though in this scenario all samples are also resolved remotely[6].

Key observation 1: The positive impact of incorporating local exits, even in a fully centralized DNN, as observed in past work for different DNN architectures and objectives, is also clearly evident for the problem at hand, despite its aforementioned differences detailed in the previous sections.

The comparison between the Oracle and Random offloading policies in both cases leads to a crucial insight: certain

[5]While it's theoretically possible to devise a less efficient offloading policy, such as consistently selecting the more costly exit, this would not yield a meaningful or practical benchmark for comparison.

[6]Our distributed model's superior performance compared to a centralized LSTM, achieved with lower overhead, is due to the local exit's influence on gradient flow. This effect, noted in other studies [7], [15], [19], shows that the local exit enhances operational efficiency and adds a regularization effect, which results in improved performance with reduced overhead, creating a beneficial "win-win" situation.



(a) $(w_L, w_R) = (0.9, 0.1)$ (b) $(w_L, w_R) = (0.8, 0.2)$ (c) $(w_L, w_R) = (0.1, 0.9)$

Fig. 3: Trade-off curves (Total loss vs Percentage of samples predicted locally) for three weight pairs

offline offloading strategies can surpass the performance of the Random policy. This observation underscores the potential for developing online offloading policies that closely approximate the efficacy of the Oracle, suggesting room for significant advancements in this area. Conversely, if the performance of the Random and Oracle policies were identical, it would imply a lack of opportunity for any online offloading policy to offer improvements.

Moreover, in both Figs. 3(a) and 3(b), we observe that both offloading mechanisms (confidence-based and ML-based) are able to achieve close-to-optimal performance when up to 40% of samples are resolved locally. However, in more challenging scenarios where a greater proportion of samples are processed locally, a deviation from the optimal bound is observed in both methods. Notably, the theoretically-motivated ML-based scheme consistently outperforms the heuristic confidence-based scheme across all scenarios and operating points.

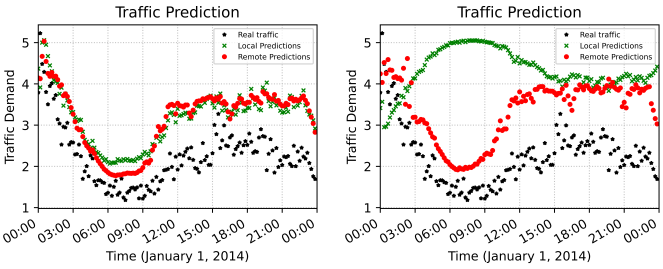
Key observation 2: The capability for online decision-making in determining which and how many samples to process locally offers very important performance trade-offs (e.g., resolving 40% of decisions locally "for free", *without any impact on the total provisioning cost*).

Key observation 3: The ML-based offloading mechanism consistently outperforms heuristics in all scenarios, as expected, since it stems as a solution of the optimization problem that the oracle knows beforehand.

In Fig. 3(c), where the training weights are set to $(w_L, w_R) = (0.1, 0.9)$, indicating very low strength on local layers (unlike in Figs. 3(a) and 3(b), where higher weights are given to local layers), the significant impact of weight selection on overall DDNN performance is evident. While pinpointing the optimal weight pair (in this case the pair $(0.9, 0.1)$) cannot be predetermined, our analysis across various scenarios consistently demonstrates the necessity of assigning higher weights to the local exit for optimal functioning.

Key observation 4: The selection of training weights (w_L, w_R) plays a pivotal role in the performance of the model.

Experiment 2 (SLA Violations Avoidance): Fig. 4 demonstrates the actual demand (**d**), local allocations ($\hat{\mathbf{y}}_L$), and remote allocations ($\hat{\mathbf{y}}_R$) for one of the base stations in the dataset under



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

Fig. 4: Traffic demand predictions for a base station using two weight pairs: Data forwarded without offloading mechanism

two distinct training weight sets: $(w_L, w_R) = (0.9, 0.1)$ and $(w_L, w_R) = (0.1, 0.9)$. To facilitate the generation of both local and remote predictions for all samples, the offloading block is deactivated. As observed in Fig. 4(a), the local exit performs effectively under the weight pair $(w_L, w_R) = (0.9, 0.1)$, enabling a significant number of allocations to be processed locally. Conversely, Fig. 4(b) reveals decreased efficiency in the local exit under the weight pair $(w_L, w_R) = (0.1, 0.9)$, resulting in higher costs for local allocations, which aligns with the trade-off curve presented in Fig. 3(c).

Key Observation 5: It is critical to prioritize the local weight over the remote weight $(w_L > w_R)$ to compensate for the simplicity/shallowness of the local module, enabling it to outperform the centralized DNN.

Key Observation 6: The models prioritize preventing SLA violations over matching demand (as an MSE objective would), aligning with the objective function's emphasis on the higher cost of under-provisioning.

Experiment 3 (Latency Reduction): As explained in Section IV-B, we assess whether our model reduces or increases latency by analyzing both "communication" and "computation" times. For **communication time**, we refer to a recent systems-oriented study [20], which estimates the average round-trip transmission time (RTT) from edge to cloud at 42.46 ms per sample. **Computation time** is evaluated by running each model multiple times on the same server and calculating the average

TABLE II: Latency Comparison (milliseconds per sample)

L (%)	5	20	40	50	60	80	95	Centralized LSTM	DeepCog
T (ms)	40.40	34.05	25.43	21.09	16.78	8.20	1.75	42.67	42.63

processing time per sample across various models[7].

Table II presents our findings, where "L" denotes the percentage of samples processed locally and "T" signifies the average time taken to resolve a single sample under each scenario. For instance, with 40% local resolution, "T" equates to the sum of {the average offloading block processing time per sample, 40% of the average local inference time per sample, 60% of the RTT, and 60% of average remote inference time per sample}. The data clearly indicate that an increase in local sample resolution leads to a decrease in inference latency. Notably, when 50% of samples are handled locally by the DDNN, it matches the cost of centralized baselines while achieving a 49% reduction in inference latency.

Key Observation 7: The inference latency diminishes as an increasing number of samples are processed locally.

VI. Conclusions and Future Work

In this study, we have developed and implemented a Distributed Deep Neural Network (DDNN) specifically tailored for forecasting future traffic demand in order to manage resource allocation in 5G networks. The proposed DDNN is characterized by its multiple exits architecture, encompassing a local exit (e.g., at the Edge) and a remote exit (e.g., in the Cloud). To ensure the DDNN fulfills its intended objectives, it undergoes a process of joint training, during which distinct weights are allocated to both the local and remote exits. We have demonstrated that a DDNN based on LSTMs is able to resolve a large amount of resource allocation decisions locally, with a lightweight component, greatly improving overall latency and communication footprint, compared to fully centralized architectures. We also demonstrated that properly designing the offloading mechanism that determines which samples to send to the cloud can outperform existing methods, getting closer to the theoretically optimal performance of an oracle. In the future, we plan to investigate hierarchical DDNN architectures with distributed multiple local components/exits that are responsible only for subsets of the decisions, while the cloud layers now also *aggregate* information from various local components.

VII. Acknowledgment

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

[7]It is important to note that the implementation setup for such a distributed architecture can vary. We have designed our experiment with practical values to effectively demonstrate the potential for latency reduction. We run the models on a server using an Nvidia V100 GPU, 16 GB HBM2, and 32 GB RAM.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[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] 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.

[8] 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.

[9] 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, No. 90, pp. 1-30, 2022.

[10] 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.

[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] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-Supervised Nets," Proceedings of the 18th International Conference Machine Learning Research (PMLR), vol. 38, pp. 562-570, USA, May 2015.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

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